BE SOFTWARE ARCHITECTURE WISE · CHAPITRE 6

Styles architecturaux II : systèmes distribués

Ce chapitre fait passer l'architecture à travers le réseau. Il explique quand la distribution d'un système vaut son coût, les illusions qui la rendent plus difficile qu'il n'y paraît, et les principaux styles distribués : l'architecture à base de services, l'architecture orientée services, les microservices, l'architecture pilotée par les événements, le serverless et l'architecture basée sur l'espace. Il traite ensuite des mécanismes dont ils dépendent tous : la communication synchrone et asynchrone et la conception d'API, les modèles de cohérence et le théorème CAP, ainsi que les transactions distribuées avec les sagas et le patron outbox. Il s'adresse aux développeurs et aux architectes qui conçoivent ou évaluent des systèmes composés de nombreux services, et il se montre précis sur les défaillances, les garanties de livraison et la cohérence.

bewiseapps.com/fr/software-architecture-wise/distributed-systems/ · Version 2026-11-03. Information pédagogique uniquement, pas un conseil professionnel.

EN UN COUP D'ŒIL

Vous apprendrez à

- Expliquer les raisons légitimes de distribuer un système et les coûts qu'entraîne chacune d'elles.
- Décrire les huit illusions de l'informatique distribuée et la défaillance partielle, et concevoir des réponses à y apporter.
- Calculer la disponibilité et la latence d'une chaîne d'appels synchrones, et proposer des changements qui les améliorent.
- Comparer l'architecture à base de services, la SOA classique et les microservices, et choisir entre elles pour une organisation donnée.
- Décider de la granularité des services en mettant en balance les facteurs de désintégration et d'intégration, et reconnaître le monolithe distribué.
- Distinguer événements, commandes et messages, topologies à courtier et à médiateur, et les quatre patrons d'événements de Fowler.
- Expliquer les garanties de livraison, pourquoi « exactement une fois » demande une définition rigoureuse, et comment fonctionnent les consommateurs idempotents.
- Choisir les styles de communication, les niveaux de cohérence et les patrons de transaction (sagas, outbox) pour les workflows distribués.

Avant de commencer

Les chapitres 0 à 5, en particulier la disponibilité en série et en parallèle (chapitre 2, section 2.5), le couplage et la loi de Conway (chapitre 3), ainsi que le quantum d'architecture, le monolithe modulaire et les contextes délimités (chapitre 5).

Durée

Environ 3 h 30 de lecture ; 45 minutes pour le test.

Termes clés

système distribué · défaillance partielle · illusions de l'informatique distribuée · couplage temporel · architecture à base de services · bus de services d'entreprise · microservices · déployabilité indépendante · base de données par service · architecture pilotée par les événements · livraison au moins une fois (at-least-once) · consommateur idempotent · Théorème CAP · saga

Pourquoi ce chapitre compte

Le chapitre 5 a décrit des styles dans lesquels l'ensemble du système est déployé comme une seule unité. Ce chapitre pose les mêmes questions à travers un réseau. Presque tous les systèmes d'une certaine taille sont aujourd'hui distribués d'une manière ou d'une autre : une application mobile dialogue avec des services backend, qui appellent des prestataires de paiement, des services de cartographie et des courtiers de messages, et les services eux-mêmes tournent sur de nombreuses machines. Découper un système peut apporter une mise à

l'échelle indépendante, l'isolation des pannes, des mises en production indépendantes et des équipes autonomes. Cela apporte aussi une catégorie de problèmes qui ne peuvent pas se produire à l'intérieur d'un seul processus : des messages perdus ou dupliqués, des services lents plutôt que morts, des données qui divergent d'une copie à l'autre, et des pannes sur des machines dont vous ignoriez dépendre.

L'idée la plus importante à retenir est que **la distribution est un échange, pas une amélioration**. Chaque avantage se paie en latence, en cohérence, en observabilité et en effort d'exploitation, et un bon architecte sait dire quel avantage est acheté, à quel prix, et pourquoi une option moins chère à l'intérieur d'un monolithe ne suffit pas. Les illusions de l'informatique distribuée et l'arithmétique des chaînes d'appels synchrones présentées dans la première section sont les outils qui permettent d'évaluer le prix de cet échange avant de s'y engager.

Comment les sections s'articulent

Le chapitre va du pourquoi au comment. La section 6.1 expose les raisons de distribuer, les huit illusions, la défaillance partielle et les coûts, et montre par un exemple commenté comment cinq services qui atteignent chacun 99,9 % de disponibilité donnent un chemin de requête à environ 99,5 % seulement. Elle se termine en revenant sur le quantum d'architecture, qui explique pourquoi les appels synchrones lient entre eux des services déployés séparément.

Les trois sections suivantes présentent les principaux styles distribués, chacun traité comme les styles du chapitre 5 : définition et origine, diagramme, fonctionnement, exemple, tableau des caractéristiques, quand l'utiliser et les anti-patrons habituels. La section 6.2 traite de l'architecture à base de services, un juste milieu pragmatique composé de quelques services de domaine à gros grain qui partagent souvent une base de données, et de l'architecture orientée services classique, avec ses bus de services d'entreprise et ses modèles canoniques, y compris les raisons de sa disgrâce. La section 6.3 traite des microservices : des services alignés sur les contextes délimités, la déployabilité indépendante, la base de données par service, les patrons d'appui comme les API gateways et les service meshes, la manière de choisir la granularité, ainsi que le monolithe distribué et d'autres anti-patrons. La section 6.4 traite de l'architecture pilotée par les événements : événements et commandes, topologies à courtier et à médiateur, partitions et groupes de consommateurs de Kafka, garanties de livraison et consommateurs idempotents.

La seconde moitié se tourne vers les mécanismes dont dépend tout style distribué. La section 6.5 compare la communication synchrone et asynchrone et le couplage que crée chacune, et traite de REST, gRPC, GraphQL et de la messagerie, des bonnes pratiques de conception d'API, du versionnage et des tests de contrat. La section 6.6 porte sur ce que peuvent attendre les lecteurs de données répliquées : les modèles de cohérence, de la linéarisabilité à la cohérence à terme, le théorème CAP et ce qu'il dit réellement, PACELC et les quorums. La section 6.7

explique pourquoi on évite la validation en deux phases entre services et comment les sagas, les transactions compensatoires et l'outbox transactionnelle maintiennent la justesse des processus métier sans elle. La section 6.8 conclut avec deux styles adaptés à des profils de charge particuliers : les fonctions serverless, qui descendent à zéro et sont facturées à l'invocation, et l'architecture basée sur l'espace, qui garde les données en mémoire pour survivre à des pics extrêmes comme l'ouverture d'une billetterie.

Les exemples récurrents et comment lire le chapitre

Deux exemples reviennent. La plateforme de livraison de repas des chapitres précédents (clients, restaurants, livreurs, paiements) illustre une conception distribuée ordinaire : elle apparaît sous la forme de cinq services de domaine, d'un ensemble de microservices, d'un flux d'événements `OrderPlaced` et d'une saga de commande. Un système de billetterie qui vend des places pour la tournée des stades d'un artiste montre ce qui change sous une charge extrême et en pics, quand des dizaines de milliers de personnes veulent les mêmes quelques places à la même seconde.

Le chapitre est de niveau intermédiaire à avancé. Il suppose acquis les chapitres 0 à 5, en particulier la disponibilité en série et en parallèle (chapitre 2), le couplage et la loi de Conway (chapitre 3), ainsi que le quantum d'architecture, le monolithe modulaire et les contextes délimités (chapitre 5). Les débutants peuvent lire d'abord le résumé d'ouverture de chaque section et les encadrés « Si vous débutez ». Les ingénieurs expérimentés trouveront de la précision là où elle compte : ce qu'implique une livraison au moins une fois (at-least-once) pour chaque consommateur, ce que « exactement une fois » signifie et ne signifie pas, et pourquoi la cohérence à terme doit être prise en compte dès la conception plutôt que simplement tolérée.

Ce qu'il faut retenir pour la suite

Trois habitudes traversent toutes les sections. Premièrement, comptez les dépendances synchrones, pas les services : le nombre d'éléments qui doivent tous fonctionner au même moment pour que la requête d'un utilisateur aboutisse détermine la disponibilité et la latence. Deuxièmement, partez du principe que les messages sont perdus, retardés, dupliqués et réordonnés, et concevez des récepteurs idempotents. Troisièmement, décidez à qui appartient chaque donnée, car la propriété détermine où les transactions sont possibles et où la cohérence doit être gérée entre services. Le chapitre 7 développe cette troisième habitude en un traitement complet de l'architecture des données.

SECTION 6.1 SUR 6.8 · INTERMÉDIAIRE · 26 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Pourquoi distribuer, et les illusions de l'informatique distribuée

AVANT DE LIRE

Cette section s'appuie sur :

- 0.4 Comment les ordinateurs communiquent
- 2.5 Disponibilité et fiabilité
- 5.1 Styles et patrons

Termes déjà vus qu'elle utilise : Quantum d'architecture (architecture quantum), Latence, Disponibilité, Loi de Conway, Monolithe modulaire

L'IDÉE EN BREF

Imaginez un petit restaurant où un seul cuisinier fait tout dans une seule cuisine. C'est simple : si le cuisinier est malade, le restaurant est fermé, mais rien ne se perd entre plusieurs personnes. Imaginez maintenant une chaîne avec une boulangerie centrale, une cuisine à sauces et de nombreuses succursales reliées par des camionnettes de livraison. Elle peut servir bien plus de clients, et une cuisine peut fermer sans entraîner la fermeture des autres. Mais les camionnettes restent coincées dans les embouteillages, des commandes se perdent, et quand une livraison n'arrive pas, personne ne sait si la boulangerie n'a jamais reçu la commande ou si la camionnette est tombée en panne. Les logiciels distribués fonctionnent comme la chaîne.

CE QU'IL FAUT SAVOIR

- Un système distribué est un ensemble de programmes, sur des machines différentes, qui coopèrent en échangeant des messages sur un réseau pour fournir ce que les utilisateurs perçoivent comme un seul service.
- Les raisons légitimes de distribuer sont la montée en charge, l'isolation des pannes, la déployabilité indépendante, l'autonomie des équipes, la géographie et la résidence des données, des besoins technologiques différents et l'isolation de sécurité ; chacune a, à l'intérieur d'un monolithe, un substitut partiel moins coûteux qu'il faut envisager en premier.
- La défaillance partielle est le problème qui définit le domaine : après un délai d'expiration, l'appelant ne peut pas savoir si la requête a été perdue, si l'appelé est tombé en panne ou si seule la réponse a été perdue ; les opérations retentées doivent donc être idempotentes.

- Les huit illusions de l'informatique distribuée (Deutsch et ses collègues de Sun, plus la huitième de Gosling) sont de fausses hypothèses sur la fiabilité, la latence, la bande passante, la sécurité, la topologie, l'administration, le coût du transport et l'homogénéité.
- La distribution coûte en latence, en observabilité, en cohérence et en effort d'exploitation, et ces coûts augmentent à peu près avec le nombre d'unités déployables.
- Dans une chaîne synchrone, les disponibilités se multiplient et les latences s'additionnent : cinq services à 99,9 % donnent environ 99,5 %, soit à peu près 3,6 heures d'indisponibilité par mois.
- Les appels synchrones lient les services entre eux à l'exécution (couplage temporel) : le nombre de sauts synchrones dans les parcours clés compte donc davantage que le nombre de services.

Ce qui rend un système distribué

Un système distribué est un ensemble de programmes, s'exécutant sur des ordinateurs différents, qui coopèrent en échangeant des messages sur un réseau pour fournir ce que les utilisateurs perçoivent comme un seul service. Votre application de livraison de repas en est un : le téléphone, les services de commandes et de paiement de l'entreprise, la tablette du restaurant, le téléphone du livreur et un processeur de paiement par carte jouent tous un rôle dans une même commande. Le chapitre 5 a décrit des styles déployés comme une seule unité (les monolithes) ; ce chapitre porte sur les styles dont les parties sont déployées séparément et communiquent à travers le réseau.

Leslie Lamport, qui a plus que quiconque contribué à donner au domaine des fondements mathématiques, en a résumé la difficulté essentielle dans un e-mail interne de mai 1987 : un système distribué est un système dans lequel «la défaillance d'un ordinateur dont vous ignoriez jusqu'à l'existence» peut rendre votre propre ordinateur inutilisable. La boutade fonctionne parce qu'elle est vraie. Dès qu'un système dépend de machines qu'il ne contrôle pas et ne voit pas, toute une catégorie de pannes apparaît, qui ne peuvent tout simplement pas se produire à l'intérieur d'un seul processus.

C'est pourquoi la distribution est une décision qui a un prix, et non un signe de maturité. La suite de cette section expose les raisons qui justifient ce prix, les fausses hypothèses qui conduisent à le sous-estimer, et une manière d'en calculer une partie avant de payer.

Pourquoi distribuer

Il existe une poignée de raisons légitimes de répartir un système entre plusieurs processus et machines. Chacune correspond à un attribut de qualité du chapitre 2 qu'une seule unité de déploiement peine à fournir au-delà d'une certaine taille. Chacune a aussi, à l'intérieur d'un

monolithe, un substitut partiel moins coûteux, et un bon architecte examine d'abord ce substitut.

Les raisons de distribuer, et ce qu'un monolithe bien construit sait déjà faire

Raison	Ce qu'offre la distribution	Ce qu'un déploiement unique sait déjà faire
Montée en charge	Des parties soumises à des charges très différentes peuvent être mises à l'échelle séparément : le flux de positions des livreurs reçoit des milliers de mises à jour par seconde, les écrans d'inscription des restaurants quelques-unes par heure.	Exécuter de nombreuses répliques identiques derrière un équilibreur de charge, ajouter des répliques en lecture et des caches. Cela suffit pour la plupart des systèmes, et coûte bien moins cher.
Isolation des pannes	Une fuite de mémoire dans les recommandations ne peut pas faire tomber l'étape de paiement, car elles s'exécutent dans des processus différents, sur des machines différentes.	Des pools de threads séparés et des cloisonnements (bulkheads) à l'intérieur d'un processus donnent une isolation partielle ; un plantage fait quand même tout tomber.
Déployabilité indépendante	Une équipe peut mettre sa partie en production vingt fois par jour sans coordonner de mise en production avec qui que ce soit.	Un monolithe modulaire doté d'une bonne automatisation peut aussi être mis en production plusieurs fois par jour, mais chaque mise en production livre tout.
Autonomie des équipes	Les équipes sont responsables d'un service de bout en bout, choisissent leur rythme et en assurent l'astreinte ; la loi de Conway joue en votre faveur au lieu de jouer contre vous.	La responsabilité par module fonctionne bien jusqu'à quelques équipes ; au-delà, un build et une mise en production partagés deviennent une file d'attente.
Géographie et résidence des données	Exécuter des parties près des utilisateurs pour réduire la latence, ou conserver les données à caractère personnel dans une seule juridiction parce qu'une loi ou un contrat l'exige.	Déployer une copie complète par région (une cellule). C'est en soi une forme de distribution, mais une forme simple.
Besoins technologiques différents	Un modèle de recommandation a besoin de GPU, la recherche d'un moteur de recherche, une intégration avec un système hérité d'un environnement d'exécution particulier.	Appeler le composant spécialisé et garder tout le reste ensemble.
Isolation de sécurité	Confier les données de carte à un petit service distinct, renforcé et audité séparément, réduit le périmètre d'une évaluation PCI DSS.	Pas vraiment : tout ce qui se trouve dans le processus peut atteindre n'importe quoi d'autre en mémoire.

Deux observations découlent de ce tableau. Premièrement, les raisons les plus fortes sont souvent organisationnelles : la déployabilité indépendante et l'autonomie des équipes comptent surtout quand de nombreuses équipes travaillent sur un même produit, ce qui explique pourquoi la loi de Conway (chapitre 3) revient sans cesse dans ce chapitre. Deuxièmement, la distribution est rarement du tout ou rien. Beaucoup de systèmes réussis sont un monolithe modulaire accompagné de deux ou trois services séparés, chacun détaché pour une raison précise et nommée.

La défaillance partielle : le problème qui définit le domaine

À l'intérieur d'un processus, un appel de fonction soit retourne, soit le processus entier meurt avec lui. À travers un réseau, il existe une troisième issue, et elle change tout : la défaillance partielle, dans laquelle certaines parties du système fonctionnent tandis que d'autres sont en défaillance, lentes ou injoignables. Pire, l'appelant ne peut souvent pas savoir lesquelles.

Prenons le service des commandes qui demande au service de paiement d'autoriser £24,50. Si aucune réponse n'arrive avant le délai d'expiration, le service des commandes ne sait pas si la requête s'est perdue en chemin, si le service de paiement a planté ou s'est bloqué avant ou après avoir débité la carte, ou si la carte a été débitée et que seule la réponse s'est perdue. Du côté de l'appelant, les trois situations sont identiques.

Trois défaillances qui se ressemblent



Pour l'appelant, les trois sont identiques : pas de réponse avant le timeout.

Après un délai d'expiration, l'appelant ne peut pas savoir si le travail a été effectué ; seules des nouvelles tentatives idempotentes ou une vérification ultérieure de l'état permettent de trancher.

Les conséquences pratiques façonnent toute conception distribuée. Une nouvelle tentative pourrait débiter le client deux fois : les opérations susceptibles d'être retentées doivent donc pouvoir être répétées sans risque. Cette propriété est l'idempotence, et on l'obtient

généralement en donnant à chaque requête une clé unique que le récepteur mémorise. Une dépendance lente est souvent pire qu'une dépendance morte, car les appelants continuent d'attendre et immobilisent des threads et des connexions ; quand un service ralentit, ses appelants épuisent leurs ressources et ralentissent à leur tour : c'est une défaillance en cascade. Et comme un réseau peut se scinder de sorte que deux groupes de machines ne puissent plus se joindre tout en continuant de fonctionner, des données conservées aux deux endroits peuvent diverger : c'est le sujet de la section sur CAP, plus loin dans ce chapitre.

Il n'existe pas non plus d'horloge globale. L'horloge de chaque machine dérive et est corrigée séparément : les horodatages de deux services ne peuvent donc pas être comparés en toute confiance, et « qu'est-ce qui s'est passé en premier ? » peut être une question difficile. L'article de Lamport de 1978 sur le temps et les horloges dans les systèmes distribués a montré comment ordonner des événements sans s'appuyer sur des horloges physiques ; pour les architectes, la leçon est simplement de ne pas fonder la justesse sur la comparaison d'heures murales entre machines.

SI VOUS DÉBUTEZ

Pas à pas : quand une réponse n'arrive jamais

L'idée qui demande le plus d'attention ici est la défaillance partielle. À l'intérieur d'un programme, un appel à une autre partie du programme soit se termine, soit le programme entier s'arrête. À travers un réseau, il existe une troisième possibilité : vous demandez, et aucune réponse ne revient.

- Le service des commandes demande au service de paiement de prélever £24,50 sur la carte d'un client, et attend la réponse.
- Il ne peut pas attendre indéfiniment ; il abandonne donc après une durée fixée, appelée délai d'expiration (timeout) : disons deux secondes. Les deux secondes passent sans réponse.
- Trois choses différentes ont pu se produire, et du côté du service des commandes elles se ressemblent exactement : la requête s'est perdue en chemin et rien n'a été débité ; le service de paiement a planté ou s'est bloqué, et personne ne sait s'il a débité la carte ; ou la carte a été débitée et seule la réponse s'est perdue.
- Si le service des commandes réessaie simplement, il risque de débiter le client deux fois. S'il abandonne, le client a peut-être payé une commande qui n'a jamais été passée.
- Les issues sûres consistent à rendre la requête idempotente, c'est-à-dire que l'envoyer deux fois a le même effet que l'envoyer une fois, ou à demander ensuite au service de paiement ce qui s'est réellement passé. L'idempotence fonctionne généralement en envoyant un numéro de requête unique avec le débit : le service de paiement mémorise les numéros qu'il a traités et ne débite jamais deux fois le même.
- **La leçon :** sur un réseau, le silence n'est pas une réponse. Chaque appel distant a besoin d'un délai limite et d'un plan pour l'incertitude, c'est pourquoi ce chapitre revient sans cesse sur les nouvelles tentatives et les doublons.

Les huit illusions de l'informatique distribuée

Les illusions de l'informatique distribuée sont huit hypothèses que font les personnes qui découvrent les systèmes distribués ; elles sont toutes fausses et finissent toutes par causer des problèmes. La liste vient de Sun Microsystems : L Peter Deutsch en a établi sept en 1994, en s'appuyant sur quatre que des collègues, dont Bill Joy, avaient déjà relevées, et James Gosling, le créateur de Java, a ajouté la huitième vers 1997. Trois décennies plus tard, chacune décrit encore, cette semaine, une vraie panne quelque part.

Les huit illusions, avec une conséquence et une réponse de conception pour chacune

Illusion	Ce qui tourne mal quand on y croit	Que faire à la place
1. Le réseau est fiable	Un appel de paiement reste bloqué ; le thread appelant attend indéfiniment ; sous charge, tous les threads sont vite en attente et le service des commandes cesse de répondre à qui que ce soit.	Fixer un délai d'expiration pour chaque appel distant, ne retenter que les opérations idempotentes, placer des files là où le travail peut attendre (patrons de résilience).
2. La latence est nulle	Une page de panier qui faisait 40 appels internes au processus fait désormais 40 appels réseau à 2 ms chacun ; 80 ms surgissent de nulle part, et bien plus en queue de distribution.	Concevoir des interfaces à gros grain, regrouper les requêtes, appeler en parallèle les services indépendants, mettre en cache les données stables.
3. La bande passante est infinie	Chaque événement `MenuUpdated` transporte toutes les photos des plats ; une soirée chargée sature la liaison et retarde les mises à jour de position qui la partagent.	Envoyer des identifiants et les champs dont les consommateurs ont besoin, paginer, compresser, garder les données volumineuses à côté du code qui les traite.
4. Le réseau est sécurisé	Les services à l'intérieur du centre de données acceptent n'importe quelle requête sans authentification ; un seul conteneur compromis peut appeler l'API de remboursement.	Chiffrer et authentifier le trafic entre services (par exemple avec TLS mutuel) et autoriser chaque appel : zero trust.
5. La topologie ne change pas	Une adresse IP codée en dur dans la configuration cesse de fonctionner quand le conteneur est replanifié sur une autre machine.	Utiliser la découverte de services et des noms DNS, et s'attendre à ce que des instances apparaissent et disparaissent à chaque minute.
6. Il y a un seul administrateur	Un fournisseur de cloud, l'équipe pare-feu d'un partenaire ou un autre service de l'entreprise modifie une règle, renouvelle un certificat ou abaisse une limite de débit, et vos appels commencent à échouer.	Savoir qui contrôle chaque saut, superviser les dépendances qui ne vous appartiennent pas, convenir de contrats et de préavis de changement.
7. Le coût du transport est nul	La sérialisation consomme du CPU, et les données qui traversent des zones de disponibilité, des régions ou l'internet public sont facturées au gigaoctet ; la facture réseau augmente plus vite que le trafic.	Mesurer le coût des flux de données, garder le trafic bavard en local, choisir des formats compacts là où le volume est élevé.

Illusion	Ce qui tourne mal quand on y croit	Que faire à la place
8. Le réseau est homogène	Un service suppose que chaque client parle la même version de protocole, utilise le même encodage de caractères et la même horloge ; un réseau mobile, l'ancien système d'un partenaire ou un proxy contredit cette hypothèse.	Utiliser des formats et des protocoles standard, explicitement versionnés, et tester l'interopérabilité avec les vrais clients.

D'OÙ CELA VIENT

La liste ne cesse de s'allonger

Dans une interview de 2021, Deutsch a approuvé une neuvième illusion proposée, qu'il a décrite comme une extension de la quatrième : croire que l'interlocuteur avec lequel vous communiquez est digne de confiance. Mark Richards et Neal Ford, dans la deuxième édition de *Fundamentals of Software Architecture* (2025), en ont proposé trois autres, tirées de projets de microservices : le versionnage est facile, les mises à jour compensatoires fonctionnent toujours, et l'observabilité est facultative. Toutes trois reviennent plus loin dans ce chapitre : le versionnage dans communication et API, la compensation dans les sagas, et l'observabilité ci-dessous.

Ce que coûte la distribution

Chaque raison de distribuer se paie dans la même monnaie. La **latence** est le premier coût. Un appel de fonction à l'intérieur d'un processus prend quelques nanosecondes ; un aller-retour réseau entre machines du même centre de données prend généralement de quelques centaines de microsecondes à une milliseconde ou plus, avant toute sérialisation ou tout traitement. Entre continents, la physique fixe un plancher : la lumière dans une fibre optique voyage à environ 200 000 km par seconde, si bien qu'un aller-retour Londres–New York (environ 5600 km dans chaque sens) ne peut pas prendre moins d'environ 56 ms. L'arithmétique de la latence de queue du chapitre 2 s'applique à chaque saut.

L'**observabilité** est le deuxième. Dans un monolithe, une trace de pile raconte toute l'histoire d'une requête en échec. Dans un système distribué, l'histoire est éparpillée dans les journaux de nombreux services sur de nombreuses machines. L'observabilité, c'est-à-dire la capacité à comprendre ce que fait le système à partir de ce qu'il produit, doit être prévue dès la conception : identifiants de corrélation transmis à chaque appel, journaux centralisés, métriques par service et traçage distribué (chapitre 9).

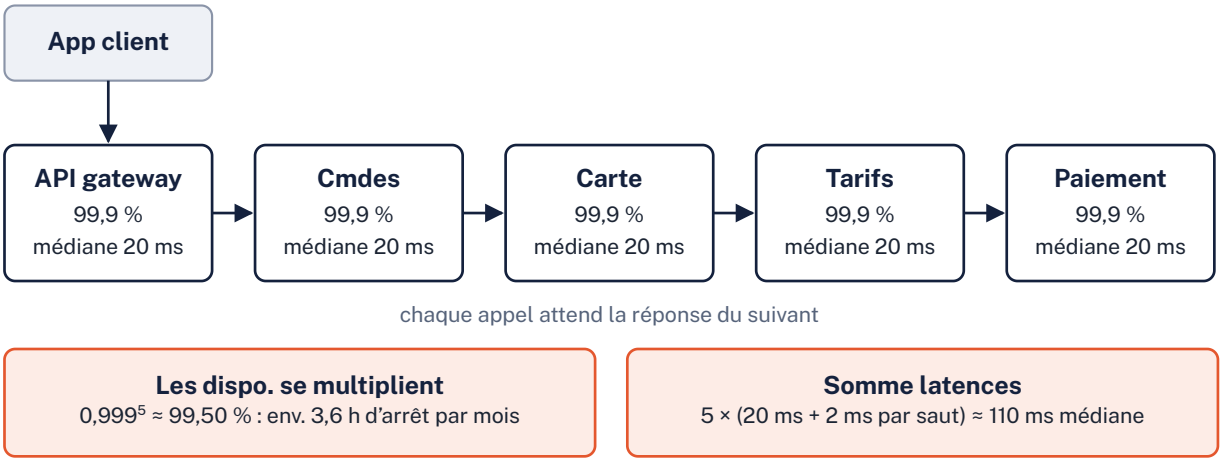
La **cohérence** est le troisième. Une seule transaction de base de données peut mettre à jour une commande et son paiement de façon atomique. Quand la commande et le paiement vivent dans des services différents avec des bases de données différentes, il n'y a pas de transaction partagée ; le système doit accepter une période pendant laquelle les deux divergent et s'assurer qu'ils convergent. Les sections suivantes présentent les outils : les modèles de cohérence, ainsi que les sagas et le patron outbox.

La **charge d'exploitation** est le quatrième, et celui que l'on sous-estime le plus souvent. Chaque service a besoin d'un pipeline de build, d'un déploiement, d'une configuration, de secrets, d'une supervision, d'alertes, d'une planification de la capacité, de correctifs de sécurité et de quelqu'un d'astreinte. Les tests changent aussi : une modification n'est sûre que si chaque consommateur de son interface fonctionne encore, ce qui demande des tests de contrat et des changements rétrocompatibles. Ces coûts sont à peu près proportionnels au nombre d'unités déployables, c'est pourquoi le nombre de services compte.

Faire le calcul : les chaînes synchrones

Le moment le moins coûteux pour découvrir qu'une conception distribuée est trop fragile ou trop lente, c'est sur le papier. L'outil est la règle des éléments en série du chapitre 2 : si une requête a besoin que chaque composant d'une chaîne fonctionne, la disponibilité de la chaîne est le produit des disponibilités des composants, et sa latence est à peu près la somme de leurs latences.

Une chaîne synchrone de cinq services



Cinq services qui atteignent chacun 99,9 % produisent un chemin de requête qui n'atteint qu'environ 99,5 %, avec cinq fois la latence d'un seul.

EXEMPLE COMMENTÉ

Exemple commenté : le budget de disponibilité et de latence d'une chaîne d'appels

Une plateforme de livraison de repas traite **Commander** au moyen d'une chaîne synchrone : l'API gateway appelle le service des commandes, qui appelle le service de la carte, qui appelle la tarification, qui appelle le paiement. Chaque service, pris isolément, atteint une disponibilité de 99,9 % (mesurée dans le temps, sur un mois de 30 jours soit 43 200 minutes), répond avec une médiane de 20 ms pour son propre travail et a un p99 de 250 ms. Chaque saut réseau ajoute environ 2 ms. Supposons, avec générosité, que les défaillances sont indépendantes et que toute défaillance fait échouer la requête entière.

Un service	Disponibilité 0,999 = 99,9 % ; indisponibilité autorisée 0,001 × 43 200 = 43,2 minutes par mois.
Deux en série	0,999 × 0,999 = 0,998 (99,8 %) ; environ 86 minutes par mois.
Cinq en série	$0{,}999^5 \approx 0{,}99501$ (99,50 %) ; indisponible environ 0,005 × 43 200 ≈ 216 minutes, soit 3,6 heures, par mois. Cinq services respectables chacun pris isolément ont produit un système à deux neuf et demi.
Dix en série	$0{,}999^{10} \approx 0{,}99004$ (99,0 %) ; environ 430 minutes, plus de 7 heures, par mois.
Latence médiane, cinq en série	Les latences s'additionnent : environ 5 × (20 ms + 2 ms) ≈ 110 ms, contre 20 ms pour le même travail effectué dans un seul processus. (À strictement parler, les médianes ne s'additionnent pas exactement ; la somme est une bonne première estimation.)
Latence de queue, cinq en série	La probabilité qu'une requête rencontre au moins un saut dans son 1 % le plus lent est de $1 - 0{,}99^5 \approx 4{,}9$ % : environ 1 requête sur 20 est donc ralentie par au moins un saut à 250 ms. Avec dix sauts, elle est de $1 - 0{,}99^{10} \approx 9{,}6$ %.
Ce qu'il faut pour garder 99,9 % de bout en bout	Chacun des cinq services aurait besoin d'environ 99,98 % ($0{,}9998^5 \approx 0{,}9990$) : environ 8,6 minutes d'indisponibilité par mois chacun, ce qui, en pratique, suppose une reprise automatisée.
Réponses de conception	Appeler la carte et la tarification en parallèle (la latence devient celle du plus lent des deux, et non la somme) ; mettre en cache les données de la carte dans le service des commandes ; rendre la confirmation du paiement asynchrone, pour que la commande soit acceptée puis payée peu après ; supprimer les sauts qui ne font que transmettre des requêtes.

Le chapitre 2 a présenté la règle selon laquelle les disponibilités se multiplient en série ; c'est avec la distribution que cette règle commence à mordre, car chaque appel réseau ajoute un composant à la série.

L'hypothèse d'indépendance flatte le résultat. Les services qui partagent une base de données, un cluster, un système de configuration ou un train de mises en production tombent en panne ensemble, et un appelé lent peut épuiser les threads de l'appelant et propager la panne vers le haut.

Le remède consiste rarement à rendre chaque service héroïquement fiable. Il consiste à raccourcir les chaînes synchrones et à laisser les parties qui n'ont pas besoin de répondre immédiatement travailler à partir d'une file.

Le quantum d'architecture, revisité

Le chapitre 5 a présenté le quantum d'architecture : une partie déployable indépendamment, à forte cohésion fonctionnelle, données comprises, à laquelle s'applique un même ensemble de caractéristiques d'architecture (Styles architecturaux, patrons et tactiques). Dans *Software Architecture: The Hard Parts* (2021), Ford, Richards, Sadalage et Dehghani affinent l'idée avec deux types de couplage. Le couplage statique est ce qui doit être présent pour qu'une partie puisse être déployée et exécutée : son code, ses bibliothèques, sa base de données et son infrastructure. Le couplage dynamique est la manière dont les parties dépendent les unes des autres pendant l'exécution : si elles s'appellent de façon synchrone ou asynchrone, et comment elles se coordonnent et restent cohérentes.

L'exemple commenté montre pourquoi l'aspect dynamique compte. Pendant toute la durée d'un appel synchrone, l'appelant ne peut pas terminer son travail si l'appelé n'est pas disponible et rapide ; les deux sont liés sur le plan opérationnel, et leurs disponibilités et latences se combinent comme calculé ci-dessus. Cinq services déployés séparément dans une chaîne synchrone constituent cinq unités de déploiement, mais pour la disponibilité de **Commander**, ils se comportent comme une seule unité fragile. Cette dépendance au fait que l'autre partie soit disponible au même moment est le couplage temporel. La messagerie asynchrone en supprime une grande partie : si le service des commandes dépose un message dans une file durable, le service de paiement peut être indisponible pendant une minute puis rattraper son retard, et le client obtient quand même un numéro de commande. Le prix à payer est que le client n'obtient pas immédiatement de réponse définitive, ce qui est autant une décision produit qu'une décision technique.

C'est la leçon la plus profonde de la section. Tracer les frontières des services, c'est la moitié de la conception ; décider quelles interactions doivent être synchrones, et garder cet ensemble réduit, c'est l'autre moitié.

ATTENTION

Erreurs courantes au moment de décider de distribuer

- **Distribuer pour corriger un problème de qualité du code.** Un monolithe enchevêtré découpé en services devient un système distribué enchevêtré, avec les pannes réseau en plus. Corrigez d'abord les frontières ; le monolithe modulaire du chapitre 5 est la voie habituelle.
- **Traiter les appels distants comme des appels locaux.** Les bibliothèques qui font ressembler un appel distant à un appel de méthode masquent la latence et les défaillances ; la « première loi de la conception d'objets distribués » de Martin Fowler consistait simplement à ne pas distribuer ses objets.
- **Compter les services au lieu des sauts synchrones.** Dix services qui échangent surtout des événements peuvent être plus robustes que trois qui s'appellent en chaîne.
- **Supposer que les défaillances sont indépendantes.** Des services qui partagent une base de données, un cluster Kubernetes ou un magasin de configuration tombent en panne ensemble ; la formule en série devient alors optimiste et la formule en parallèle relève de la fiction.
- **Ignorer le coût humain.** Dix services demandent dix jeux de pipelines, de tableaux de bord et de tours d'astreinte. Si une seule équipe de six personnes en est responsable, le principal avantage de la distribution, l'autonomie des équipes, est absent alors que tous les coûts demeurent.

Liste de contrôle : avant de distribuer

Utilisez ces questions avant de répartir un système, ou une partie de système, à travers le réseau.

- [ ] **Nommer le facteur**
  Quelle raison du tableau ci-dessus s'applique, à quelle partie, avec quelles preuves ? « Les microservices, c'est moderne » n'est pas un facteur.
- [ ] **Essayer d'abord la solution la moins coûteuse**
  Des répliques, du cache, une réplique en lecture, un processus d'arrière-plan ou une frontière de module plus nette à l'intérieur du monolithe pourraient-ils répondre au besoin ?
- [ ] **Dessiner le graphe d'appels**
  Compter les sauts synchrones dans chaque parcours utilisateur clé et calculer le budget de disponibilité et de latence, comme dans l'exemple commenté.
- [ ] **Décider à qui appartiennent quelles données**
  Chaque donnée a besoin d'un seul service propriétaire ; tout ce qui doit changer de façon atomique avec elle appartient probablement au même service.
- [ ] **Concevoir pour la défaillance partielle**
  Délais d'expiration, nouvelles tentatives avec idempotence, solutions de repli et ce que voit l'utilisateur quand une dépendance est indisponible, décidés avant la première panne.
- [ ] **Vérifier les prérequis**
  Le provisionnement et le déploiement automatisés, la supervision, les journaux centralisés et le traçage distribué doivent exister, ou être planifiés et financés.
- [ ] **Aligner les équipes sur les frontières**
  Chaque service doit avoir une seule équipe propriétaire, capable de le modifier, de le déployer et de l'assurer sans rien demander aux autres.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Un service appelle un service de stock sans délai d'expiration. Le service de stock devient très lent au lieu de tomber en panne. Pourquoi cela peut-il être pire que s'il avait planté ?**

Un service planté échoue vite et ses appelants peuvent réagir ; un service lent fait attendre ses appelants, si bien que leurs threads et leurs connexions se remplissent jusqu'à ce qu'eux aussi cessent de répondre, ce qui propage la panne sous forme de défaillance en cascade.

▼ **Trois services ont chacun une disponibilité de 99,95 % et sont appelés en série à chaque requête. Quelle est la disponibilité approximative du chemin de requête ?**

$0{,}9995^3 \approx 0{,}9985$, soit environ 99,85 %, en supposant des défaillances indépendantes.

▼ **Pourquoi l'illusion « le coût du transport est nul » compte-t-elle même quand la latence et la bande passante sont satisfaisantes ?**

Déplacer des données coûte du CPU pour la sérialisation et de l'argent pour le trafic entre zones, régions ou fournisseurs : une conception bavarde ou volumineuse peut donc coûter cher même quand elle est assez rapide.

▼ **Pourquoi cinq services déployés séparément peuvent-ils quand même se comporter comme une seule unité du point de vue de la disponibilité ?**

Si une requête les appelle de façon synchrone en chaîne, les cinq doivent être disponibles au même moment ; leurs disponibilités se combinent, ils sont donc couplés temporellement malgré des déploiements séparés.

▼ **À vous : L'action « Commander » appelle quatre services de façon synchrone, chacun disponible à 99,9 %. Calculez la disponibilité et l'indisponibilité mensuelle (43 200 minutes). Si la notification et l'affectation deviennent asynchrones et qu'il en reste deux synchrones, qu'est-ce qui change ? Que devez-vous ajouter si une nouvelle tentative suit un délai d'expiration (timeout) ?**

Quatre : 0,999^4 = 0,99601, soit environ 99,6 %, ou 173 minutes par mois. Deux : 0,999^2 = 0,99800, soit environ 86 minutes, à peu près la moitié, en supposant des défaillances indépendantes. Après un délai d'expiration, l'appelant ne peut pas savoir si le travail a été fait : les appels réessayés doivent donc être idempotents, par exemple avec une clé d'idempotence, sinon un client pourrait être débité deux fois. Les étapes asynchrones ont besoin de files supervisées, mais elles n'arrêtent plus la commande.

SOURCES

- Leslie Lamport: the 1987 email defining a distributed system (lamport.org): https://lamport.org/pubs/distributed-system.txt
- The Eight Fallacies of Distributed Computing (James Gosling's page): https://nighthacks.com/jag/res/Fallacies.html
- Software Engineering Radio episode 470: L Peter Deutsch on the fallacies of distributed computing (2021): https://se-radio.net/2021/07/episode-470-l-peter-deutsch-on-the-fallacies-of-distributed-computing/
- Martin Fowler: Microservices and the First Law of Distributed Objects: https://martinfowler.com/articles/distributed-objects-microservices.html
- Google SRE book: Addressing Cascading Failures: https://sre.google/sre-book/addressing-cascading-failures/

Architecture à base de services et architecture orientée services

AVANT DE LIRE

Cette section s'appuie sur :

- 0.5 Stocker les données
- 5.3 Monolithe modulaire
- 6.1 Pourquoi distribuer

Termes déjà vus qu'elle utilise : ACID, Schéma, Quantum d'architecture (architecture quantum), Partitionnement par domaine, Contexte délimité (bounded context)

L'IDÉE EN BREF

Pensez à un hôpital devenu trop grand pour un seul bâtiment. Il pourrait tout éclater en des centaines de minuscules cliniques, chacune avec son propre accueil et son propre service des dossiers, ou bien s'installer en cinq départements dans des ailes séparées : maternité, chirurgie, consultations externes, pharmacie, urgences. Chaque aile a son propre personnel et peut être rénovée sans fermer les autres, mais elles partagent toujours un service central des dossiers. C'est cela, l'architecture à base de services : quelques grandes parties séparées, chacune chargée de tout un domaine de l'activité, qui partagent souvent encore une même base de données.

CE QU'IL FAUT SAVOIR

- L'architecture à base de services (ainsi nommée par Richards et Ford) découpe un système en quelques services de domaine à gros grain déployés séparément, généralement de 4 à 12, qui partagent le plus souvent une même base de données et se trouvent souvent derrière une couche API.
- Comme la plupart des workflows restent à l'intérieur d'un seul service de domaine, ils peuvent utiliser des transactions ACID locales, et les services s'appellent rarement entre eux, ce qui garde les chaînes synchrones courtes.
- Une base de données partagée couple les changements de schéma et lie les services en un seul quantum ; des schémas par service, de petites bibliothèques d'entités par domaine et des bases de données séparées là où c'est justifié réduisent ce couplage.

- La SOA classique pilotée par l'orchestration organisait les services en couches techniques autour d'un bus de services d'entreprise, avec un modèle de données canonique et les standards WS-*, en visant la réutilisation à l'échelle de l'entreprise.
- La SOA a déçu parce que la réutilisation a créé du couplage, que le bus est devenu un goulet d'étranglement détenu par une seule équipe, que les modèles canoniques ne convenaient à personne et que les standards étaient lourds.
- Les microservices conservent des contrats de service explicites mais placent la logique dans les services (smart endpoints, dumb pipes), s'organisent par capacité métier et donnent à chaque service ses propres données.
- L'architecture à base de services est un juste milieu pragmatique pour quelques domaines et quelques équipes, et une étape peu risquée pour sortir d'un monolithe.

Entre le monolithe et les microservices

Une grande partie des discussions du secteur présente un choix entre un seul monolithe et de nombreux microservices. En pratique, une large part des systèmes réels se situent entre les deux : une poignée de services déployés séparément, chacun couvrant tout un domaine d'activité, qui partagent souvent encore une même base de données. Mark Richards et Neal Ford ont nommé ce style architecture à base de services dans *Fundamentals of Software Architecture* et soutiennent qu'il s'agit de l'un des styles distribués les plus pragmatiques qui soient. Cette section le décrit, l'oppose à l'architecture orientée services (SOA), plus ancienne, avec laquelle on le confond souvent, et montre en quoi les microservices diffèrent de l'une comme de l'autre.

L'architecture à base de services : la topologie

Un système à base de services comporte trois parties. Une ou plusieurs interfaces utilisateur appellent un ensemble de services de domaine déployés séparément, chacun couvrant un domaine d'activité vaste et cohérent, comme les commandes ou les paiements. Richards et Ford observent que ces systèmes comptent généralement entre 4 et 12 services de domaine, environ 7 en moyenne. Les services partagent le plus souvent une base de données, et une couche API facultative (un reverse proxy ou une gateway) peut s'intercaler entre les interfaces utilisateur et les services pour gérer le routage, l'authentification et la limitation de débit.

Chaque service de domaine est à gros grain : il contient tout ce qu'il faut pour son domaine, de l'API à la logique métier et à l'accès aux données, et il est structuré en interne comme un petit monolithe, souvent en couches ou en modules (chapitre 5). Un service est généralement déployé sous la forme de quelques instances derrière un équilibreur de charge.

Deux conséquences en découlent. Premièrement, la plupart des workflows métier s'exécutent entièrement à l'intérieur d'un seul service : ils peuvent donc utiliser une transaction de base de données locale ordinaire, avec toutes les garanties ACID (chapitre 0). Deuxièmement, les services de domaine ont rarement besoin de s'appeler entre eux : une requête issue de l'interface utilisateur va au seul service propriétaire du domaine. Cela garde les chaînes synchrones courtes, ce qui compte pour les raisons calculées à la section 6.1.

SI VOUS DÉBUTEZ

Pas à pas : comment une base de données partagée lie les services entre eux

Le point qui fait le plus souvent trébucher ici est que des services déployés séparément peuvent rester liés par la base de données qu'ils partagent. Un petit exemple montre comment.

- Supposons que les services de la plateforme de livraison de repas lisent librement les tables les uns des autres. Commandes, Restaurants et Livraison lisent tous la table des restaurants dans l'unique base de données partagée.
- L'équipe Restaurants veut scinder la colonne d'adresse de la table en colonnes séparées pour la rue, la ville et le code postal, afin de faciliter la recherche par code postal. Dans son propre code, c'est une modification d'une journée.
- Mais Commandes et Livraison lisent aussi la colonne d'adresse. Dès qu'elle disparaît, ils cassent, alors que personne n'a touché à leur code.
- Les trois services doivent donc être modifiés, testés et mis en production ensemble. Des services liés par des tables partagées ne peuvent pas être déployés indépendamment, c'est pourquoi la section dit qu'ils forment un seul quantum d'architecture.
- Le principal remède proposé dans cette section consiste à donner à chaque service son propre ensemble de tables, son propre schéma, auquel aucun autre service ne peut toucher. Commandes et Livraison lisent alors les adresses au moyen d'une vue en lecture seule que publie Restaurants, et Restaurants peut réorganiser librement ses propres tables tant que cette vue garde la même forme.
- **La leçon :** partager une base de données est ce qui rend ce style économique et simple, mais chaque table que lisent plusieurs services est une promesse que plusieurs équipes doivent tenir ensemble. Donnez à chaque table un seul service propriétaire, et ne partagez que quelques vues publiées.

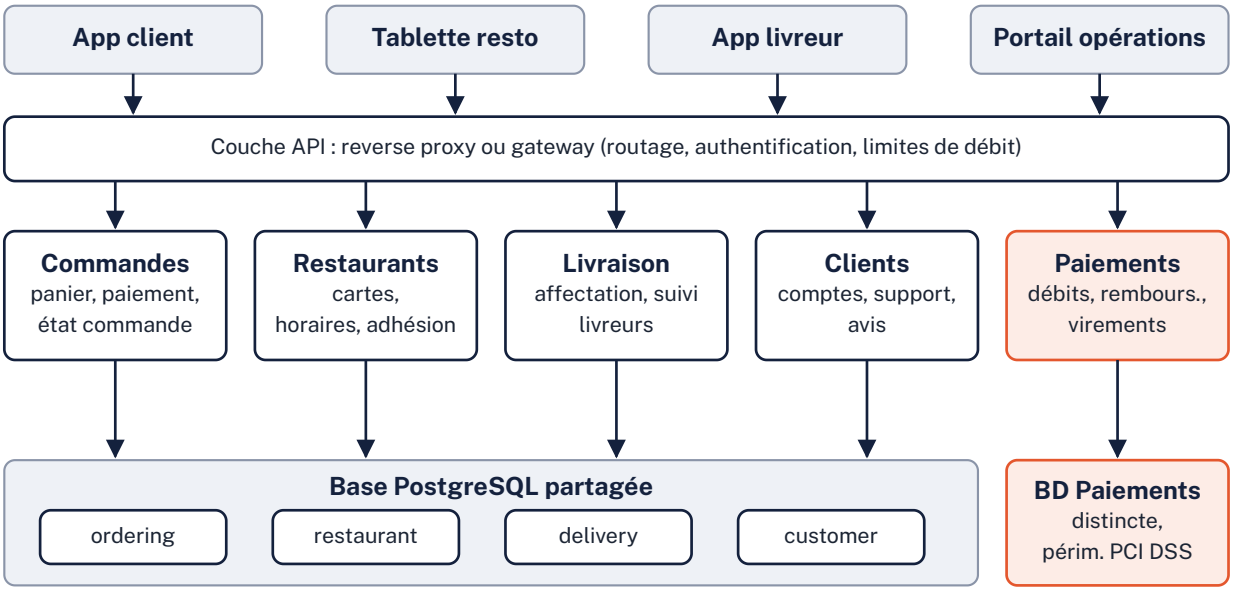
La base de données partagée et comment la maîtriser

Partager une base de données est ce qui rend l'architecture à base de services économique et simple : pas de synchronisation des données entre services, pas de cohérence à terme, des rapports en SQL ordinaire. C'est aussi sa principale faiblesse. Si chaque service lit et écrit les

mêmes tables, une modification des colonnes d'une table peut casser plusieurs services, qui doivent alors tous être testés et redéployés ensemble. La définition du chapitre 5 s'applique : des services qui partagent une base de données sont liés en un seul quantum d'architecture, si bien qu'ils partagent un même niveau de disponibilité et un même rythme de publication du schéma.

Les praticiens utilisent trois techniques pour réduire ce couplage. La première est le partitionnement logique : donner à chaque service son propre schéma (ou ensemble de tables) et lui interdire de toucher à ceux des autres, en faisant respecter cette règle par les permissions de la base de données autant que par convention. La deuxième consiste à placer les classes d'entités de la base partagée dans de petites bibliothèques par domaine plutôt que dans une grande bibliothèque partagée, afin qu'une modification des tables des restaurants n'affecte que les services qui les utilisent. La troisième consiste à donner à un service sa propre base de données quand il a une raison forte, comme un périmètre de sécurité, un profil de charge très différent ou un type de stockage de données différent. Les équipes qui appliquent ces techniques de façon systématique constatent qu'un passage ultérieur à des bases de données séparées, s'il devient un jour nécessaire, est bien plus facile.

La plateforme de livraison de repas en cinq services de domaine



Chaque service possède son propre schéma ; seul Paiements a une base distincte

Cinq services à gros grain, chacun confié à une équipe, derrière une couche API ; quatre partagent une base de données avec un schéma chacun, et Paiements garde la sienne.

EXEMPLE COMMENTÉ

Exemple commenté : la plateforme de livraison de repas en cinq services de domaine

Une entreprise de livraison de repas est passée d'une équipe à cinq, soit environ 35 ingénieurs. Son monolithe Django est bien modularisé, mais les mises en production sont devenues une file d'attente, et une faute dans le suivi des livreurs a fait tomber l'étape de paiement deux fois le trimestre dernier. L'architecte propose une architecture à base de services avec cinq services de domaine, un par module et par équipe existants.

Commandes	Panier, paiement, cycle de vie et état des commandes. Propriétaire du schéma `ordering`. Très sollicité aux heures des repas ; fonctionne avec 6 à 20 instances.
Restaurants	Cartes, prix, horaires d'ouverture, inscription des restaurants. Surtout sollicité en lecture ; les cartes sont mises en cache dans la couche API. Fonctionne avec 3 instances.
Livraison	Affectation des livreurs, estimations d'itinéraire, suivi en direct. Reçoit des mises à jour de position toutes les quelques secondes de chaque livreur actif : il est donc mis à l'échelle séparément de tout le reste.
Paiements	Autorisation des cartes via le prestataire de paiement, remboursements, versements aux restaurants et aux livreurs. Possède sa propre base de données, afin que le périmètre de l'évaluation PCI DSS soit un petit service plutôt que toute la plateforme.
Clients	Comptes, adresses, tickets de support, avis. Faible charge, mais détient la plupart des données à caractère personnel : les règles d'accès et de conservation y sont donc concentrées.
Déroulement d'un workflow	**Commander** s'exécute à l'intérieur de Commandes dans une seule transaction locale (créer la commande, réserver le créneau du restaurant), puis fait un seul appel à Paiements. Restaurants et Livraison apprennent l'existence de la commande par un message, et non par une chaîne d'appels.
Ce qui est partagé	Un seul serveur PostgreSQL avec un schéma par service. Chaque service ne peut lire et écrire que son propre schéma ; quelques vues en lecture seule (les noms des restaurants, par exemple) sont publiées pour les autres.
Ce que cela apporte	Cinq pipelines de déploiement au lieu d'un ; une faute dans le suivi n'arrête plus l'étape de paiement ; les paiements peuvent être audités séparément ; chaque équipe met en production quand elle est prête.
Ce que cela coûte	Cinq unités déployables et cinq tableaux de bord au lieu d'un ; des appels réseau entre Commandes et Paiements qui nécessitent délais d'expiration et idempotence ; des changements de schéma toujours coordonnés par l'intermédiaire du serveur de base de données partagé.

Le pas est volontairement modeste. Il fait passer d'un quantum à environ deux (Paiements avec ses propres données, et les quatre services qui partagent la base), et il donne à chaque équipe sa propre mise en production, ce qui était la principale difficulté.

Si le flux de positions de Livraison devient un jour trop important pour la base partagée, ce seul service peut recevoir son propre stockage et être découpé davantage ; les autres peuvent rester tels quels.

L'architecte consigne les règles de frontière sous forme de tests d'architecture : aucun service ne peut interroger le schéma d'un autre, et les services ne peuvent s'appeler entre eux que par des API publiées.

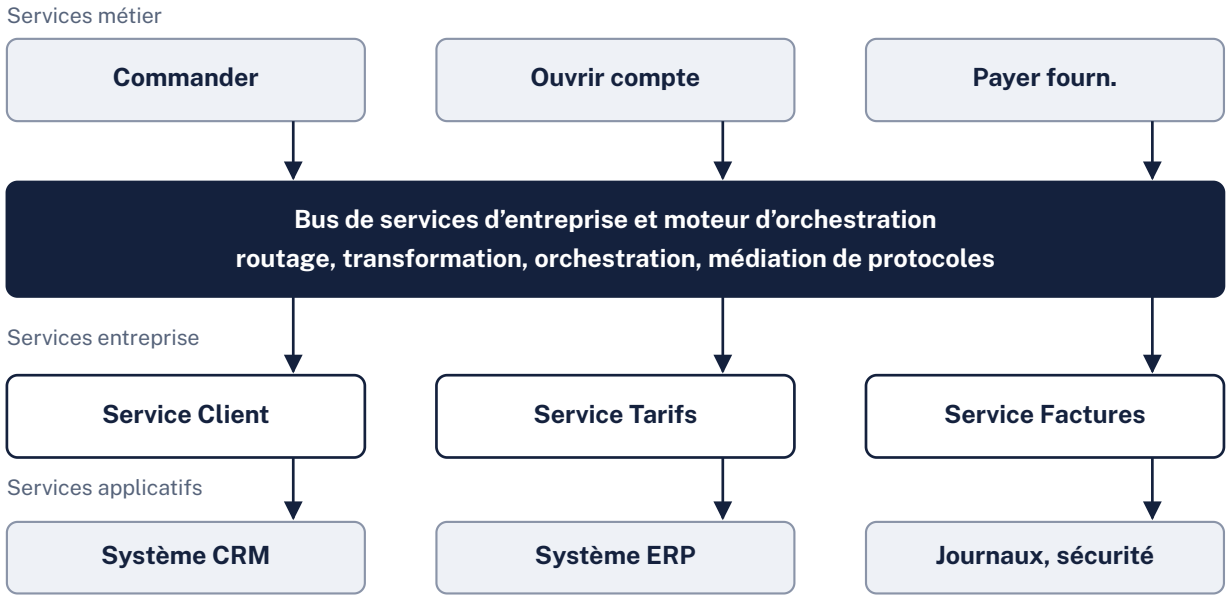
L'architecture orientée services classique

L'architecture orientée services est plus ancienne et plus large. Le terme est entré dans le secteur par un rapport du Gartner de 1996 signé Roy Schulte et Yefim Natis, et à partir du début des années 2000 il est devenu le modèle dominant de l'intégration d'entreprise. Son idée centrale était saine : exposer les capacités métier sous forme de services dotés de contrats explicites et indépendants de la technologie, afin que les applications d'une grande organisation puissent être assemblées à partir d'eux au lieu d'être reconstruites ou intégrées de point à point.

Ce que la plupart des entreprises ont réellement construit, cependant, c'est ce que Richards et Ford appellent la SOA pilotée par l'orchestration. Les services étaient organisés en couches techniques : des services métier décrivant des processus (passer commande, ouvrir un compte), des services d'entreprise partagés fournissant des capacités réutilisables (client, tarification, facture), et des services applicatifs et d'infrastructure enveloppant les systèmes existants et les préoccupations techniques. Au centre se trouvait un bus de services d'entreprise (ESB) : un middleware qui acheminait les messages, les transformait d'un format à un autre, servait d'intermédiaire entre les protocoles et, au moyen d'un moteur d'orchestration, exécutait lui-même les processus métier.

Deux autres idées complétaient le tableau. Un modèle de données canonique définissait une représentation unique, à l'échelle de l'entreprise, de chaque entité métier (un seul Customer, un seul Order), et chaque message était traduit vers ce modèle et depuis ce modèle dans le bus. Et une famille de standards de services web, généralement appelée WS-*, définissait comment les services se décrivaient et s'appelaient : SOAP pour les messages, WSDL pour les descriptions d'interface, UDDI pour les registres, WS-Security, WS-ReliableMessaging et WS-AtomicTransaction pour les qualités, et WS-BPEL (version 2.0 approuvée par OASIS en 2007) pour les définitions de processus exécutables.

La SOA pilotée par l'orchestration



Chaque message est traduit vers et depuis un modèle de données canonique dans le bus

Dans la SOA classique, le bus portait la logique de routage, de transformation et de processus, et tous les services partageaient un même modèle canonique de l'activité.

Pourquoi la SOA classique est tombée en disgrâce

À la fin des années 2000, beaucoup de programmes SOA avaient déçu. Anne Thomas Manes, du Burton Group, a saisi l'état d'esprit du moment dans un billet très lu de janvier 2009 intitulé *SOA is Dead; Long Live Services* : l'étiquette était devenue toxique, mais l'idée de services avait survécu. Les raisons méritent d'être connues, car chacune est un piège dans lequel les architectures modernes peuvent retomber.

La réutilisation a créé du couplage. Un service d'entreprise Customer partagé, utilisé par quarante applications, ne pouvait pas changer sans l'accord de quarante consommateurs ; la réutilisation, l'objectif principal, transformait chaque changement en négociation. **Le bus est devenu le goulet d'étranglement.** Placer la logique de routage, de transformation et de processus dans un ESB central, généralement détenu par une seule équipe d'intégration, signifiait que presque chaque changement métier avait besoin de cette équipe, et le bus est devenu un point unique de défaillance et de retard. **Le modèle canonique était trop gros.** Une définition unique de Customer qui satisfasse les ventes, la facturation, le support et la logistique est soit énorme, soit inadaptée à la plupart d'entre eux ; les contextes délimités du chapitre 5 expliquent pourquoi un terme peut légitimement signifier des choses différentes dans différentes parties d'une entreprise. **Les standards étaient lourds.** Les spécifications WS-* étaient complexes, l'outillage coûteux, et les transactions distribuées entre services se sont révélées fragiles. **Le partitionnement technique** faisait qu'une fonctionnalité simple traversait plusieurs couches de services détenues par des groupes différents.

Les leçons durables sont celles sur lesquelles les microservices ont été bâtis : des contrats de service explicites, des services détenus par ceux qui les construisent, la logique dans les services plutôt que dans le middleware, et des données détenues par un seul service plutôt que partagées dans toute l'entreprise.

En quoi les microservices diffèrent

Les microservices, traités en détail dans la section suivante, conservent l'idée de la SOA de services dotés de contrats explicites et inversent la plupart de ses choix de mise en œuvre. James Lewis et Martin Fowler ont résumé l'un de ces renversements par la formule *smart endpoints and dumb pipes* (des extrémités intelligentes et des tuyaux simples) : le réseau ne fait que transporter les messages, et toute la logique vit dans les services. Les services s'organisent autour des capacités métier plutôt que de couches techniques, la réutilisation passe délibérément après l'indépendance, chaque service est propriétaire de ses données, et la gouvernance est décentralisée.

Par rapport à l'architecture à base de services, les microservices sont à grain plus fin, bien plus nombreux, et exigent des données séparées. Cela apporte davantage d'indépendance, une mise à l'échelle plus ciblée et une meilleure isolation des pannes, et cela coûte beaucoup plus en exploitation, en cohérence et en coordination. L'architecture à base de services peut être vue comme le point du spectre où arrivent la plupart des avantages liés au déploiement et aux équipes, pour une fraction du coût.

Architecture à base de services, SOA classique et microservices comparées

Aspect	À base de services	SOA classique (pilotée par l'orchestration)	Microservices
Objectif principal	Modularité pragmatique et déploiement indépendant de quelques domaines	Réutilisation de services partagés à l'échelle de l'entreprise	Changement et déploiement indépendants par de nombreuses équipes autonomes
Partitionnement	Par domaine : un service par grand domaine d'activité	Technique : couches métier, entreprise, application et infrastructure	Par domaine : un service par contexte délimité, ou plus petit
Nombre typique	Environ 4 à 12 par système	Des dizaines à des centaines dans une entreprise	Des dizaines à des centaines par système
Granularité	Gros grain : tout un domaine, par exemple l'ensemble des commandes	Variable ; les services d'entreprise partagés sont souvent à grain fin	Grain fin à moyen : une capacité cohérente
Données	Souvent une base de données partagée, idéalement partitionnée par schéma	Systèmes de référence partagés derrière des services d'entreprise	Chaque service est propriétaire de ses données ; aucune table partagée
Communication	Surtout de l'interface utilisateur vers un service ; peu d'appels entre services	Via le bus de services d'entreprise, généralement en SOAP ou en messagerie XML	HTTP léger, gRPC ou messagerie ; smart endpoints, dumb pipes
Où vit la logique	Dans les services de domaine	En grande partie dans le bus : routage, transformation, orchestration	Dans les services ; le réseau ne fait que transporter
Transactions	Transactions ACID locales au sein d'un service de domaine	Transactions distribuées tentées (WS-AtomicTransaction, XA)	Pas de transactions distribuées ; sagas et cohérence à terme
Gouvernance	Légère ; une seule équipe d'architecture suffit	Centrale et lourde : modèle canonique, registre de services, comité des standards	Décentralisée ; plateforme partagée et chemins balisés (paved roads)

Aspect	À base de services	SOA classique (pilotée par l'orchestration)	Microservices
Coût d'exploitation	Modéré	Élevé, plus des licences de middleware coûteuses	Élevé : de nombreuses unités déployables, pipelines et tours d'astreinte

Caractéristiques de l'architecture à base de services

Caractéristique	Évaluation	Raison
Déployabilité	Moyenne à élevée	Chaque service de domaine est déployé séparément : une modification des cartes ne redéploie donc pas les paiements ; un changement de schéma partagé nécessite toujours une coordination.
Tolérance aux pannes	Moyenne à élevée	Un plantage du service de livraison laisse les commandes et les paiements fonctionner ; une base de données partagée reste un point de défaillance commun.
Scalabilité	Moyenne	Les services se mettent à l'échelle séparément, mais chacun est à gros grain, et la base de données partagée limite la mise à l'échelle des écritures.
Élasticité	Faible à moyenne	Les gros services démarrent lentement et se mettent à l'échelle par grands paliers : les pics soudains sont donc absorbés moins gracieusement qu'avec de petits services.
Cohérence des données	Élevée	La plupart des workflows métier restent à l'intérieur d'un seul service, où une transaction de base de données ordinaire garde les données cohérentes.
Simplicité	Moyenne	Bien moins de pièces mobiles que les microservices ; cela reste un système distribué, avec des appels réseau, des pipelines de déploiement et une supervision par service.
Coût global	Moyen (modéré)	Une poignée de services et une seule base de données coûtent peu à exploiter comparées à des dizaines de services et de bases de données.
Testabilité	Moyenne à élevée	Chaque service a un périmètre plus restreint que le monolithe et peut être testé isolément face à son schéma.
Évolutivité	Moyenne	Le partitionnement par domaine garde les changements courants à l'intérieur d'un seul service, mais la base de données partagée couple l'évolution des schémas.

Quand l'architecture à base de services est le juste milieu pragmatique

Choisissez-la quand le système compte quelques domaines métier clairs, quand deux à peut-être huit équipes doivent mettre en production indépendamment, quand la plupart des workflows restent naturellement à l'intérieur d'un domaine, et quand une cohérence forte au sein d'un domaine a de la valeur. C'est aussi une excellente étape de migration pour sortir d'un monolithe : découper un monolithe bien modularisé en une poignée de services de domaine est bien moins risqué que de sauter directement à cinquante microservices, et le résultat est souvent assez bon pour s'arrêter là.

Évitez-la, ou prévoyez de la dépasser, quand des parties d'un domaine ont besoin d'une scalabilité ou d'une disponibilité très différentes (elles relèvent de quanta séparés), quand des dizaines d'équipes ont besoin d'autonomie, quand la base de données partagée est devenue le goulet d'étranglement des écritures, ou quand les services de domaine ne cessent de s'appeler entre eux pour traiter des requêtes ordinaires, signe que les frontières sont mal placées.

ATTENTION

Anti-patrons

- **La base de données partagée en libre-service.** Des services qui lisent directement les tables les uns des autres. En quelques mois, le schéma ne peut plus être modifié sans une mise en production coordonnée de l'ensemble.
- **Des services de domaine bavards.** Commandes appelle Restaurants, qui appelle Clients, qui appelle Commandes. Si des workflows ordinaires ont besoin de plusieurs services, fusionnez-les ou déplacez le workflow.
- **L'unique bibliothèque d'entités géante.** Une seule bibliothèque partagée contenant les classes de toutes les tables, si bien que tout changement de schéma oblige chaque service à se reconstruire et à se redéployer.
- **La SOA sous un autre nom.** Une couche d'intégration centrale qui accumule peu à peu règles de routage, transformations et logique métier : un ESB réinventé, avec le même goulet d'étranglement.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Pourquoi la plupart des workflows d'une architecture à base de services peuvent-ils utiliser une transaction de base de données ordinaire ?**

Parce que chaque service de domaine couvre tout un domaine d'activité et que le workflow reste généralement à l'intérieur de celui-ci, en ne touchant que des données du schéma propre à ce service, dans une seule base de données.

▼ **Qu'est-ce qui a fait du bus de services d'entreprise un goulet d'étranglement dans de nombreux programmes SOA ?**

La logique de routage, de transformation et de processus vivait dans le bus, généralement détenu par une seule équipe centrale : la plupart des changements métier avaient donc besoin de cette équipe, et le bus était un point de défaillance partagé.

▼ **Citez deux techniques qui réduisent le couplage causé par une base de données partagée.**

Donner à chaque service son propre schéma et lui interdire l'accès à ceux des autres (en le faisant respecter par les permissions), et répartir les classes d'entités dans de petites bibliothèques par domaine ; ou encore donner à un service sa propre base de données là où c'est justifié.

▼ **Qu'inverse la formule « smart endpoints and dumb pipes » par rapport à la SOA classique ?**

Elle sort la logique du middleware (l'ESB) pour la placer dans les services, et laisse au réseau le seul transport des messages.

▼ **À vous : Une société de billetterie de 12 ingénieurs, avec un monolithe bien rangé, souffre de files d'attente pour les mises en production, et une panne de reporting a deux fois fait tomber le paiement. Recommanderiez-vous une architecture basée sur les services ou des microservices à grain fin ? Donnez votre raisonnement et un risque lié à votre choix.**

À base de services : environ quatre à six services de domaine à gros grain, tels que Réservation, Paiements, Salles et Reporting. La plupart des workflows restent dans un seul service : les transactions locales survivent et les chaînes synchrones restent courtes. Cette approche isole les pannes du reporting et allège les files de mise en production sans le coût d'exploitation de dizaines de services, que 12 ingénieurs ne pourraient pas assumer. Le risque est la base de données partagée, qui couple les changements de schéma et les maintient dans un seul quantum. Utilisez des schémas par service et séparez les données de Reporting.

SOURCES

- James Lewis and Martin Fowler: Microservices (2014), on smart endpoints and dumb pipes: https://martinfowler.com/articles/microservices.html
- OASIS: Reference Model for Service Oriented Architecture 1.0: https://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf
- W3C: Web Services Architecture (Working Group Note): https://www.w3.org/TR/ws-arch/

- OASIS: Web Services Business Process Execution Language Version 2.0: https://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html
- Enterprise Integration Patterns: messaging patterns catalogue: https://www.enterpriseintegrationpatterns.com/patterns/messaging/

Microservices : principes, avantages et coûts

AVANT DE LIRE

Cette section s'appuie sur :

- 5.7 Conception pilotée par le domaine
- 6.1 Pourquoi distribuer
- 6.2 Architecture à base de services et SOA

Termes déjà vus qu'elle utilise : Contexte délimité (bounded context), Quantum d'architecture (architecture quantum), Cohérence à terme (eventual consistency), Bus de services d'entreprise (ESB), Loi de Conway

L'IDÉE EN BREF

Imaginez une grande aire de restauration plutôt qu'un grand restaurant unique. Chaque stand a sa propre petite équipe, sa propre cuisine, son propre réfrigérateur et sa propre carte, et peut changer ses recettes ou fermer pour travaux sans demander l'avis des autres stands. Les clients commandent à un comptoir d'accueil commun qui transmet chaque commande au bon stand. L'aire de restauration peut s'agrandir en ajoutant des stands, et un incendie dans la cuisine d'un stand ne fait pas fermer les autres. Mais il faut bien que quelqu'un gère le comptoir commun, les livraisons, le nettoyage et les contrôles de sécurité pour tous les stands, et un repas composé de plats de trois stands exige une coordination soigneuse. Les microservices, c'est l'aire de restauration.

CE QU'IL FAUT SAVOIR

- Les microservices (Lewis et Fowler, 2014) construisent une application sous forme de petits services, chacun dans son propre processus, organisés autour d'une capacité métier, déployables indépendamment et communiquant par des mécanismes légers.
- Les frontières des services fonctionnent le mieux lorsqu'elles sont alignées sur les contextes délimités et sur les équipes ; des frontières tracées autour de couches techniques ou de tables de base de données produisent des services bavards et fortement couplés.
- La déployabilité indépendante est la propriété qui les définit ; elle repose sur des contrats rétrocompatibles, des tests de contrat et le versionnage, pas seulement sur des conteneurs séparés.

- La base de données par service fait de chaque service son propre quantum, mais remplace les jointures et les transactions par des appels d'API, des événements, des sagas et la cohérence à terme.
- Les patrons de soutien comprennent les passerelles d'API (API gateways), les backends for frontends, la découverte de services et les maillages de services (service meshes) ; chacun résout un vrai problème et ajoute un élément à exploiter.
- La granularité se décide en pesant les désintégrateurs (périmètre, volatilité, scalabilité, tolérance aux pannes, sécurité, extensibilité) face aux intégrateurs (transactions, flux de travail, code partagé, relations entre données).
- Les échecs typiques sont le monolithe distribué, les services d'entités, les services bavards et les bibliothèques partagées qui couplent les déploiements ; ce style exige de l'automatisation, de la supervision et des équipes qui exploitent ce qu'elles construisent.

Définition et origine

Les microservices sont un style architectural dans lequel une application est construite comme un ensemble de petits services, chacun s'exécutant dans son propre processus, chacun organisé autour d'une capacité métier, chacun déployable indépendamment grâce à un outillage automatisé, et communiquant par des mécanismes légers tels que des API HTTP ou la messagerie, avec un minimum de gestion centralisée. Ce résumé suit l'article qui a défini le terme pour la majeure partie du secteur : *Microservices*, de James Lewis et Martin Fowler, publié sur martinfowler.com en mars 2014. L'article indique que le nom a été discuté lors d'un atelier d'architectes logiciels près de Venise en mai 2011, puis adopté par le groupe en mai 2012 ; des entreprises comme Amazon et Netflix travaillaient déjà de cette façon.

Lewis et Fowler n'ont pas donné de définition formelle. Ils ont décrit neuf caractéristiques communes, que les praticiens utilisent encore comme liste de contrôle : la composantisation par les services ; l'organisation autour des capacités métier ; des produits plutôt que des projets ; des extrémités intelligentes et des tuyaux passifs (smart endpoints and dumb pipes) ; une gouvernance décentralisée ; une gestion décentralisée des données ; l'automatisation de l'infrastructure ; la conception pour la défaillance ; et la conception évolutive. Les sections ci-dessous examinent tour à tour les plus significatives d'entre elles sur le plan architectural.

SI VOUS DÉBUTEZ

Si vous débutez

Un *service* est ici simplement un programme qui s'exécute de façon autonome, généralement sur plusieurs serveurs à la fois, et qui offre une API à d'autres programmes (chapitre 0). Dans un monolithe, le code des cartes, des commandes et des paiements est compilé en un seul programme et déployé d'un bloc. Avec les microservices, chacun d'eux devient un programme distinct, détenu par sa propre petite équipe, avec sa propre base de données, et mis en production dès que cette équipe est prête. La suite de cette section porte sur l'endroit où tracer les frontières entre ces programmes, et sur ce que coûte l'exploitation d'un si grand nombre d'entre eux.

Des services alignés sur les contextes délimités

La question la plus difficile des microservices est de savoir où tracer les frontières. La réponse la plus fiable vient de la conception stratégique pilotée par le domaine (chapitre 5) : un contexte délimité (bounded context) est une partie de l'activité à l'intérieur de laquelle un modèle et son langage sont cohérents. Dans la livraison de repas, « commande » désigne un panier en cours de paiement dans Commandes, un bon à préparer dans le contexte de la cuisine du restaurant, et une prise en charge suivie d'une remise au client dans Affectation. Chaque contexte est un candidat naturel pour un service, ou pour un petit nombre de services détenus par une seule équipe.

Aligner les services sur les contextes délimités leur donne une forte cohésion (ce qui change ensemble vit ensemble) et un faible couplage (les contextes communiquent par quelques contrats explicites, traduits aux frontières). Cela les aligne aussi sur les équipes, car un contexte délimité a généralement la taille de ce qu'une équipe peut comprendre et prendre en charge. Les services découpés selon des couches techniques ou des tables de base de données n'ont aucune de ces deux propriétés, et les anti-patrons présentés plus loin dans cette section en sont pour la plupart la conséquence.

La déployabilité indépendante : la propriété qui les définit

Si une propriété définit les microservices, c'est la déployabilité indépendante : une équipe peut modifier son service et le mettre en production sans modifier, tester ni mettre en production aucun autre service en même temps. Sam Newman, dans *Building Microservices* (deuxième édition, 2021), la considère comme la propriété à protéger avant toutes les autres, car presque tous les autres avantages en dépendent. Livraison plus rapide, rayon d'impact (blast radius) réduit, mise à l'échelle ciblée et autonomie des équipes disparaissent tous si les mises en production doivent être coordonnées.

La déployabilité indépendante est une propriété des contrats, pas des conteneurs. Deux services déployés séparément mais qui doivent toujours être mis en production ensemble, parce qu'un changement dans l'un casse l'autre, ne sont pas déployables indépendamment. La protéger exige des changements d'interface rétrocompatibles (ajouter des champs, ne jamais en supprimer ni en renommer sans période de dépréciation), des tests de contrat pilotés par le consommateur et des API versionnées, tous traités dans communication et API.

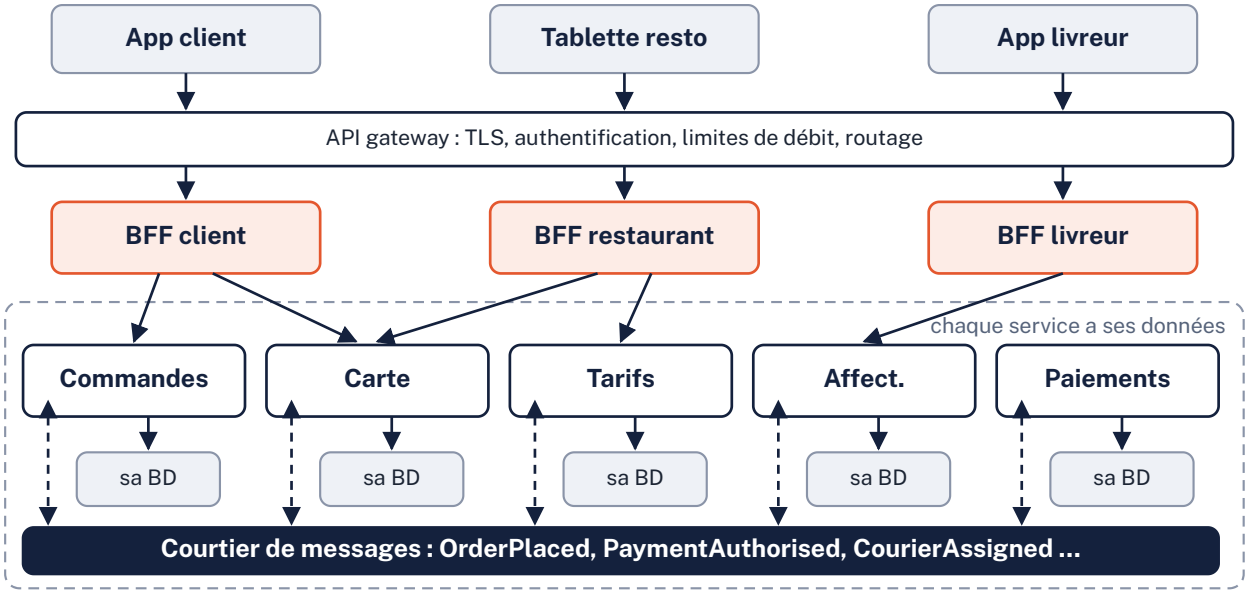
Base de données par service, et extrémités intelligentes avec tuyaux passifs

La gestion décentralisée des données de Lewis et Fowler devient, en pratique, la règle de la base de données par service : chaque service possède ses données, et aucun autre service ne les lit ni ne les écrit directement. Les autres services obtiennent les données par l'API du propriétaire ou par les événements qu'il publie. C'est cette règle qui fait d'un service un quantum distinct : son schéma peut changer, et il peut être mis à l'échelle, sauvegardé et restauré, sans demander l'avis de personne. Chaque service peut aussi choisir le stockage qui lui convient (une base de données relationnelle pour les commandes, un stockage clé-valeur pour les sessions, un moteur de recherche pour les cartes), ce que l'on appelle la persistance polyglotte.

Le prix est élevé et doit être payé en connaissance de cause. Les requêtes qui étaient une jointure SQL entre tables traversent désormais plusieurs services. Les changements qui tenaient en une seule transaction exigent désormais une saga et deviennent cohérents à terme. Le reporting nécessite un pipeline séparé. Le chapitre 7 traite en profondeur de la propriété des données.

L'autre caractéristique aux conséquences architecturales les plus importantes est celle des *extrémités intelligentes et tuyaux passifs*. La logique métier a sa place dans les services ; le réseau et le courtier de messages (message broker) ne font que transporter les messages. C'est une réaction directe au bus de services d'entreprise de la SOA classique, où les règles de routage, les transformations et les processus s'accumulaient dans un middleware central.

Une topologie microservices pour la livraison de repas



Flèches pleines : appels synchrones. Pointillés : événements publiés et consommés via le courtier.

Les clients accèdent aux services par une passerelle et un backend for frontend chacun ; chaque service possède ses données, et les événements circulent par un courtier.

Patrons de soutien

Un ensemble de patrons de soutien s'est développé autour de ce style. Chacun résout un vrai problème et chacun ajoute un composant à exploiter ; adoptez-les donc quand le problème apparaît plutôt que tous à la fois.

Patrons de soutien courants

Patron	Problème résolu	Ce que c'est	Points de vigilance
API gateway (passerelle d'API)	Les clients ne devraient pas connaître les adresses de dizaines de services, ni répéter l'authentification auprès de chacun.	Un point d'entrée unique qui termine les connexions TLS, authentifie, limite le débit et route les requêtes vers les services ; il peut combiner plusieurs appels en une seule réponse.	Une logique métier qui s'insinue dans la passerelle et recrée un ESB ; un point de défaillance unique qui exige une haute disponibilité.
Backend for frontend (BFF)	Une app mobile, une app web et une API partenaire ont besoin de données de formes différentes ; une API générale unique ne sert bien aucune d'entre elles.	Un backend distinct et léger pour chaque type de client, détenu par l'équipe de ce client, qui appelle les services et met en forme les réponses. Le nom vient de Phil Calçado, chez SoundCloud ; Sam Newman a popularisé le patron en 2015.	Une logique dupliquée entre les BFF ; un BFF qui devient un mini-monolithe.
Découverte de services	Les instances de service démarrent, s'arrêtent et se déplacent en permanence ; les adresses ne peuvent pas être configurées à la main.	Un registre des instances saines du moment, interrogé par les clients ou par un équilibreur de charge. Kubernetes fournit cette fonction au moyen de noms DNS attribués aux Services.	Entrées de registre périmées ; clients qui gardent des adresses en cache trop longtemps.
Service mesh (maillage de services)	Chaque service a besoin de TLS mutuel, de nouvelles tentatives, de délais d'expiration, de bascule progressive du trafic et de télémétrie ; les implémenter dans chaque langage est du gaspillage.	Une couche d'infrastructure faite de proxys (traditionnellement un sidecar à côté de chaque instance de service), pilotée de façon centralisée, comme Istio ou Linkerd.	Une complexité opérationnelle considérable ; des nouvelles tentatives configurées à la fois dans le maillage et dans le code multiplient la charge.

Les prérequis de Fowler, et les coûts organisationnels

En août 2014, Martin Fowler a énuméré trois choses qu'une organisation devrait posséder avant même de construire des microservices. **Le provisionnement rapide** : la capacité de créer un nouveau serveur ou environnement en quelques heures, ce qui suppose de l'automatisation. **La supervision (monitoring) de base** : des mesures techniques (erreurs, latence, disponibilité) et des mesures métier (commandes par minute) pour chaque service, afin de repérer vite les

problèmes. **Le déploiement rapide des applications** : un pipeline automatisé qui amène vite un service en test et en production, avec un retour arrière facile. Sans cela, de nombreux petits services multiplient le travail manuel. Son idée complémentaire, le *surcoût des microservices* (microservice premium, 2015), est que ce style comporte un coût fixe en complexité et en productivité qui n'est rentable que pour des systèmes assez complexes pour en avoir besoin.

Les coûts organisationnels sont aussi réels que les coûts techniques. Ce style suppose des équipes durables qui exploitent leurs services en production, selon le principe que le CTO d'Amazon, Werner Vogels, a résumé en 2006 par «You build it, you run it» (« vous le construisez, vous l'exploitez »). Cela signifie que chaque équipe a besoin de compétences d'exploitation et d'un planning d'astreinte. Il faut une plateforme : pipelines de déploiement, observabilité, gestion des secrets et modèles de services, généralement fournis par une équipe plateforme (chapitre 10). Il faut s'accorder sur des standards transversaux (format de journalisation, identifiants de corrélation, conventions d'API) sans approbation centrale de chaque changement. Et il faut utiliser délibérément la loi de Conway : les frontières des équipes et celles des services doivent correspondre, sinon chaque changement exige la coordination de plusieurs équipes.

Le côté client : rendu, micro-frontends, mobile, hors ligne et BFF

Les clients rencontrent un système à travers ses applications clientes, et celles-ci ont leur propre architecture. La première décision porte sur l'endroit où une page est rendue. Dans une **application monopage** (SPA), le navigateur construit chaque écran avec JavaScript, ce qui convient aux sessions longues et interactives comme l'écran des commandes d'un restaurant, mais le code tend à grossir avec l'application et peut ralentir la réponse aux touchers. Avec le **rendu côté serveur** (SSR), le serveur envoie du HTML terminé : le contenu apparaît donc vite et moins de code s'exécute sur l'appareil, au prix d'un travail du serveur à chaque requête. Le **rendu statique** produit le HTML au moment du build pour un CDN, ce qui convient aux pages d'aide, mais pas aux pages personnalisées. Beaucoup de frameworks effectuent le rendu sur le serveur puis procèdent à l'*hydratation*, qui attache le comportement dans le navigateur ; web.dev de Google avertit qu'une telle page peut sembler prête avant de répondre. Le choix peut différer d'une page à l'autre.

Les **micro-frontends** transposent le déploiement indépendant dans l'interface : « des applications frontend livrables indépendamment sont composées en un ensemble plus vaste » (Cam Jackson, 2019). Les équipes de la carte et du paiement pourraient chacune publier leur propre partie du site web, composée à l'exécution par un outil comme Module Federation de webpack. Les coûts sont des bibliothèques dupliquées dans chaque téléchargement, ou des versions partagées qui réintroduisent le couplage, ainsi que davantage de pipelines et plus d'efforts pour garder une apparence cohérente. Ce style répond au problème de nombreuses équipes sur un seul frontend ; une petite équipe y gagne peu.

Le mobile change les règles de mise en production. Une mise à jour atteint les utilisateurs progressivement, par la publication échelonnée sur sept jours d'Apple ou le déploiement progressif de Google Play, et interrompre un déploiement sur Play laisse la mise à jour à ceux qui l'ont déjà. Les anciennes versions restent utilisées ; les API doivent donc rester compatibles longtemps (voir la section 6.5).

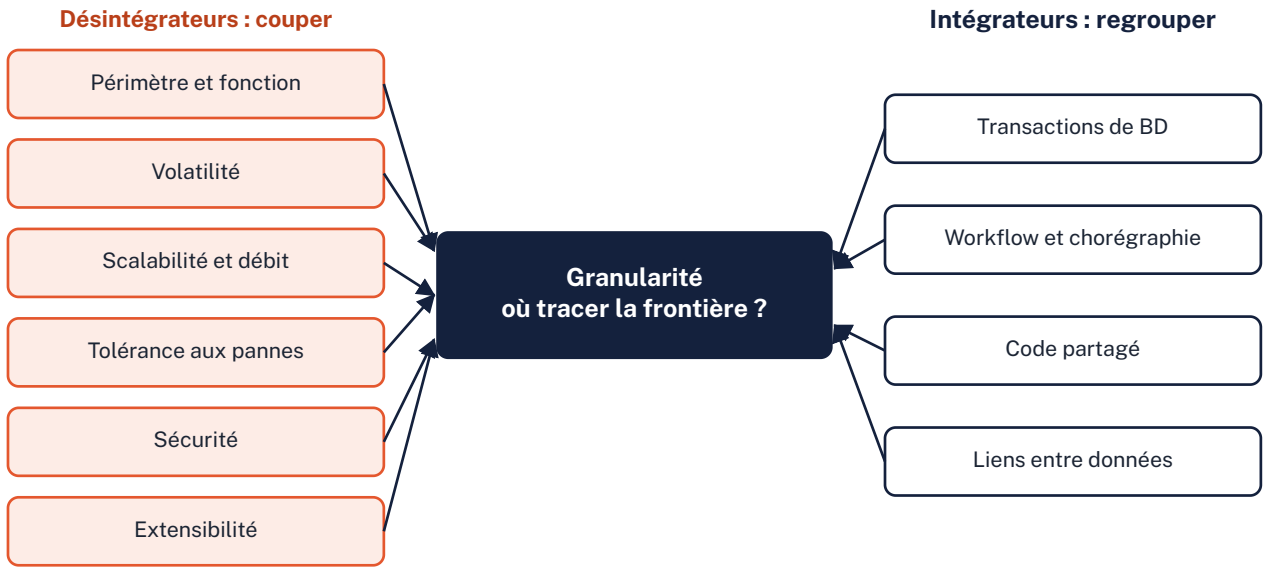
Un client **offline-first** (d'abord hors ligne) traite un stockage local comme sa source de vérité : les écrans lisent les données locales et un processus de synchronisation les réconcilie avec le serveur, qui garde l'autorité finale. Les recommandations d'Android nomment trois stratégies d'écriture : en ligne uniquement, adaptée aux paiements ; en file d'attente ; et différée, qui écrit d'abord localement et doit résoudre les conflits. Un livreur dans un parking souterrain peut marquer une commande comme livrée et laisser l'app l'envoyer plus tard, avec une clé d'idempotence. Dans les navigateurs, les service workers peuvent servir des pages en cache, mais la Background Sync API ne fonctionne pas dans tous les principaux navigateurs : vérifiez donc la prise en charge actuelle.

Le **backend for frontend** relie le tout. La règle de Sam Newman est une expérience, un BFF, tenu par l'équipe qui construit cette interface : le BFF des livreurs peut ainsi renvoyer de petites charges utiles pour les réseaux médiocres, tandis que le BFF web met en forme les données page par page. Il peut aussi garder les jetons hors du navigateur (voir la section 8.3). Les règles métier restent dans les services : Newman suggère de déplacer la logique partagée par les BFF dans un service de domaine. Son avertissement sur l'option inverse, une API générale pour tous les clients, est qu'elle devient un goulot d'étranglement.

Granularité : quelle taille pour un service ?

Il n'existe pas de taille correcte en lignes de code ou en nombre de personnes. Lewis et Fowler ont observé des équipes d'une douzaine de personnes détenant un service et des équipes d'environ six personnes détenant six services, et regrettaient que le mot « micro » attire l'attention sur la taille. Ford, Richards, Sadalage et Dehghani proposent un meilleur outil dans *Software Architecture: The Hard Parts* (2021) : peser les forces qui poussent à découper un service, qu'ils appellent **désintégrateurs de granularité**, face à celles qui poussent à le garder entier, les **intégrateurs**.

Les forces qui s'exercent sur la granularité des services



Forces selon Ford, Richards, Sadalage et Dehghani, Software Architecture: The Hard Parts (2021)

Six désintégrateurs plaident pour des services plus petits ; quatre intégrateurs plaident pour des services plus grands. La granularité est un équilibre, à trancher au cas par cas.

Désintégrateurs et intégrateurs de granularité, avec des exemples tirés de la livraison de repas

Force	Sens	Question à poser	Exemple
Périmètre et fonction	Découper	Le service fait-il une seule chose cohérente, ou plusieurs choses sans rapport entre elles ?	Un service Notifications qui calcule aussi les points de fidélité fait deux métiers.
Volatilité	Découper	Une partie change-t-elle bien plus souvent que le reste ?	Les règles de promotion changent chaque semaine ; le reste de Tarification change chaque trimestre.
Scalabilité et débit	Découper	Les parties ont-elles besoin de capacités très différentes ?	Les mises à jour de position des livreurs arrivent des milliers de fois par seconde ; l'inscription des livreurs, quelques fois par jour.
Tolérance aux pannes	Découper	Une défaillance dans une partie en ferait-elle tomber une autre sans nécessité ?	Un plantage du générateur de reçus PDF ne devrait pas empêcher la prise de commande.
Sécurité	Découper	Les parties ont-elles besoin de contrôles d'accès ou d'audits différents ?	Les jetons de cartes bancaires stockés face aux données publiques des cartes de restaurant.
Extensibilité	Découper	De nouvelles variantes seront-elles ajoutées souvent ?	De nouveaux moyens de paiement tous les quelques mois, chacun déployable séparément.
Transactions de BD	Garder ensemble	Ces changements doivent-ils réussir ou échouer de façon atomique ?	Créer une commande et réserver son créneau de livraison.
Workflow et chorégraphie	Garder ensemble	Le découpage créerait-il de longues chaînes d'appels pour des requêtes ordinaires ?	Panier, paiement et statut de commande s'appellent mutuellement à chaque requête.
Code partagé	Garder ensemble	Les parties partagent-elles une logique métier importante qui change en même temps ?	Carte et Tarification partagent des règles complexes d'options de plats.
Liens entre données	Garder ensemble	Les données sont-elles étroitement liées, avec des jointures et des clés étrangères ?	Les commandes et les lignes de commande ; les séparer ne rapporte rien.

EXEMPLE COMMENTÉ

Exemple commenté : faut-il découper Tarification ?

La plateforme de livraison de repas dispose d'un service Tarification qui calcule le total des paniers : prix des plats, suppléments d'options, frais de livraison (qui dépendent de la distance et de la demande) et promotions (bons de réduction, remises sur la première commande, offres des restaurants). L'équipe promotions livre des changements plusieurs fois par semaine, et chaque mise en production de Tarification exige un passage complet des tests de non-régression du calcul des frais. Quelqu'un propose de découper Tarification en deux services, Frais et Promotions. L'architecte pèse les forces.

Volatilité du code (découper)	Forte : les promotions changent chaque semaine, les règles de frais chaque trimestre. Les séparer permettrait de livrer les promotions sans retester les frais.
Scalabilité (découper)	Faible : les deux sont appelés une fois par affichage du panier, au même rythme.
Tolérance aux pannes (découper)	Modérée : une règle de promotion erronée a fait planter Tarification deux fois. Si Promotions était séparé, une défaillance pourrait se réduire à « aucune remise affichée » tandis que les frais fonctionneraient toujours.
Sécurité (découper)	Faible : même sensibilité des données.
Transactions de base de données (garder)	Faible : aucune écriture ne doit couvrir les deux ; un bon de réduction est utilisé au moment où la commande est passée, dans Commandes.
Flux de travail (garder)	Modérée : chaque total de panier a besoin des deux. Après un découpage, Tarification (ou un BFF) doit appeler Promotions de façon synchrone à chaque affichage du panier, ce qui ajoute un saut réseau et ses modes de défaillance.
Code partagé (garder)	Modérée : les deux ont besoin du modèle de panier et des règles d'arrondi monétaire, qui changent rarement.
Relations entre données (garder)	Faible : les promotions ne font référence aux restaurants et aux plats que par leur identifiant.
Décision	Découper, avec précaution : extraire Promotions ; Tarification l'appelle avec un délai d'expiration (timeout) de 150 ms et, en cas d'échec, calcule le prix du panier sans remises et le signale pour un nouveau calcul à l'étape de paiement. Ne partager qu'une petite bibliothèque de modèle de panier, versionnée.

La décision repose sur deux désintégrateurs forts (volatilité et isolation des défaillances) qui l'emportent sur des intégrateurs modérés. Si les promotions avaient dû mettre à jour les mêmes lignes que le calcul des frais dans une seule transaction, la réponse aurait

probablement été différente.

Écrire les forces noir sur blanc transforme une affaire de goût en un argument que d'autres peuvent vérifier, et cette trace a sa place dans un enregistrement de décision d'architecture.

Anti-patrons

La plupart des programmes de microservices qui échouent le font de manière reconnaissable. L'échec le plus grave est le monolithe distribué : un système qui a tous les coûts de la distribution et aucune de ses indépendances, parce que ses services partagent une base de données, s'appellent en longues chaînes synchrones ou dépendent des mêmes bibliothèques déployables.

Anti-patrons des microservices

Anti-patron (anti-pattern)	Symptôme	Cause et remède
Monolithe distribué	Les services doivent être mis en production ensemble dans un ordre fixe ; la panne d'un service les arrête tous.	Base de données partagée, chaînes synchrones et bibliothèques déployables partagées. Remède : redessiner les frontières autour des contextes délimités, donner à chaque service ses données, remplacer les appels par des événements, ou fusionner à nouveau en un monolithe modulaire.
Services d'entités	Des services nommés d'après des noms communs (Client, Commande, Restaurant), avec des opérations de création, lecture, mise à jour et suppression ; chaque processus métier en appelle plusieurs.	Des frontières tracées autour des données plutôt que du comportement. Michael Nygard appelle cela l'anti-patron du service d'entité (entity service). Remède : s'organiser autour des capacités et des processus métier.
Services bavards	Une seule requête utilisateur provoque des dizaines d'appels entre services ; la latence et les taux de défaillance augmentent à chaque fonctionnalité.	Les frontières traversent un flux de travail cohérent, ou les interfaces sont trop fines. Remède : des API à plus gros grain, des copies locales des données de référence, la fusion des services qui changent toujours ensemble.
Bibliothèques partagées qui couplent les déploiements	Un changement dans une bibliothèque commune oblige chaque service à être reconstruit et redéployé ; les versions divergent.	Logique métier ou modèles de domaine placés dans du code partagé. Remède : ne partager que des bibliothèques techniques stables ; dupliquer de petites quantités de code métier ; versionner tout ce qui est partagé et permettre l'utilisation de plusieurs versions à la fois.
L'envie de microservices	Une petite équipe avec un nouveau produit démarre avec trente services et passe l'essentiel de son temps sur l'infrastructure.	Choisir ce style par effet de mode ou pour le CV plutôt qu'en fonction des facteurs. Remède : commencer par un monolithe modulaire et extraire des services quand un facteur apparaît.
Nanoservices	Des services si petits que l'essentiel de leur code est de la plomberie ; une fonctionnalité exige des changements dans cinq d'entre eux.	Appliquer « petit » à l'excès. Remède : utiliser les intégrateurs de granularité ; dimensionner les services selon leur cohésion, pas selon les lignes de code.

Caractéristiques des microservices

Caractéristique	Évaluation	Raison
Déployabilité	Élevée	Chaque service est mis en production seul, plusieurs fois par jour, avec un faible rayon d'impact.
Scalabilité et élasticité	Élevée	Chaque service passe à l'échelle indépendamment ; les petits services démarrent vite et s'ajustent par paliers fins.
Tolérance aux pannes	Élevée (si la conception le prévoit)	Les défaillances peuvent être isolées par service, mais seulement avec des délais d'expiration, des cloisonnements, des solutions de repli et une communication asynchrone ; sinon, les fautes se propagent en cascade.
Évolutivité	Élevée	Des services petits et bien délimités peuvent être réécrits ou remplacés individuellement.
Testabilité d'un service	Élevée	Chaque service est petit ; des tests de contrat vérifient ses interactions.
Tester l'ensemble	Faible	Le comportement de bout en bout émerge de nombreux services et il est difficile à reproduire dans un environnement de test.
Performances	Faible à moyenne	Les sauts réseau, la sérialisation et les contrôles de sécurité ajoutent de la latence à chaque interaction.
Cohérence des données	Faible	Pas de transactions partagées ; les invariants métier qui couvrent plusieurs services reposent sur la cohérence à terme et les sagas.
Simplicité	Faible	De nombreuses pièces mobiles, de nombreux modes de défaillance, beaucoup d'infrastructure.
Coût global	Faible (coûteux)	Plateforme, outillage, observabilité et personnel pour exploiter des dizaines de services.

Quand les microservices conviennent, et quand ils ne conviennent pas

Les microservices conviennent aux organisations dont de nombreuses équipes travaillent sur un même produit et doivent mettre en production indépendamment, aux systèmes dont les parties ont des besoins réellement différents en scalabilité, en disponibilité ou en sécurité, et aux domaines suffisamment bien compris pour tracer des frontières stables. Ils exigent un investissement dans une plateforme et dans les compétences d'exploitation avant même que le premier service soit extrait.

Ils ne conviennent pas à une petite équipe, à un nouveau produit dont le domaine est encore en cours de découverte (des frontières tracées trop tôt coûtent cher à déplacer à travers le réseau), à un système dont les flux de travail centraux exigent une forte cohérence transactionnelle sur la plupart de ses données, ni à une organisation dépourvue de déploiement et de supervision automatisés. Le conseil *MonolithFirst* de Fowler (chapitre 5) et les retours en arrière que certaines entreprises ont documentés, comme la consolidation en 2018, chez Segment, de plus de 140 services de destination en un seul, montrent que ce style est un outil qui a un coût, pas une destination.

> **REMARQUE**
>
> ### Les praticiens divergent, et c'est utile
>
> Certains architectes expérimentés soutiennent que la plupart des organisations ne devraient jamais aller au-delà d'un monolithe modulaire ou d'une architecture à base de services ; d'autres soutiennent que de petits services, avec une bonne plateforme, sont la façon la plus productive de travailler pour de nombreuses équipes. Les deux camps s'accordent sur les conditions : des frontières de domaine claires, la déployabilité indépendante, la propriété des données et l'automatisation. Là où ces conditions manquent, le nombre de services n'est pas le problème à résoudre.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Deux services sont déployés dans des conteneurs séparés mais doivent toujours être mis en production ensemble. Sont-ils déployables indépendamment ?

Non. La déployabilité indépendante consiste à pouvoir modifier et mettre en production l'un sans l'autre ; un contrat couplé ou une base de données partagée en font une seule unité du point de vue de la mise en production.

▼ Quelle est la différence entre une API gateway et un backend for frontend ?

Une passerelle est un point d'entrée général qui gère des préoccupations transversales comme TLS, l'authentification, la limitation de débit et le routage ; un BFF est un backend propre à un client, qui met en forme les données pour un type de frontend et qui est détenu par l'équipe de ce frontend.

▼ Quel intégrateur de granularité s'oppose le plus fortement au découpage de deux parties qui doivent mettre à jour des données de façon atomique ?

Les transactions de base de données : les découper remplacerait une transaction ACID par un flux de travail distribué tel qu'une saga.

▼ **Pourquoi des services nommés Client, Commande et Restaurant, offrant uniquement des opérations de création, lecture, mise à jour et suppression, sont-ils un signal d'alerte ?**

Ce sont des services d'entités : des frontières tracées autour des données plutôt que du comportement, si bien que chaque processus métier doit appeler plusieurs services, ce qui crée des interactions bavardes et fortement couplées.

▼ **À vous : Une équipe propose CustomerService, OrderService et OrderLineService, chacun enveloppant une table. Passer une commande exige d'écrire ensemble une commande et cinq lignes. À l'aide des désintégrateurs et des intégrateurs, OrderLineService est-il un bon découpage ? Que feriez-vous ?**

Non. C'est un service d'entité autour d'une table, et non une capacité métier. Les commandes et les lignes partagent une transaction, un workflow et des relations de données, tous des intégrateurs forts, et aucun désintégrateur, comme une scalabilité, une volatilité ou une sécurité différentes, ne s'applique. Une fois découpé, il serait bavard, avec une saga nécessaire pour ce qu'une seule transaction locale fait aujourd'hui. Fusionnez les lignes dans un service Commandes aligné sur son contexte délimité. Ne découpez que lorsque de vraies forces apparaissent.

SOURCES

- James Lewis and Martin Fowler: Microservices (2014): https://martinfowler.com/articles/microservices.html
- Martin Fowler: Microservice Prerequisites (2014): https://martinfowler.com/bliki/MicroservicePrerequisites.html
- Sam Newman: Pattern: Backends For Frontends: https://samnewman.io/patterns/architectural/bff/
- Michael Nygard: The Entity Service Antipattern: https://michaelnygard.com/blog/2017/12/the-entity-service-antipattern/
- Thoughtworks: Software Architecture: The Hard Parts, chapter 7 on service granularity (sample PDF): https://www.thoughtworks.com/content/dam/thoughtworks/documents/books/bk_software_architecture_hard_parts_ch7_en.pdf
- Cam Jackson: Micro Frontends (martinfowler.com, 2019): https://martinfowler.com/articles/micro-frontends.html
- web.dev: Rendering on the Web: https://web.dev/articles/rendering-on-the-web
- webpack: Module Federation: https://webpack.js.org/concepts/module-federation/
- Apple: Release a version update in phases: https://developer.apple.com/help/app-store-connect/update-your-app/release-a-version-update-in-phases/

- Google Play Console Help: staged rollouts: https://support.google.com/googleplay/android-developer/answer/6346149
- Android Developers: Build an offline-first app: https://developer.android.com/topic/architecture/data-layer/offline-first
- MDN: Offline and background operation: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/Offline_and_background_operation
- MDN: Background Synchronization API: https://developer.mozilla.org/en-US/docs/Web/API/Background_Synchronization_API

SECTION 6.4 SUR 6.8 · AVANCÉ · 28 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Architecture pilotée par les événements : courtiers, médiateurs et flux d'événements

AVANT DE LIRE

Cette section s'appuie sur :

- 0.5 Stocker les données
- 0.6 API et intégration
- 6.1 Pourquoi distribuer

Termes déjà vus qu'elle utilise : Messagerie asynchrone, Idempotent, Transaction, Contexte délimité (bounded context), Couplage

L'IDÉE EN BREF

Pensez à la cuisine d'un restaurant très fréquenté. Quand un serveur accroche un bon de commande sur le rail, personne ne reçoit d'ordre : le cuisinier du grill voit qu'il y a un steak à cuire, le poste des salades voit une salade, et le pâtissier voit un dessert pour plus tard. Chaque poste réagit au bon à son propre rythme. Si le pâtissier est en pause, le bon attend sur le rail jusqu'à son retour. Ajouter un nouveau poste ne change pas la façon dont les serveurs rédigent les bons. Un système piloté par les événements fonctionne de la même façon : des parties annoncent ce qui s'est produit, et d'autres parties réagissent.

CE QU'IL FAUT SAVOIR

- Un événement est un fait qui s'est produit (au passé, avec de nombreux destinataires possibles) ; une commande est une demande adressée à un seul gestionnaire, qui peut la refuser ; un message est l'enveloppe qui transporte l'un ou l'autre.
- La topologie à courtier répartit un flux de travail en réactions indépendantes (chorégraphie) ; la topologie à médiateur confie la séquence à un coordinateur unique (orchestration). La chorégraphie couple le moins ; l'orchestration rend les flux de travail visibles et les défaillances plus faciles à gérer.
- Fowler distingue la notification d'événement, le transfert d'état porté par les événements, l'event sourcing et CQRS ; les deux premiers sont des façons de faire communiquer les services, les deux derniers des façons pour un service de stocker ses données.
- Dans Kafka, un topic est divisé en partitions ; les événements de même clé vont dans la même partition, l'ordre n'est garanti qu'à l'intérieur d'une partition, et chaque groupe de consommateurs répartit les partitions entre ses membres et suit ses propres offsets.

- La livraison peut être au plus une fois, au moins une fois ou effectivement une fois ; la livraison exactement une fois n'est pas réalisable en général : partez donc du principe d'une livraison au moins une fois et rendez chaque consommateur idempotent.
- Les événements publiés sont des contrats durables : donnez-leur des identifiants, des versions et des horodatages, faites-les évoluer de façon compatible et vérifiez-les avec un registre de schémas.
- Relancez les défaillances transitoires avec un backoff, déplacez les messages empoisonnés vers une file de lettres mortes supervisée, et ne laissez jamais un mauvais message bloquer une partition.

Définition : réagir à ce qui se produit

Dans une architecture pilotée par les événements (EDA), les composants communiquent en produisant des événements et en y réagissant : des enregistrements de faits qui se sont produits dans l'activité, comme *une commande a été passée* ou *un livreur a été affecté*. Le composant qui publie un événement n'appelle pas les composants qui y réagissent et, en général, ne sait même pas qu'ils existent. Un courtier de messages (message broker) transporte les événements des producteurs vers les consommateurs, les conserve pendant que les consommateurs sont occupés ou indisponibles, et permet d'ajouter de nouveaux consommateurs sans modifier le producteur.

Les conceptions pilotées par les événements ne sont pas nouvelles : les interfaces graphiques, les systèmes d'exploitation et les systèmes de trading fonctionnent ainsi depuis des décennies, et les produits de messagerie d'entreprise sont antérieurs au web. Ce qui a changé dans les années 2010, c'est l'échelle et la durabilité. Les courtiers fondés sur un journal, comme Apache Kafka, créé chez LinkedIn et publié en open source en 2011, ont rendu possible de conserver chaque événement métier pendant des jours ou des années, et de les rejouer. L'EDA peut être le style principal d'un système ou, plus souvent, la façon dont les services d'un système en microservices ou à base de services communiquent entre eux.

SI VOUS DÉBUTEZ

Si vous débutez

Un *courtier de messages* est un programme qui se place entre d'autres programmes et fait passer les messages, un peu comme un bureau de poste. Un expéditeur remet un message au courtier et poursuit son travail ; le courtier le conserve en sécurité jusqu'à ce que chaque destinataire soit prêt à le récupérer. Comme l'expéditeur n'attend pas, peu importe qu'un destinataire soit occupé ou brièvement hors ligne. Le chapitre 0 a présenté cette idée sous le nom de messagerie asynchrone via une file ; cette section montre comment des architectures entières sont construites autour d'elle.

Événements, commandes et messages

Trois mots sont souvent employés de façon approximative, et la différence compte. Un message est l'enveloppe : toute donnée envoyée d'un programme à un autre par un canal. Ce que transporte le message peut être un événement (un fait, nommé au passé), une commande (une demande adressée à un seul gestionnaire, nommée comme une instruction) ou une requête et sa réponse. *Enterprise Integration Patterns* (2003), de Gregor Hohpe et Bobby Woolf, a catalogué ces types de messages et les canaux qui les transportent, et son vocabulaire reste la référence.

Événement, commande et message

	Événement	Commande	Message (l'enveloppe)
Signification	Un fait : quelque chose qui s'est produit	Une demande : merci de faire quelque chose	Toute donnée envoyée d'un programme à un autre par un canal
Nommage	Au passé : `OrderPlaced`, `PaymentAuthorised`	À l'impératif : `AuthorisePayment`, `AssignCourier`	Transporte un événement, une commande, une requête ou une réponse
Destinataires	Zéro, un ou plusieurs ; l'émetteur ne les connaît pas	Exactement un gestionnaire prévu	Dépend du canal : une file (un seul) ou un topic (plusieurs)
Peut-on le refuser ?	Non ; il s'est déjà produit. Les destinataires décident comment réagir	Oui ; le gestionnaire peut la rejeter ou échouer à l'exécuter	Sans objet
Couplage	L'émetteur ne dépend pas des consommateurs	L'expéditeur dépend de la capacité et du contrat du destinataire	Dépend de ce qu'il transporte

Les messages circulent par deux types de canaux. Une file remet chaque message à l'un de ses consommateurs, ce qui convient aux commandes et à la répartition du travail. Un topic remet une copie de chaque message à chaque abonné : c'est la publication-abonnement (publish-subscribe, ou pub-sub), qui convient aux événements. Une conception qui envoie `OrderPlaced` à un topic permet à Paiements, Restaurant, Notifications et Analytique de réagir, chacun à son rythme ; ajouter un service de détection de fraude l'an prochain ne demande qu'un nouvel abonnement et aucun changement dans Commandes.

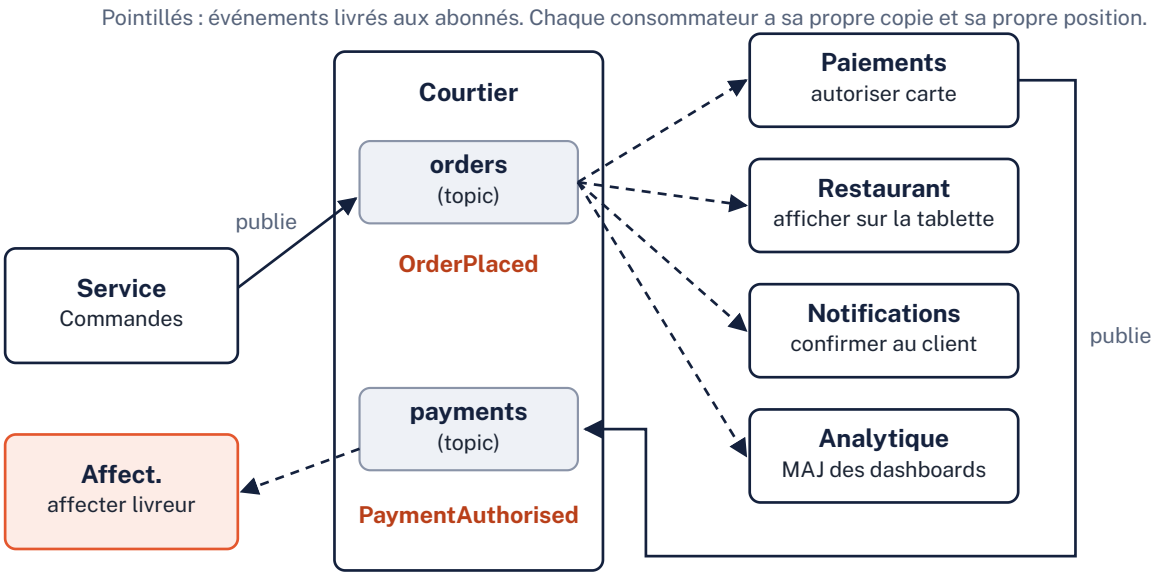
Topologie à courtier et topologie à médiateur

Richards et Ford décrivent deux topologies pour les systèmes pilotés par les événements. Dans la topologie à courtier (broker), il n'y a pas de coordinateur central. Un événement initiateur est publié ; chaque processeur intéressé réagit et peut publier d'autres événements, auxquels d'autres réagissent à leur tour. Le flux de travail n'existe que comme la somme de ces réactions. C'est la chorégraphie : comme des danseurs qui connaissent chacun leurs propres pas, personne ne dirige l'ensemble.

Dans la topologie à médiateur, un médiateur d'événements reçoit l'événement initiateur et dirige le flux de travail, en envoyant des commandes aux processeurs dans un ordre défini et en agissant selon leurs réponses. C'est l'orchestration : un chef d'orchestre connaît la partition.

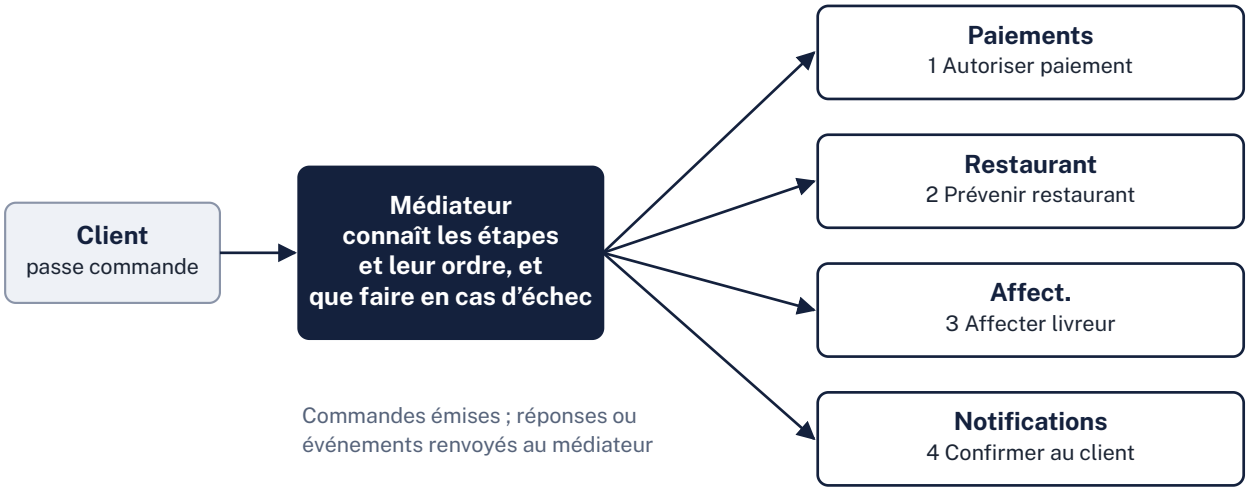
Les médiateurs vont d'un simple service doté d'une machine à états à un moteur de workflow comme Temporal, AWS Step Functions ou un moteur BPMN.

Topologie à courtier : le flux OrderPlaced



En topologie à courtier, Commandes ne fait qu'annoncer le fait ; chaque consommateur réagit indépendamment, et c'est l'événement propre de Paiements qui déclenche l'étape suivante.

Topologie à médiateur : le même flux, orchestré



En topologie à médiateur, un seul composant détient la séquence : le flux de travail est donc visible et les défaillances sont gérées en un seul endroit.

Chorégraphie ou orchestration ?

Question	Chorégraphie (courtier)	Orchestration (médiateur)
Couplage	Le plus faible : les émetteurs ne connaissent pas les consommateurs	Le médiateur dépend du contrat de chaque participant
Voir l'ensemble du flux de travail	Difficile : il est réparti entre les services ; il faut du traçage pour le reconstituer	Facile : il est écrit en un seul endroit
Gestion des erreurs	Chaque participant doit détecter les défaillances et y réagir, souvent en publiant des événements de compensation	Le médiateur décide de façon centralisée des nouvelles tentatives, des délais d'expiration et des compensations
Ajouter une étape	Ajouter un abonné ; rien d'autre ne change	Modifier le médiateur
Risques	Chaînes d'événements cycliques ; personne ne remarque quand une étape s'arrête silencieusement	Le médiateur devient un goulot d'étranglement ou accumule de la logique métier
Bien adaptée à	Flux simples et stables ; nombreuses réactions indépendantes à un même fait	Flux de travail longs et conditionnels avec compensation, comme la saga de commande

La plupart des systèmes réels mélangent les deux : chorégraphie entre contextes délimités (Commandes annonce des faits et ne se soucie pas de savoir qui écoute) et orchestration à l'intérieur d'un contexte, là où un processus en plusieurs étapes a besoin d'un responsable clair.

Quatre façons d'utiliser les événements

Dans un article de 2017, Martin Fowler a souligné que « piloté par les événements » sert à désigner au moins quatre patrons assez différents, et que les conversations tournent mal quand les interlocuteurs ne parlent pas du même. Savoir les distinguer est l'une des habitudes les plus utiles qu'un architecte puisse avoir.

Les quatre patrons d'événements de Fowler (2017)

Patron	Ce que porte l'événement, et pourquoi	Avantage	Coût
Notification d'événement	Un petit signal indiquant que quelque chose a changé, souvent un simple identifiant : `OrderPlaced {order_id}`. Les destinataires rappellent l'émetteur pour obtenir les détails s'ils en ont besoin.	Couplage très faible ; le modèle de l'émetteur reste privé.	Les rappels ajoutent de la charge et une dépendance à l'exécution envers l'émetteur ; le flux réparti sur de nombreux événements est difficile à voir.
Transfert d'état porté par les événements (event-carried state transfer)	Assez de données pour que les destinataires conservent leur propre copie : la commande complète avec ses articles et ses totaux.	Les destinataires fonctionnent sans appeler l'émetteur, même quand il est indisponible ; latence plus faible.	Les données sont copiées largement et sont cohérentes à terme ; événements plus volumineux ; les schémas deviennent des contrats.
Event sourcing	Chaque changement de l'état d'une entité est stocké sous forme d'événement ; le journal des événements est la source de vérité et l'état courant est dérivé en le rejouant.	Piste d'audit complète ; reconstitution des états passés ; nouvelles vues tirées de l'historique ancien.	Complexe ; le rejeu est difficile quand les résultats dépendaient de systèmes externes ; les schémas d'événements doivent évoluer indéfiniment.
CQRS	Des modèles séparés (souvent des stockages séparés) pour l'écriture et pour la lecture, fréquemment alimentés par des événements.	Chaque côté est optimisé pour sa tâche ; convient aux systèmes très sollicités en lecture avec des requêtes complexes.	Deux modèles à maintenir synchronisés ; souvent appliqué là où un seul modèle suffirait.

La notification d'événement et le transfert d'état porté par les événements concernent la communication entre services et ont leur place dans ce chapitre. L'event sourcing et CQRS concernent la façon dont un service stocke ses données ; on les combine souvent avec l'EDA, mais ce sont des choix indépendants, traités au chapitre 7.

Flux d'événements et journaux : comment fonctionne Kafka

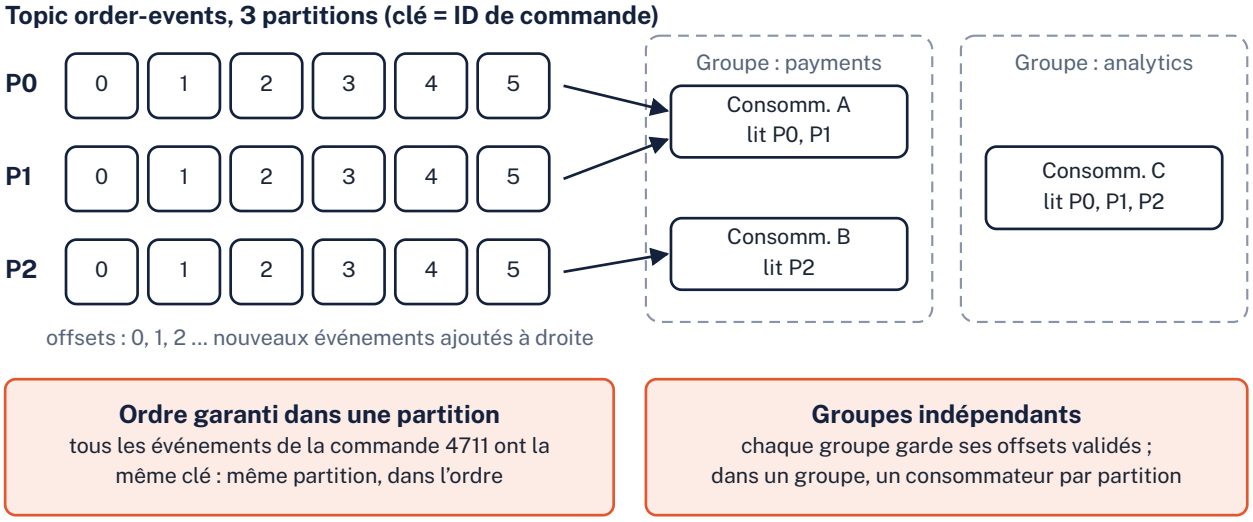
Les files de messages traditionnelles suppriment un message dès qu'un consommateur l'a acquitté. Un courtier fondé sur un journal, comme Apache Kafka, stocke au contraire les événements dans un journal en ajout seul (append-only) et les conserve pendant une durée de rétention configurée (ou indéfiniment), qu'ils aient été lus ou non. Plusieurs conséquences découlent de cette conception, et elles expliquent à la fois les forces de Kafka et ses pièges.

Un topic est divisé en partitions, chacune étant un journal ordonné, stocké et répliqué sur plusieurs courtiers. Un producteur choisit la partition, normalement en hachant la clé de l'événement, si bien que tous les événements de même clé (le même identifiant de commande, par exemple) vont dans la même partition. Chaque événement d'une partition a une position séquentielle appelée son offset (décalage). Les consommateurs tirent les événements et enregistrent l'offset qu'ils ont atteint ; ils peuvent aussi revenir en arrière et rejouer.

Les consommateurs rejoignent un groupe de consommateurs. Au sein d'un groupe, chaque partition est lue par un seul consommateur à la fois, si bien que le groupe se partage le travail ; des groupes différents lisent indépendamment, chacun avec ses propres offsets, et c'est ainsi que l'on obtient le pub-sub. Le nombre de partitions limite donc le parallélisme d'un groupe : un topic de 12 partitions peut occuper au plus 12 consommateurs d'un groupe. (Les *share groups* de Kafka, proposés en préversion dans les versions 4.x antérieures et prêts pour la production depuis Kafka 4.2 en 2026, sont une alternative de type file : plusieurs consommateurs peuvent se partager une partition et acquitter les messages individuellement, mais l'ordre au sein de la partition n'est alors plus préservé. Cette section décrit les groupes de consommateurs classiques.)

Kafka ne garantit l'ordre qu'au sein d'une partition. Les événements d'une même commande arrivent dans l'ordre où ils ont été écrits s'ils partagent une clé ; les événements de commandes différentes, dans des partitions différentes, n'ont aucun ordre défini les uns par rapport aux autres. Les conceptions qui ont besoin d'un ordre global doivent utiliser une seule partition (en renonçant au parallélisme) ou, mieux, être repensées pour que seul l'ordre par entité compte.

Partitions, offsets et groupes de consommateurs



Ordre garanti dans une partition
tous les événements de la commande 4711 ont la même clé : même partition, dans l'ordre

Groupes indépendants
chaque groupe garde ses offsets validés ; dans un groupe, un consommateur par partition

Chaque partition est un journal ordonné ; un groupe de consommateurs répartit les partitions entre ses membres, et chaque groupe suit ses propres offsets.

Garanties de livraison, et ce que signifie vraiment « exactement une fois »

Comme les réseaux et les processus tombent en panne (section 6.1), un courtier et ses clients doivent choisir ce qui se passe lorsqu'un plantage ou un délai d'expiration laisse planer un doute sur le traitement d'un message. Il existe trois garanties possibles, et le choix dépend du moment où les acquittements et les offsets sont enregistrés.

Garanties de livraison

Garantie	Ce qu'elle promet	Comment on l'obtient	Conséquence typique
Au plus une fois (at-most-once)	Chaque message est traité une fois ou pas du tout ; jamais deux fois.	Le consommateur enregistre sa position (valide l'offset ou acquitte) avant le traitement, ou le producteur ne fait jamais de nouvelle tentative.	Des messages peuvent être perdus après un plantage. Acceptable pour des échantillons de métriques ou des signaux de position des livreurs, où la mise à jour suivante remplace la précédente.
Au moins une fois (at-least-once)	Chaque message est traité une ou plusieurs fois ; jamais perdu.	Le producteur réessaie jusqu'à l'acquittement, et le consommateur ne valide sa position qu'après le traitement.	Des doublons après les plantages, les nouvelles tentatives et les rééquilibrages. C'est le comportement par défaut de la plupart des courtiers et le choix habituel pour les événements métier.
Effectivement une fois (effectively-once)	L'effet de chaque message est appliqué exactement une fois, même s'il peut être livré plus d'une fois.	Une livraison au moins une fois associée à un consommateur idempotent, ou une transaction qui enregistre ensemble le résultat et la position du consommateur.	Le sens pratique de « exactement une fois » dans les systèmes métier ; exige une conception délibérée dans chaque consommateur.

« Exactement une fois » demande une définition rigoureuse. La *livraison* exactement une fois sur un réseau non fiable n'est pas réalisable en général : un expéditeur qui ne reçoit pas d'acquittement ne peut pas savoir s'il doit renvoyer le message. Ce que les systèmes peuvent offrir, c'est un *effet* exactement une fois à l'intérieur d'une frontière définie. Kafka, par exemple, propose un producteur idempotent et des transactions depuis la version 0.11 (2017), afin qu'un cycle lecture-traitement-écriture allant de topics Kafka vers des topics Kafka valide sa sortie et les offsets de son consommateur de façon atomique. Cette garantie s'arrête à la frontière de Kafka. Si le consommateur débite une carte, envoie un e-mail ou écrit dans une autre base de données, un plantage entre cette action et la validation de l'offset entraînera la répétition de l'action, à moins que le consommateur lui-même soit idempotent ou qu'il stocke ses offsets dans la même transaction que ses résultats.

La règle pratique pour les architectes est donc simple : partez du principe d'une livraison au moins une fois et faites de chaque consommateur un consommateur idempotent. Donnez à chaque événement un identifiant unique ; dans le consommateur, enregistrez les identifiants

traités dans la même transaction locale que le changement d'état, et ignorez tout événement déjà enregistré. Dans la mesure du possible, rendez aussi l'opération naturellement idempotente, par exemple « passer le statut à PAID s'il est AWAITING_PAYMENT » plutôt que « ajouter un paiement ».

```
# Idempotent consumer: processing the same event twice has the effect of once
def handle_payment_authorised(event, db):
    with db.transaction():                      # one local ACID transaction
        inserted = db.execute(
            "INSERT INTO processed_events (event_id) VALUES (%s) "
            "ON CONFLICT (event_id) DO NOTHING",
            [event["event_id"]],
        ).rowcount
        if inserted == 0:
            return                              # seen before: do nothing
        db.execute(
            "UPDATE orders SET status = 'PAID' WHERE order_id = %s "
            "AND status = 'AWAITING_PAYMENT'",
            [event["order_id"]],
        )
    # commit the broker offset only after the transaction has committed
```

Un consommateur idempotent : l'enregistrement des événements traités et le changement d'état sont validés ensemble, si bien qu'un événement relivré ne change rien.

Schémas d'événements et versionnage

Dès que d'autres équipes consomment un événement, sa structure est un contrat publié, exactement comme une API, et elle dure plus longtemps : avec un courtier fondé sur un journal, les consommateurs peuvent lire des événements écrits il y a des mois. Un bon événement comporte un identifiant unique, un type, une version de schéma, l'heure à laquelle il s'est produit, les identifiants des entités concernées et les données métier dont les consommateurs ont besoin. Les montants sont stockés en unités mineures avec une devise, jamais sous forme de nombre à virgule flottante (chapitre 0).

```
{
  "event_id": "9f1c2e7a-4b0d-4e55-9a51-0c3d8f2b6a10",
  "type": "OrderPlaced",
  "schema_version": 2,
  "occurred_at": "2026-10-22T18:42:07Z",
  "order_id": "ord-4711",
  "customer_id": "cus-208",
  "restaurant_id": "res-31",
  "items": [{"dish_id": "d-5", "qty": 2, "price_minor": 1150}],
  "total_minor": 2450,
  "currency": "GBP"
}
```

Un événement OrderPlaced qui utilise le transfert d'état porté par les événements : identifiants, version, heure et données dont les consommateurs ont besoin.

Les schémas doivent évoluer de façon rétrocompatible : ajouter des champs optionnels, ne jamais supprimer ni renommer des champs ni changer leur signification sans nouvelle version, et ne jamais réutiliser un nom de champ pour autre chose. De nombreuses équipes l'imposent avec un format de schéma comme Avro, Protocol Buffers ou JSON Schema et un registre de schémas qui rejette les changements incompatibles au moment de la publication. La spécification CloudEvents, projet de niveau « graduated » de la Cloud Native Computing Foundation, normalise les attributs d'enveloppe courants comme id, source, type et time. Les modes de compatibilité et les techniques de migration sont traités dans évolution des schémas.

Gestion des erreurs : nouvelles tentatives, messages empoisonnés et files de lettres mortes

Les consommateurs échouent pour deux raisons très différentes, qui demandent des traitements différents. Les **défaillances transitoires**, comme une base de données brièvement indisponible ou un délai d'expiration lors de l'appel d'un autre service, réussissent généralement après une nouvelle tentative ; réessayez quelques fois avec un backoff exponentiel et du jitter (des délais croissants et aléatoires) afin de ne pas submerger une dépendance en difficulté. Les **défaillances permanentes**, comme un événement mal formé ou qui viole une règle métier, échoueront à chaque fois. Un tel « message empoisonné » (poison message) ne doit pas bloquer le consommateur indéfiniment, ce qui, sur un journal partitionné, bloquerait tous les événements suivants de cette partition.

La réponse standard est une file de lettres mortes (dead-letter queue, DLQ) : après un nombre défini de tentatives, le message est déplacé vers un canal distinct, avec l'erreur enregistrée, et le consommateur passe à la suite. Une DLQ n'est utile que si quelqu'un la surveille : déclenchez des alertes sur sa taille, confiez-en la responsabilité à une équipe nommément désignée, fournissez des outils pour inspecter, corriger et rejouer les messages, et décidez à l'avance

quelles garanties d'ordre sont perdues lorsqu'un message est rejoué tardivement. Certaines équipes ajoutent un ou plusieurs topics intermédiaires de nouvelle tentative, avec des délais plus longs, avant la DLQ.

Caractéristiques de l'architecture pilotée par les événements

Caractéristique	Évaluation	Raison
Scalabilité et élasticité	Élevée	Les consommateurs passent à l'échelle indépendamment et absorbent les pics grâce au tampon du courtier ; les partitions permettent le parallélisme.
Réactivité	Élevée	L'émetteur rend la main dès que l'événement est enregistré de façon durable ; le travail lent se fait plus tard.
Tolérance aux pannes	Élevée	Un consommateur peut être indisponible pendant des minutes puis rattraper son retard depuis le courtier ; les défaillances ne remontent pas en cascade jusqu'à l'émetteur.
Extensibilité	Élevée	De nouveaux consommateurs s'abonnent aux événements existants sans aucun changement chez l'émetteur.
Cohérence des données	Faible	L'état se propage entre services avec un délai ; les lecteurs voient des données périmées pendant un certain temps.
Testabilité et débogabilité	Faible	Le comportement émerge de réactions asynchrones ; reproduire une séquence d'événements à travers les services est difficile sans un bon traçage.
Visibilité du flux de travail	Faible (chorégraphie) à moyenne (médiateur)	Personne ne détient le flux de bout en bout en chorégraphie ; un médiateur le rend explicite.
Simplicité	Faible	Courtiers, schémas, nouvelles tentatives, traitement des lettres mortes, ordre et doublons doivent tous être conçus.

Quand l'utiliser, et erreurs courantes

L'architecture pilotée par les événements convient quand de nombreuses parties d'une activité doivent réagir aux mêmes faits, quand le travail peut se faire après le retour de la requête de l'utilisateur, quand la charge arrive par à-coups et doit être lissée, et quand de nouveaux consommateurs sont attendus au fil du temps. Elle convient moins bien quand un utilisateur a

besoin d'une réponse immédiate et définitive qui dépend de plusieurs services (même si un appel synchrone pour la réponse, complété par des événements pour la suite, est un hybride courant), ou quand une forte cohérence entre services est requise.

ATTENTION

Erreurs courantes

- **Des commandes déguisées en événements.** `SendConfirmationEmail` publié sur un topic est une commande dont le couplage est masqué ; publiez `OrderPlaced` et laissez Notifications décider.
- **Supposer une livraison exactement une fois.** Un consommateur qui débite une carte ou envoie un e-mail sans contrôle d'idempotence finira par le faire deux fois.
- **Supposer un ordre global.** Les événements de partitions différentes, ou de producteurs différents, peuvent arriver dans n'importe quel ordre ; concevez un ordre par entité avec des clés et des numéros de version.
- **Écrire dans la base de données et publier séparément.** Si le service plante entre les deux, l'événement est perdu ou annonce quelque chose qui ne s'est jamais produit. Le patron de l'outbox transactionnelle résout ce problème.
- **Les spaghettis d'événements.** Des dizaines de services réagissant aux événements les uns des autres sans carte des flux. Tenez un catalogue d'événements, tracez avec des identifiants de corrélation, et utilisez un médiateur là où un flux de travail a besoin d'un responsable.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **`ReserveCourier` est-il un événement ou une commande, et comment le savoir ?**

Une commande : c'est une instruction adressée à un seul gestionnaire (Affectation), qui peut la refuser. Un événement serait nommé au passé, comme `CourierReserved`.

▼ **Un topic Kafka a 6 partitions et un groupe de consommateurs compte 10 consommateurs. Combien de consommateurs seront occupés ?**

Au plus 6. Au sein d'un groupe, chaque partition est attribuée à un seul consommateur : 4 consommateurs restent donc inactifs.

▼ **Pourquoi la prise en charge de l'exactement une fois par Kafka n'empêche-t-elle pas un consommateur d'envoyer deux fois un e-mail de confirmation ?**

Ses transactions couvrent la lecture et l'écriture dans des topics Kafka ainsi que la validation des offsets ; un e-mail envoyé à un système externe est hors de cette frontière, si bien qu'un plantage avant la validation de l'offset entraîne un nouveau traitement, à moins que le consommateur soit idempotent.

▼ **Quelle est la différence entre la notification d'événement et le transfert d'état porté par les événements ?**

La notification envoie un petit signal, souvent un simple identifiant, si bien que les destinataires rappellent l'émetteur pour obtenir les détails ; le transfert d'état inclut les données dont les destinataires ont besoin, ce qui leur permet de conserver leur propre copie et d'éviter d'appeler l'émetteur.

▼ **À vous : Un consommateur voit parfois OrderCancelled avant OrderPaid pour une même commande. Le topic a 12 partitions et les événements ont des clés aléatoires. Il traite aussi parfois un événement deux fois. Diagnostiquez les deux problèmes et donnez des correctifs.**

L'ordre n'est garanti qu'à l'intérieur d'une partition, et des clés aléatoires dispersent les événements d'une commande entre les partitions. Clé des événements : l'ID de commande, afin que tous les événements d'une commande arrivent dans la même partition, dans l'ordre. Les doublons sont normaux avec une livraison au moins une fois : rendez donc le consommateur idempotent, par exemple en enregistrant les ID d'événements traités dans la même transaction que sa mise à jour. Incluez un numéro de version, et envoyez les messages invalides vers une file de lettres mortes supervisée plutôt que de bloquer la partition.

SOURCES

- Martin Fowler: What do you mean by "Event-Driven"? (2017): https://martinfowler.com/articles/201701-event-driven.html
- Apache Kafka: Introduction (topics, partitions and ordering): https://kafka.apache.org/intro
- Confluent documentation: Kafka message delivery guarantees: https://docs.confluent.io/kafka/design/delivery-semantics.html
- Enterprise Integration Patterns: Idempotent Receiver: https://www.enterpriseintegrationpatterns.com/patterns/messaging/IdempotentReceiver.html
- CloudEvents specification: https://cloudevents.io/

Communication synchrone et asynchrone : REST, gRPC, GraphQL et messagerie

AVANT DE LIRE

Cette section s'appuie sur :

- 0.6 API et intégration
- 6.1 Pourquoi distribuer
- 6.4 Pilotée par les événements

Termes déjà vus qu'elle utilise : HTTP, Code d'état HTTP, Défaillance partielle, Changement cassant (breaking change), Publication-abonnement (publish-subscribe)

L'IDÉE EN BREF

Les services dialoguent de deux façons. Un appel téléphonique est synchrone : vous restez en ligne jusqu'à obtenir une réponse, et si votre interlocuteur n'est pas là, vous n'obtenez rien. Une lettre est asynchrone : vous la postez et vaquez à vos occupations, et la réponse arrive plus tard. Les appels téléphoniques sont simples, mais lient les deux personnes au même moment ; les lettres sont plus souples, mais vous ne savez pas tout de suite si votre demande a abouti. En outre, les deux parties ont besoin d'un accord écrit et partagé sur ce qui peut être demandé et ce qui est renvoyé, ainsi que d'un moyen de modifier cet accord sans gêner ceux qui en dépendent.

CE QU'IL FAUT SAVOIR

- Les appels synchrones créent un couplage temporel : l'appelant et l'appelé doivent être disponibles en même temps, si bien que les latences s'additionnent et que les disponibilités se multiplient le long d'une chaîne ; la messagerie asynchrone supprime ce couplage, mais pas le couplage de format ni le couplage sémantique.
- REST est l'ensemble de contraintes de Fielding (client-serveur, sans état, mise en cache possible, interface uniforme, système en couches, code à la demande facultatif) ; la plupart des API réelles atteignent le niveau 2 du modèle de maturité de Richardson, pas le niveau 3 de l'hypermédia.
- La RFC 9110 définit GET, HEAD, OPTIONS, PUT et DELETE comme idempotentes et POST comme ni sûre ni idempotente ; les clients ne doivent rejouer automatiquement que les requêtes idempotentes après un délai d'expiration, un 502, un 503 ou un 504 : POST a donc besoin d'une clé d'idempotence pour pouvoir être rejouée sans risque.

- gRPC utilise Protocol Buffers sur HTTP/2, avec du code généré et du streaming, et convient aux appels internes ; GraphQL permet à des clients variés de choisir leurs champs, mais exige du regroupement des requêtes, des limites de coût des requêtes et une autorisation au niveau des champs.
- Les files remettent chaque message à un seul des consommateurs concurrents ; les topics en remettent une copie à chaque abonné.
- Les API durables modélisent des ressources du domaine, paginent avec des curseurs, renvoient des problem details RFC 9457, n'évoluent que de façon rétrocompatible, et sont versionnées ou passent par l'expansion-contraction lorsqu'une rupture est inévitable.
- OpenAPI et AsyncAPI décrivent les contrats HTTP et les contrats de messages ; les tests de contrat pilotés par le consommateur, comme Pact, indiquent aux fournisseurs quels changements casseraient quels consommateurs.

Deux façons de dialoguer, et le couplage que chacune crée

Une fois qu'un système est découpé en parties déployées séparément (voir pourquoi distribuer), ces parties doivent dialoguer à travers un réseau, et la façon dont elles dialoguent façonne le système autant que la façon dont elles sont découpées. Il existe deux modes de base. Dans la communication synchrone, l'appelant envoie une requête et attend la réponse avant de continuer, comme lors d'un appel téléphonique. Dans la communication asynchrone, l'émetteur confie un message à un intermédiaire, généralement un courtier de messages (message broker), et poursuit son travail ; le destinataire récupère le message quand il est prêt, comme une lettre dans une boîte aux lettres. La section 0.6 a présenté la différence ; elle est examinée ici en tant que décision architecturale.

Les appels synchrones créent un couplage temporel : les deux parties doivent être en fonctionnement et joignables au même moment. La disponibilité de l'appelant est donc au mieux égale au produit des disponibilités de ses dépendances, et son temps de réponse, y compris sa latence de queue (tail latency), contient le leur : c'est l'arithmétique de la section 6.1. Un service de paiement lent rend le service des commandes lent, et si tous les threads du service des commandes l'attendent, le service des commandes cesse de répondre à qui que ce soit. Les délais d'expiration, les nouvelles tentatives et les disjoncteurs limitent les dégâts ; ils sont traités dans les patrons de résilience.

La messagerie asynchrone supprime le couplage temporel : le courtier stocke le message, si bien que le destinataire peut être lent, occupé ou brièvement indisponible sans que l'émetteur s'en aperçoive. Elle ne supprime pas toutes les formes de couplage. Les deux parties dépendent toujours du format et de la signification du message (couplage de format et couplage sémantique), ainsi que du courtier lui-même, qui devient une infrastructure critique.

L'émetteur perd aussi la connaissance immédiate du résultat : la conception doit donc gérer le travail qui échoue plus tard, les messages qui arrivent deux fois et ceux qui arrivent dans un ordre différent. L'asynchronisme déplace la complexité au lieu de la supprimer.

Une règle empirique utile : utilisez un appel synchrone quand l'appelant ne peut vraiment pas continuer sans la réponse (« cette carte est-elle autorisée ? »), et la messagerie asynchrone quand l'appelant a seulement besoin de savoir que le travail sera fait (« envoyer le reçu », « prévenir le restaurant »). Beaucoup de bonnes conceptions gardent le chemin côté utilisateur court et synchrone, et placent tout le reste derrière une file.

Le couplage temporel, avec et sans courtier



Dans une chaîne synchrone, l'appelant est bloqué et dépend de chaque service en aval ; avec un courtier entre les deux, l'émetteur et le destinataire n'ont plus besoin d'être disponibles au même moment.

SI VOUS DÉBUTEZ

Pas à pas : un appel téléphonique ou une lettre ?

L'idée la plus difficile de cette section est de savoir ce dont vous libère, et ne vous libère pas, le fait de poster un message au lieu d'attendre une réponse. Le chapitre 0 a présenté les deux modes ; suivez une commande de repas à travers chacun d'eux.

- Un client appuie sur « Commander ». Le service des commandes doit savoir si la carte est autorisée avant de pouvoir dire oui : il appelle donc le service de paiement et attend. C'est un appel synchrone, comme un appel téléphonique.
- Si le service de paiement est en panne, la commande ne peut pas être passée : les deux doivent être en fonctionnement et joignables au même moment. C'est le couplage temporel.
- Ensuite, le client a besoin d'un reçu par e-mail. Le service des commandes n'a pas besoin de l'attendre : il dépose donc un message ReceiptNeeded auprès d'un courtier de messages, une sorte de boîte aux lettres, et répond tout de suite au client. C'est asynchrone, comme une lettre.
- Supposons que le service e-mail soit en panne pendant dix minutes. Le courtier conserve le message, et le reçu arrive simplement en retard. Si le service des commandes avait téléphoné au service e-mail et attendu, chaque commande passée pendant ces dix minutes aurait échoué ou serait restée bloquée.
- La lettre ne libère pas complètement les deux services l'un de l'autre. Si le service des commandes renomme un champ du message, par exemple email en emailAddress, le service e-mail ne peut plus le lire : les deux parties dépendent toujours de la même structure convenue et de la même signification de ce qui est envoyé. Et le service des commandes ne sait plus immédiatement si l'e-mail est parti.
- **La leçon :** n'appelez et n'attendez que lorsque vous ne pouvez pas continuer sans la réponse ; postez un message lorsque vous avez seulement besoin de savoir que le travail sera fait. Dans les deux cas, le format de ce qui est envoyé est une promesse, à modifier avec soin, comme l'explique la fin de cette section.

REST tel que Fielding l'a défini

REST (Representational State Transfer) a été défini par Roy Fielding dans sa thèse de doctorat de 2000 comme un style architectural pour les grands systèmes hypermédias en réseau : le web lui-même. Fielding l'a construit comme un ensemble de contraintes, dont chacune apporte une qualité. Le **client-serveur** sépare l'interface utilisateur du stockage des données, afin que les deux évoluent indépendamment. Une interaction **sans état** signifie que chaque requête contient tout ce qu'il faut pour la comprendre, si bien que n'importe quelle instance de serveur peut la traiter, ce qui favorise la scalabilité et la reprise. Des réponses **pouvant être mises en**

cache indiquent si elles peuvent être réutilisées, ce qui réduit la latence et la charge. Une **interface uniforme** (des ressources identifiées par des URI, manipulées au moyen de représentations, des messages autodescriptifs, et l'hypermédia comme moteur de l'état de l'application) permet à des intermédiaires génériques, comme les caches et les proxys, de comprendre le trafic. Un **système en couches** permet à des proxys, des passerelles et des équilibreurs de charge de s'insérer sur le chemin de façon invisible. Le **code à la demande**, comme le JavaScript envoyé à un navigateur, est la seule contrainte facultative.

La plupart des API dites RESTful ne satisfont que certaines de ces contraintes. Le modèle de maturité de Leonard Richardson, présenté en 2008 et popularisé par Martin Fowler en 2010, en décrit les étapes. Au **niveau 0**, HTTP n'est qu'un tunnel : chaque appel est un POST vers un seul endpoint, avec l'opération nommée dans le corps, comme en SOAP. Au **niveau 1**, l'API a des ressources distinctes (`/orders/1042` , `/restaurants/7`). Au **niveau 2**, elle utilise les méthodes et les codes d'état HTTP tels que HTTP les définit. Au **niveau 3**, les réponses portent des contrôles hypermédias : des liens qui indiquent au client ce qu'il peut faire ensuite (une commande passée contient un lien pour l'annuler, jusqu'à ce que l'annulation ne soit plus permise).

Les praticiens sont réellement en désaccord sur le niveau 3. Fielding a toujours soutenu qu'une API sans hypermédia n'est pas du REST. La plupart des API publiques s'arrêtent au niveau 2, car leurs clients sont écrits d'après la documentation plutôt que découverts à l'exécution, et le bénéfice du suivi des liens compense rarement la conception supplémentaire. Une lecture pragmatique est que le niveau 2, appliqué de façon cohérente, apporte l'essentiel de la valeur ; l'hypermédia se justifie quand le serveur doit guider de nombreux clients écrits indépendamment à travers un processus qui évolue.

La sémantique HTTP qui compte pour l'architecture

La sémantique de HTTP est définie dans la RFC 9110 (2022). Deux propriétés des méthodes comptent plus que toutes les autres pour la conception distribuée. Une méthode **sûre** ne fait que lire : GET, HEAD et OPTIONS ne modifient rien que le client ait demandé de modifier. Une méthode idempotente peut modifier l'état, mais la répéter a le même effet voulu que l'envoyer une seule fois : PUT (remplacer cette ressource par cette représentation) et DELETE sont idempotentes, comme toutes les méthodes sûres. POST n'est ni l'une ni l'autre, et PATCH n'est pas idempotente en général. Cette propriété compte parce que les réseaux tombent en panne entre le moment où une requête est traitée et celui où la réponse arrive. La RFC 9110 dit que les requêtes idempotentes peuvent être répétées automatiquement après une telle défaillance, et que les clients ne doivent pas rejouer automatiquement des requêtes non idempotentes, à moins de savoir que la requête peut en fait être répétée sans risque.

Les codes d'état font partie du contrat, car le code client se ramifie en fonction d'eux ; la section 0.6 a présenté les plus courants. Les codes de succès doivent dire ce qui s'est passé : 201 Created (avec un en-tête `Location`) quand une ressource est créée, 202 Accepted quand le travail a seulement été mis en file, 204 No Content pour un succès sans corps. Pour les conflits, 409 signifie que la requête est en contradiction avec l'état actuel de la ressource, 412 qu'une précondition a échoué et 422 qu'une requête bien formée est sémantiquement invalide. La règle architecturale porte sur les nouvelles tentatives : une réponse 4xx échouera de nouveau si la requête est renvoyée sans modification (429 Too Many Requests est l'exception : attendez le délai indiqué, puis réessayez), tandis qu'un délai d'expiration, un 502, un 503 ou un 504 peut réussir plus tard. Les clients ne doivent rejouer que ces cas-là, avec backoff et jitter, et seulement si la requête est idempotente ou porte une clé d'idempotence.

Deux autres fonctionnalités de HTTP résolvent à peu de frais des problèmes distribués. Les **requêtes conditionnelles** offrent un contrôle de concurrence optimiste : un client lit une commande avec son étiquette de version (`ETag`), puis envoie `If-Match` avec cette étiquette lors de la mise à jour ; si quelqu'un d'autre a modifié la commande entre-temps, le serveur répond 412 et rien n'est écrasé. Les **en-têtes de cache** (`Cache-Control`, `ETag`) permettent aux navigateurs, aux CDN et aux proxys de réutiliser les réponses, ce qui est souvent la tactique de performances la moins coûteuse qui soit.

Méthodes HTTP : sûres, idempotentes, ou ni l'un ni l'autre (RFC 9110 ; PATCH d'après la RFC 5789)

Méthode	Usage typique	Sûre ?	Idempotente ?	Rejouable à l'aveugle sans risque ?
GET, HEAD	Lire une ressource	Oui	Oui	Oui
PUT	Créer ou remplacer à une URI connue	Non	Oui	Oui
DELETE	Supprimer une ressource	Non	Oui	Oui (un second appel peut renvoyer 404, mais l'effet est le même)
POST	Créer dans une collection, ou exécuter une action	Non	Non	Seulement avec une clé d'idempotence ou un autre contrôle des doublons
PATCH	Mise à jour partielle	Non	Pas en général	Seulement si le patch est écrit pour être idempotent, ou s'il est conditionné à une version

gRPC : le contrat d'abord, du binaire sur le réseau

gRPC est un framework d'appel de procédure à distance que Google a publié en open source en 2015 pour succéder à Stubby, le système RPC qu'il utilisait en interne depuis plus de dix ans ; la version 1.0 a suivi en 2016. Le contrat est écrit dans un fichier `.proto` à l'aide de Protocol Buffers, un format de sérialisation binaire compact, et des générateurs de code produisent des stubs clients typés et des squelettes de serveur dans de nombreux langages. Les appels passent par HTTP/2, qui multiplexe de nombreuses requêtes simultanées sur une seule connexion.

gRPC propose quatre sortes d'appels : **unaire** (une requête, une réponse), **streaming serveur** (une requête, un flux de réponses, comme les positions des livreurs en direct), **streaming client** (un flux de requêtes, une réponse, comme l'envoi de télémétrie) et **streaming bidirectionnel**. Les clients fixent une **échéance** (deadline) ; une fois celle-ci passée, l'appel se termine par `DEADLINE_EXCEEDED`, et les échéances peuvent être propagées le long d'une chaîne d'appels afin que le travail s'arrête partout dès que plus personne n'attend la réponse.

Ses points forts sont l'efficacité, le typage fort et le code généré, qui le rendent populaire pour les appels internes de service à service dans les organisations polyglottes. Ses coûts en sont le reflet. Les données binaires sont difficiles à lire dans les journaux et avec les outils ordinaires. Les navigateurs ne peuvent pas appeler gRPC directement : les API publiques et exposées au web ont donc généralement besoin d'un proxy ou d'une façade REST ou GraphQL. La mise en cache HTTP ne s'applique pas de la manière habituelle. La compatibilité dépend d'une discipline sur les numéros de champ : c'est le numéro d'un champ, et non son nom, qui l'identifie sur le réseau ; les numéros ne doivent donc jamais être réutilisés, et les champs supprimés doivent être marqués comme réservés.

```
syntax = "proto3";

service Dispatch {
  rpc AssignCourier (AssignRequest) returns (Assignment);
  rpc TrackCourier (TrackRequest) returns (stream Position);
}

message AssignRequest {
  string order_id = 1;
  string restaurant_id = 2;
  reserved 3;            // field removed in v2: never reuse 3
}

message Position {
  double lat = 1;
  double lng = 2;
  int64 recorded_at_ms = 3;
}
```

Un contrat gRPC pour le service d'affectation de la plateforme de livraison de repas, avec un appel unaire et un appel en streaming serveur (syntaxe Protocol Buffers).

GraphQL : le client choisit la forme

GraphQL est un langage de requête et un environnement d'exécution pour les API que Facebook a développé en interne à partir de 2012 pour ses applications mobiles, puis publié en 2015 avec une ébauche de spécification et une implémentation de référence ; il a été confié en 2018 à la GraphQL Foundation, neutre vis-à-vis des fournisseurs. Le serveur publie un **schéma** typé ; les clients envoient à un endpoint unique des requêtes qui nomment exactement les champs voulus, éventuellement à travers des objets liés, et reçoivent une réponse JSON de la même forme. Les mutations modifient les données, et les abonnements envoient des mises à jour.

Son principal avantage concerne les clients variés. Une application mobile de livraison de repas sur une connexion lente peut demander l'état d'une commande, le nom du restaurant et le prénom du livreur en un seul aller-retour, sans recevoir les vingt autres champs dont le tableau de bord web a besoin. Cela évite à la fois la récupération excessive de données (over-fetching) et la chaîne de requêtes qu'une API REST ressource par ressource peut exiger, et cela peut remplacer un backend distinct pour chaque frontend. Le schéma est en outre autodocumenté et permet aux outils de valider les requêtes avant leur exécution.

Les compromis sont réels et sont surtout supportés par l'équipe serveur. Chaque champ est rempli par un **résolveur** (resolver) : une implémentation naïve qui récupère séparément le restaurant de chaque commande produit le problème des requêtes N+1 : une requête pour cinquante commandes, puis cinquante de plus pour leurs restaurants. La solution habituelle est

le regroupement des requêtes (batching) et un cache par requête, popularisés par la bibliothèque DataLoader. Comme les clients peuvent composer des requêtes arbitrairement profondes, les serveurs ont besoin de limites de profondeur et de coût des requêtes, de délais d'expiration et souvent d'une liste d'autorisation de requêtes approuvées (persistées) pour empêcher une surcharge accidentelle ou malveillante. La mise en cache HTTP est moins efficace quand tout est un POST vers une seule URL. L'autorisation doit être appliquée par champ ou par objet plutôt que par URL. Enfin, les erreurs sont généralement signalées dans un tableau `errors` à l'intérieur d'une réponse HTTP réussie, ce qui échappe à une supervision fondée sur les codes d'état.

Messagerie : files et topics

La messagerie déplace la conversation dans un courtier. Comme l'explique l'architecture pilotée par les événements, une file remet chaque message à l'un des consommateurs concurrents, la méthode standard pour faire monter en charge un pool de workers, tandis qu'un topic en remet une copie à chaque abonné : les services des restaurants, d'affectation et de notifications peuvent ainsi réagir chacun à `OrderPlaced` sans que le service des commandes sache qu'ils existent. Cette section-là traite aussi des garanties de livraison, de l'ordre, des doublons et des files de lettres mortes ; files d'attente et back pressure explique comment les files absorbent les pics de charge.

La messagerie peut aussi transporter des requêtes et des réponses : le demandeur envoie une commande avec un **identifiant de corrélation** et une adresse de réponse, puis associe la réponse, quand elle arrive, à sa requête. Cela conserve le découplage temporel d'un courtier tout en obtenant une réponse, au prix de pièces mobiles supplémentaires.

REST, gRPC, GraphQL et messagerie comparés

Aspect	REST sur HTTP	gRPC	GraphQL	Messagerie (files et topics)
Interaction	Requête-réponse synchrone	Synchrone ; unaire ou en streaming	Requêtes et mutations synchrones ; les abonnements poussent les données	Asynchrone ; fire-and-forget ou requête-réponse via des identifiants de corrélation
Contrat	Document OpenAPI (facultatif mais courant)	Fichiers .proto (obligatoires)	Schéma (obligatoire)	Document AsyncAPI et schémas de messages (facultatifs mais conseillés)
Charge utile	Généralement du texte JSON	Binaire Protocol Buffers	JSON	Tous : JSON, Avro, Protocol Buffers
Couplage temporel	Oui	Oui	Oui	Non : le courtier sert de tampon
Mise en cache	Excellente : caches HTTP standard et CDN	Peu de choses intégrées	Plus difficile : nécessite des requêtes persistées ou des caches côté client	Sans objet
Prise en charge par les navigateurs	Native	Nécessite un proxy (gRPC-Web)	Native	Pas directement ; via un backend ou des WebSockets
Usage idéal	API publiques et partenaires ; CRUD orienté ressources	Appels internes entre services, à fort volume et faible latence ; streaming	Nombreux types de clients aux besoins de données différents ; agrégation de plusieurs services	Travail qui peut être fait plus tard ; fan-out vers de nombreux consommateurs ; lissage de la charge
Principaux pièges	API bavardes ; conventions incohérentes	Opaque au débogage ; discipline sur les numéros de champ	Résolveurs N+1 ; coût des requêtes non borné ; autorisation au niveau des champs	Doublons, ordre, flux invisibles, exploitation du courtier

Le paysage de la messagerie : choisir une file ou un journal

La première décision se joue entre une file et un journal. Une **file** distribue du travail, et un message disparaît une fois traité, ce qui convient à des tâches comme l'envoi d'un reçu, où comptent la nouvelle tentative par message et la mise en file de lettres mortes. Un **journal** conserve les messages pour des lecteurs ultérieurs, ce qui convient aux faits dont plusieurs consommateurs ont besoin ou qui doivent être rejoués, comme les événements de commande qui reconstruisent un modèle de lecture. La section 6.4 présente Kafka, le journal le plus connu. La frontière s'est estompée : Kafka 4.2 a déclaré prêts pour la production ses share groups, proches d'une file, et RabbitMQ a ajouté des streams en ajout seul.

Quatre autres noms reviennent souvent. **RabbitMQ** est un courtier que vous exploitez vous-même ou achetez sous forme de service hébergé ; ses files conviennent à la répartition du travail, et ses streams ajoutent le rejeu et un large fan-out. **Amazon SQS et SNS** sont des services managés : les consommateurs interrogent une file SQS, et un topic SNS pousse une copie vers chaque file abonnée, de sorte qu'un événement atteint les files des restaurants, de l'affectation et des notifications, qui se vident chacune à leur rythme. **Google Cloud Pub/Sub** est un service managé de publication-abonnement avec livraison au moins une fois. **NATS** est léger : le cœur de NATS livre au plus une fois, donc un abonné non connecté ne voit jamais le message, tandis que sa couche JetStream ajoute le stockage, le rejeu et les accusés de réception. Les limites et les fonctionnalités changent : vérifiez donc la version actuelle de chacun.

Les consommateurs concurrents et l'ordre strict tirent dans des directions opposées. Même si un courtier distribue les messages dans l'ordre, les workers terminent dans n'importe quel ordre : un worker lent qui détient `OrderPaid` peut terminer après un worker rapide qui détient `OrderCancelled`. La nouvelle livraison s'y ajoute ; la documentation de RabbitMQ avertit qu'avec plusieurs consommateurs actifs « toute nouvelle livraison peut changer l'ordre ». La réponse habituelle consiste à renoncer à l'ordre global et à conserver l'ordre par clé, avec au plus un worker par clé à la fois. Kafka le fait avec des partitions à clé et les files SQS FIFO avec des groupes de messages : l'ordre est garanti au sein d'un groupe, et les groupes s'exécutent en parallèle. Les files SQS standard n'offrent qu'un ordre au mieux. Pub/Sub utilise des clés d'ordre, activées sur l'abonnement. Le consommateur actif unique (single active consumer) de RabbitMQ préserve l'ordre en ne laissant qu'un seul consommateur recevoir depuis la file, ce qui y sacrifie le parallélisme.

L'ordre par clé a un prix. Un message lent ou en échec retarde tous les messages suivants de cette clé, et une clé très active accumule un arriéré ; la documentation de Pub/Sub précise aussi que la nouvelle livraison d'un message entraîne celle des messages suivants pour cette clé, même ceux déjà acquittés. La livraison ordonnée ne supprime pas non plus les doublons : les consommateurs doivent donc rester idempotents. Lorsqu'une mise à jour périmée est

reconnaissable, une autre solution consiste à placer un numéro de version dans chaque événement et à laisser les consommateurs ignorer tout ce qui est plus ancien que ce qu'ils ont appliqué.

Concevoir des API qui durent : ressources, pagination et filtrage

Modélisez les ressources autour du domaine, pas de la base de données. Exposez les noms avec lesquels vos consommateurs raisonnent (`/orders` , `/orders/1042/items` , `/restaurants/7/menu`), pas des tables ni des services internes. Les actions qui n'entrent pas dans les catégories créer, lire, mettre à jour et supprimer peuvent généralement être modélisées elles-mêmes comme des ressources : `POST /orders/1042/cancellation` crée une annulation, qui peut porter un motif et être refusée avec un 409 si le repas est déjà en route. Cela préserve l'interface uniforme tout en exprimant honnêtement le processus métier.

Paginez chaque collection, car les collections grandissent. La **pagination par offset** (`?offset=40&limit=20`) est simple et permet de sauter à la page 9, mais elle ralentit sur les pages lointaines, car la base de données doit sauter toutes les lignes précédentes, et elle affiche des doublons ou saute des éléments quand des lignes sont insérées pendant que quelqu'un parcourt les pages. La pagination par curseur renvoie un jeton opaque qui pointe juste après le dernier élément (`?after=eyJpZCI6MTA0Mn0&limit=20`) ; elle reste rapide et stable en cas d'insertions, au prix de l'absence d'accès direct. Pour les fils d'actualité et les données volumineuses ou changeantes, les curseurs sont le choix par défaut le plus sûr. Plafonnez toujours la taille des pages côté serveur.

Le filtrage et le tri doivent utiliser des paramètres de requête prévisibles (`?status=delivered&placed_after=2026-10-01&sort=-placed_at`) et s'appuyer sur des index ; une API qui laisse les clients filtrer sur n'importe quel champ invite des requêtes auxquelles la base de données ne peut pas répondre rapidement. Ne prenez en charge que les filtres dont les consommateurs ont besoin, et documentez-les dans le contrat.

Les traitements de longue durée : 202, polling et webhooks fiables

Certaines requêtes prennent trop de temps pour rester ouvertes, comme l'importation de toute la carte d'une chaîne de restaurants. L'API vérifie la requête et renvoie `202 Accepted` avec un en-tête `Location` indiquant une ressource d'état, et éventuellement `Retry-After` . La RFC 9110 qualifie 202 d'« intentionnellement non engageant » : le travail peut encore être refusé plus tard. Le client interroge (polling) et reçoit 200 avec l'état, puis 303 vers le résultat. Pour éviter le polling, le serveur peut plutôt appeler un **webhook**. L'émetteur doit réessayer avec backoff et jitter, garder un même identifiant d'événement d'une tentative à l'autre et signer

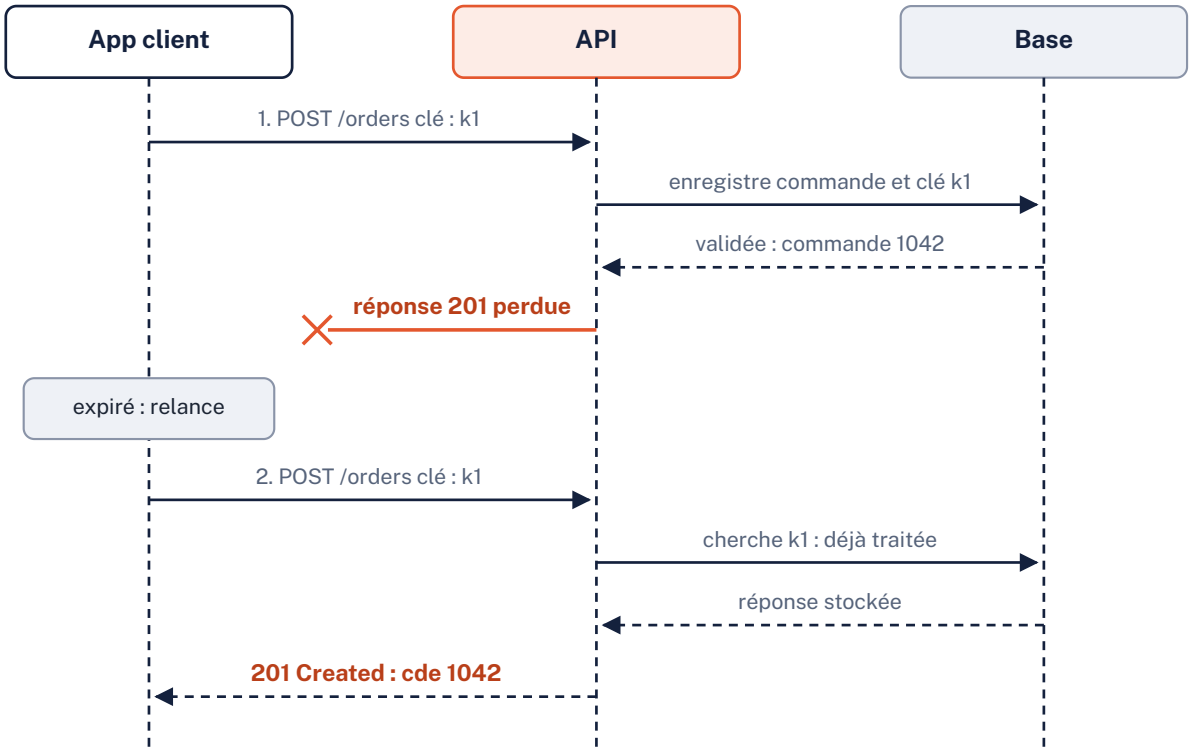
chaque livraison avec un horodatage. Le récepteur doit vérifier la signature (section 8.4), répondre 2xx rapidement, mettre le travail en file et ignorer les doublons. Stripe, par exemple, ne garantit pas l'ordre. Gardez l'endpoint d'état pour récupérer les livraisons manquées.

Clés d'idempotence : rendre POST rejouable sans risque

La défaillance partielle (section 6.1) laisse un client qui n'a reçu aucune réponse dans l'incertitude : sa requête a-t-elle été traitée ? Rejouer un GET est sans danger ; rejouer `POST /orders` peut créer une seconde commande et débiter le client deux fois. Une clé d'idempotence transforme cette nouvelle tentative risquée en une nouvelle tentative sûre. Le client génère une valeur unique (généralement un UUID aléatoire) pour chaque opération logique et l'envoie dans un en-tête tel que `Idempotency-Key`. Le serveur enregistre la clé avec le résultat de la première requête ; toute requête ultérieure portant la même clé renvoie le résultat enregistré au lieu de refaire le travail.

Les prestataires de paiement ont popularisé cette technique, et le groupe de travail HTTP APIs de l'IETF a rédigé un projet d'en-tête `Idempotency-Key` standard. À l'heure où ces lignes sont écrites, il n'a pas été publié en tant que RFC (le dernier projet, d'octobre 2025, a expiré), mais ses règles reflètent la pratique courante : une clé réutilisée avec un corps de requête différent doit être rejetée avec un 422, une nouvelle tentative qui arrive pendant que la première requête est encore en cours doit recevoir un 409 Conflict afin que deux copies ne s'exécutent jamais en même temps, et les clés sont propres à un client et expirent après une durée publiée, souvent 24 heures. L'exemple commenté applique chacune de ces règles.

Une commande rejouée avec une clé d'idempotence



La première tentative réussit, mais sa réponse est perdue ; la nouvelle tentative porte la même clé, si bien que l'API renvoie le résultat stocké au lieu de créer la commande 1043.

EXEMPLE COMMENTÉ

Exemple commenté : le double appui au moment de payer

Un client dont la connexion mobile est faible appuie sur « Commander ». L'app génère la clé `7f3c…` et envoie `POST /v1/orders`. Le service des commandes stocke la clé avec une empreinte (hash) du corps de la requête et, plus tard, la réponse.

Tentative 1	La commande 1042 est créée et le paiement autorisé ; la clé, l'empreinte du corps et la réponse 201 sont enregistrées dans la même transaction de base de données que la commande. La réponse est perdue quand le téléphone passe du Wi-Fi aux données mobiles.
Tentative 2 (nouvelle tentative automatique, même clé)	Le service trouve la clé avec le statut *completed*, vérifie que l'empreinte du corps correspond, et renvoie le 201 stocké pour la commande 1042. Pas de seconde commande, pas de second débit.
Tentative 3 (le client appuie de nouveau pendant que la tentative 2 est en cours)	L'app réutilise la même clé pour le même panier. Si la première requête était encore en cours de traitement, le service répondrait 409, et l'app attendrait avant de réessayer.
Un bogue envoie la même clé avec un panier différent	Les empreintes des corps diffèrent : le service répond 422 au lieu de renvoyer silencieusement la mauvaise commande.
Au bout de 24 heures	La clé expire et est supprimée ; l'app ne rejoue jamais cet ancien panier.

La clé doit être générée une seule fois par opération logique (par soumission de panier), et non à chaque tentative HTTP ; une nouvelle clé à chaque nouvelle tentative va à l'encontre du but recherché.

C'est l'enregistrement de la clé dans la même transaction que la commande qui rend la vérification fiable : un plantage ne peut laisser ni une commande sans sa clé, ni une clé sans sa commande.

La même idée, appliquée aux messages plutôt qu'aux requêtes HTTP, est le consommateur idempotent de l'architecture pilotée par les événements.

```
def create_order(request, db):
    key = request.headers.get("Idempotency-Key")
    if not key:
        return problem(400, "missing-idempotency-key")
    body_hash = sha256(request.body)
    with db.transaction():
        row = db.find_idempotency(request.client_id, key)
        if row and row.body_hash ≠ body_hash:
            return problem(422, "idempotency-key-reused")
        if row and row.status == "in_progress":
            return problem(409, "request-in-progress")
        if row:
            return row.saved_response          # replay, do nothing new
        db.insert_idempotency(request.client_id, key, body_hash, "in_progress")
        order = db.insert_order(request.json)
        response = created(order)
        db.complete_idempotency(request.client_id, key, response)
    return response
```

Traitement côté serveur d'une clé d'idempotence (simplifié : une transaction de base de données par requête ; la contrainte d'unicité sur la clé empêche deux requêtes simultanées de s'exécuter toutes les deux).

Des erreurs lisibles par les machines : la RFC 9457

Les clients ont besoin d'erreurs sur lesquelles ils peuvent agir par programme, et les humains d'erreurs qu'ils peuvent comprendre. Plutôt que de laisser chaque équipe inventer sa propre forme d'erreur, utilisez Problem Details for HTTP APIs, la RFC 9457 (juillet 2023), qui a remplacé la RFC 7807 (2016). Un objet problem details a le type de média `application/problem+json` et cinq membres standard : `type` (une URI qui identifie le type de problème et peut renvoyer vers de la documentation), `title` (un résumé court et lisible par un humain de ce type de problème), `status` (le code d'état HTTP), `detail` (une explication de cette occurrence) et `instance` (une URI qui identifie cette occurrence). Des membres d'extension peuvent porter des précisions lisibles par une machine, par exemple les champs qui n'ont pas passé la validation.

Deux règles rendent les réponses d'erreur sûres. Le code client doit se ramifier en fonction de `type` et de `status`, jamais du libellé de `title` ou de `detail`, qui peut être traduit ou amélioré. Et les erreurs ne doivent pas divulguer d'éléments internes : les traces de pile, les fragments SQL et les noms d'hôtes internes aident davantage les attaquants que les appelants. Journalisez les détails avec un identifiant de corrélation et renvoyez plutôt cet identifiant.

```
{
  "type": "https://api.fooddelivery.example/problems/outside-delivery-area",
  "title": "Address is outside the delivery area",
  "status": 422,
  "detail": "Restaurant 7 delivers within 4 km; the address is 6.2 km away.",
  "instance": "/v1/orders/attempts/9b1e",
  "max_distance_km": 4,
  "actual_distance_km": 6.2
}
```

Une réponse problem details (HTTP 422, Content-Type: application/problem+json) pour une commande hors de la zone de livraison.

Faire évoluer une API sans casser ses consommateurs

Une API qui a des consommateurs est une promesse, et la loi de Hyrum (section 0.6) avertit que quelqu'un dépend de chaque comportement observable. Le changement est donc régi par un seul principe : faites librement des changements rétrocompatibles et évitez ceux qui cassent la compatibilité. Les consommateurs y contribuent en étant des **lecteurs tolérants** (tolerant readers, un terme qu'emploie Martin Fowler) : ils ignorent les champs qu'ils ne comprennent pas et ne lisent que ce dont ils ont besoin, si bien que les ajouts ne les cassent jamais. Le tableau ci-dessous résume les règles habituelles ; la même logique, appliquée aux données stockées et aux schémas d'événements, apparaît dans l'évolution des schémas.

Quand un changement cassant est inévitable, versionnez. Les stratégies courantes sont une version dans le chemin de l'URL (`/v2/orders` : explicite, facile à router et à mettre en cache, mais qui encourage de grands sauts peu fréquents), un en-tête personnalisé ou un paramètre de type de média (`Accept: application/vnd.fooddelivery.v2+json` : garde les URL stables, mais est moins visible et plus difficile à tester dans un navigateur), et des versions datées fixées pour chaque compte client, que certains grands fournisseurs d'API utilisent pour publier en continu de petits changements cassants, chaque client faisant la mise à niveau quand il le décide. Les API internes entre services appartenant à la même organisation évitent souvent complètement les versions grâce à l'approche d'**expansion-contraction** (expand and contract) : ajouter le nouveau champ ou le nouvel endpoint, migrer chaque consommateur, puis supprimer l'ancien. Quelle que soit la stratégie, annoncez les dépréciations à l'avance ; les en-têtes de réponse `Deprecation` (RFC 9745, 2025) et `Sunset` (RFC 8594, 2019) permettent à un serveur d'indiquer aux clients, sous une forme lisible par une machine, qu'un endpoint est déprécié et quand il cessera de fonctionner.

Changements rétrocompatibles et changements cassants d'une API

Changement	Compatible ?	Pourquoi
Ajouter un champ de requête ou un paramètre de requête facultatif	Oui	Les clients existants ne l'envoient pas ; le serveur doit appliquer une valeur par défaut raisonnable.
Ajouter un champ à une réponse	En général	Les lecteurs tolérants l'ignorent ; les clients stricts qui rejettent les champs inconnus casseront : documentez donc que les clients doivent tolérer les ajouts.
Ajouter un nouvel endpoint ou une nouvelle ressource	Oui	Personne ne l'appelle encore.
Ajouter une valeur à une énumération dans une réponse	Risqué	Les clients dotés d'instructions switch exhaustives peuvent échouer sur la valeur inconnue ; documentez que de nouvelles valeurs peuvent apparaître.
Rendre obligatoire un champ de requête facultatif	Non	Les clients existants qui l'omettent se mettent à échouer avec un 400.
Supprimer ou renommer un champ, ou changer son type ou ses unités	Non	Les clients qui lisent l'ancien champ n'obtiennent rien, ou une valeur erronée.
Changer la signification d'un champ ou d'un code d'état	Non	La rupture la plus dangereuse : rien n'échoue visiblement, mais le comportement est faux.
Durcir la validation (longueur maximale plus courte, format plus strict)	Non	Des requêtes qui réussissaient échouent désormais.

Décrire les contrats : OpenAPI et AsyncAPI

Un contrat écrit dans un format lisible par une machine peut piloter la documentation, la génération de clients, les stubs serveur, les serveurs simulés (mock), le linting et les tests. OpenAPI est la norme pour les API HTTP. Elle est issue de la spécification Swagger, que SmartBear a donnée en 2015 à l'OpenAPI Initiative, nouvellement créée sous l'égide de la Linux Foundation, et qui a été renommée en 2016 ; la version 3.1 (2021) a aligné ses schémas sur JSON Schema, et la version 3.2 a suivi en septembre 2025. AsyncAPI applique la même idée aux API pilotées par messages : elle décrit les canaux (files et topics), les opérations qui y envoient ou y reçoivent des messages et les schémas de messages ; la version 3.0 est sortie en décembre 2023.

Les équipes choisissent entre l'approche **design-first** (rédiger et relire le contrat, puis l'implémenter) et l'approche **code-first** (générer le contrat à partir de code annoté). Le design-first fait de l'API une décision délibérée que les consommateurs peuvent examiner avant que quoi que ce soit ne soit construit, et convient aux API publiques et inter-équipes ; le code-first maintient le contrat et l'implémentation en phase avec moins d'effort, et convient aux API internes qui changent vite. Dans les deux cas, gardez le contrat en gestion de versions, passez-le au linter dans le pipeline et lancez une vérification des changements cassants qui compare chaque version proposée à la version publiée.

Tests de contrat et contrats pilotés par le consommateur

Les tests de bout en bout qui déploient tous les services ensemble sont lents, instables, et deviennent plus lents à mesure que les services se multiplient ; pourtant, les tests unitaires ne peuvent pas détecter qu'un fournisseur a modifié un champ dont dépend un consommateur. Les tests de contrat comblent ce manque : chaque partie est testée isolément par rapport à une description partagée de leurs interactions. Avec les contrats pilotés par le consommateur, un patron décrit par Ian Robinson en 2006, chaque consommateur déclare les parties de l'API du fournisseur qu'il utilise réellement, et le fournisseur exécute toutes les attentes de ses consommateurs dans son propre build. Le fournisseur sait alors exactement quels champs personne ne lit (et peut donc les modifier sans risque) et quels changements casseraient un consommateur nommément identifié.

Pact, créé chez realestate.com.au en 2013, est l'outil le plus connu. Les tests du consommateur s'exécutent contre un fournisseur simulé (mock) Pact et enregistrent les requêtes envoyées et les réponses attendues dans un fichier de contrat ; le pipeline du fournisseur rejoue ces requêtes contre le véritable fournisseur et vérifie les réponses. Un broker stocke les contrats et les résultats de vérification : avant de déployer, un pipeline peut ainsi demander si cette version d'un service est compatible avec les versions de ses consommateurs et de ses fournisseurs déjà en production. Les tests de contrat ne remplacent pas quelques tests de fumée (smoke tests) de bout en bout, et ils vérifient la structure plutôt que la signification métier, mais ils permettent de déployer les services indépendamment en toute confiance, ce qui est précisément l'intérêt de les avoir séparés.

ATTENTION

Erreurs courantes dans la communication entre services

- Construire de longues chaînes synchrones pour un travail que l'utilisateur n'attend pas, si bien qu'un seul service lent ralentit toute la requête.
- Rejouer automatiquement des appels non idempotents, puis découvrir des commandes en double et des doubles débits.
- Rejouer des erreurs 4xx, ou rejouer sans backoff ni jitter, ce qui transforme une brève panne en une surcharge auto-infligée.
- Exposer des tables de base de données comme ressources, si bien que chaque modification du schéma devient une modification de l'API.
- Renvoyer des collections non bornées sans pagination, ce qui fonctionne en test et échoue en production.
- Choisir GraphQL ou gRPC par effet de mode, puis payer pour des requêtes N+1, des limites de coût des requêtes ou des proxys pour navigateurs dont le système n'avait pas besoin.
- Faire des changements cassants dans une API interne parce que « nous possédons les deux côtés », alors qu'une équipe ailleurs l'appelait discrètement.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi placer une file entre deux services ne supprime-t-il pas tout couplage entre eux ?

Elle supprime le couplage temporel, mais les deux services dépendent toujours du format et de la signification du message, ainsi que du courtier lui-même : une modification du message peut donc encore casser le destinataire.

▼ Pourquoi DELETE est-elle idempotente alors qu'un second DELETE peut renvoyer 404 au lieu de 204 ?

L'idempotence concerne l'effet voulu sur le serveur, pas la réponse : après un ou deux DELETE, la ressource a disparu dans les deux cas.

▼ Un client mobile génère une nouvelle clé d'idempotence à chaque nouvelle tentative. Qu'est-ce qui ne va pas ?

Le serveur voit chaque nouvelle tentative comme une nouvelle opération : une réponse perdue suivie d'une nouvelle tentative peut donc encore créer une commande en double ; la clé doit appartenir à l'opération logique, pas à la tentative HTTP.

▼ Pourquoi la pagination par curseur peut-elle afficher une liste stable quand de nouvelles commandes arrivent pendant que l'utilisateur parcourt les pages, alors que la pagination par offset peut ne pas le faire ?

Le curseur pointe juste après un élément précis : des insertions ailleurs ne décalent donc pas la page suivante ; un offset compte des lignes : de nouvelles lignes font passer des éléments d'une page à l'autre, ce qui provoque des doublons ou des éléments sautés.

▼ À vous : Choisissez un style pour chaque cas : l'app client passe une commande ; deux services internes très sollicités échangent des appels de tarification ; cinq services doivent réagir quand une commande est passée. Une nouvelle tentative mobile du premier cas serait-elle sûre ? Donnez vos raisons.

Passer une commande : REST sur HTTP, avec un contrat public stable, mais POST n'est pas idempotent, si bien qu'une nouvelle tentative n'est sûre qu'avec une clé d'idempotence. Appels internes de tarification : gRPC convient aux appels internes fréquents, avec du code généré et des messages compacts. Services qui réagissent : publiez sur un topic, afin que chaque abonné reçoive sa propre copie et que le service de commandes n'ait pas besoin de les connaître. Une file donnerait chaque message à un seul consommateur, ce qui ne convient pas ici.

SOURCES

- RFC 9110: HTTP Semantics (IETF, 2022): https://www.rfc-editor.org/rfc/rfc9110.html
- RFC 9457: Problem Details for HTTP APIs (IETF, 2023): https://www.rfc-editor.org/rfc/rfc9457.html
- Martin Fowler: Richardson Maturity Model: https://martinfowler.com/articles/richardsonMaturityModel.html
- gRPC documentation: Core concepts, architecture and lifecycle: https://grpc.io/docs/what-is-grpc/core-concepts/
- Ian Robinson: Consumer-Driven Contracts: A Service Evolution Pattern: https://martinfowler.com/articles/consumerDrivenContracts.html
- Apache Kafka 4.2.0 release announcement: https://kafka.apache.org/blog/2026/02/17/apache-kafka-4.2.0-release-announcement/
- RabbitMQ: Queues: https://www.rabbitmq.com/docs/queues
- RabbitMQ: Streams: https://www.rabbitmq.com/docs/streams
- RabbitMQ: Consumers: https://www.rabbitmq.com/docs/consumers
- Amazon SQS Developer Guide: queue types: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-queue-types.html

- Amazon SNS Developer Guide: fanout to Amazon SQS queues: https://docs.aws.amazon.com/sns/latest/dg/sns-sqs-as-subscriber.html
- Google Cloud Pub/Sub: Order messages: https://docs.cloud.google.com/pubsub/docs/ordering
- NATS: Core NATS: https://docs.nats.io/nats-concepts/core-nats
- NATS: JetStream: https://docs.nats.io/nats-concepts/jetstream
- RFC 9110: HTTP Semantics, section 15.3.3: https://www.rfc-editor.org/rfc/rfc9110.txt
- Microsoft Learn: Asynchronous Request-Reply pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/asynchronous-request-reply
- Stripe: Receive events in your webhook endpoint: https://docs.stripe.com/webhooks
- Standard Webhooks specification: https://raw.githubusercontent.com/standard-webhooks/standard-webhooks/main/spec/standard-webhooks.md

SECTION 6.6 SUR 6.8 · AVANCÉ · 30 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Modèles de cohérence, CAP et PACELC

AVANT DE LIRE

Cette section s'appuie sur :

- 0.5 Stocker les données
- 2.5 Disponibilité et fiabilité
- 6.1 Pourquoi distribuer

Termes déjà vus qu'elle utilise : ACID, Transaction, Disponibilité, Latence, Invariant

L'IDÉE EN BREF

Imaginez un magasin à trois succursales qui tiennent chacune une copie de la liste des stocks et se téléphonent pour se communiquer les mises à jour. Si la ligne téléphonique est coupée et qu'un client demande à une succursale si le dernier vélo est en stock, cette succursale a deux possibilités : dire « Je ne peux pas en être sûre, veuillez patienter », ou répondre d'après sa propre liste et risquer de vendre un vélo qu'une autre succursale vient de vendre. Même quand les téléphones fonctionnent, appeler les autres succursales avant chaque réponse fait attendre les clients. Les systèmes distribués sont confrontés exactement à ces choix, et le bon choix dépend de ce que coûterait une mauvaise réponse.

CE QU'IL FAUT SAVOIR

- Un modèle de cohérence est le contrat qui indique quelles valeurs périmées ou réordonnées les clients peuvent observer ; le C de CAP (copie unique, linéarisable) diffère du C d'ACID (état valide) et des niveaux d'isolation.
- Du plus fort au plus faible : linéarisable (une seule copie, ordre en temps réel), séquentiel (un ordre convenu), causal (les causes avant les effets), garanties de session comme la lecture de ses propres écritures et les lectures monotones, et à terme (convergence si les écritures cessent).
- Le théorème CAP (conjecture de Brewer en 2000, démontrée par Gilbert et Lynch en 2002) dit que, pendant une partition réseau, un système répliqué doit choisir, pour chaque requête, entre cohérence et disponibilité ; les partitions ne sont pas facultatives : « en choisir deux sur trois » est donc une erreur d'interprétation.
- L'article de Brewer de 2012 recommande de détecter les partitions, de fonctionner dans un mode partition explicite et de rétablir la situation ensuite par des fusions et des compensations, en faisant des choix différents selon les opérations.

- PACELC (Abadi, 2010 et 2012) ajoute que, sinon (else), en l'absence de partition, les systèmes arbitrent entre latence et cohérence à chaque requête.
- Avec N répliques, W accusés de réception d'écriture et R réponses de lecture, choisir R + W > N fait se recouper les ensembles de lecture et d'écriture, mais les quorums seuls ne sont pas linéarisables.
- Choisissez la cohérence selon le cas d'usage : forte pour l'argent, les ressources uniques et les invariants ; garanties de session pour les propres données d'un utilisateur ; à terme pour les compteurs, les fils d'actualité et les positions.

Ce que « cohérent » veut dire une fois les données copiées

Dès qu'un système conserve plus d'une copie d'une donnée, sur des répliques situées dans différentes zones, dans des caches ou dans d'autres services, les copies peuvent diverger pendant un certain temps. Un modèle de cohérence est le contrat qui indique quelles divergences les clients peuvent observer. C'est une promesse sur ce que les lectures peuvent renvoyer compte tenu des écritures qui ont eu lieu, et donc une promesse sur les surprises que le code applicatif doit gérer.

Attention à ce mot aux sens multiples. Le C d'ACID (section 0.5) signifie qu'une transaction fait passer la base de données d'un état valide à un autre, en respectant des règles comme « un solde ne descend jamais en dessous de zéro ». Le C de CAP signifie quelque chose de différent et de plus restreint : chaque lecture se comporte comme s'il existait une copie unique et à jour. Les niveaux d'isolation des bases de données, comme sérialisable ou read committed, forment une troisième famille, qui concerne les transactions concurrentes ; ils sont traités dans transactions et isolation. Cette section porte sur les garanties des données répliquées.

Les modèles de cohérence, du plus fort au plus faible

La linéarisabilité, définie par Maurice Herlihy et Jeannette Wing en 1990, est le modèle à objet unique le plus fort. Chaque opération semble prendre effet instantanément à un moment situé entre son début et sa fin, et tous les clients voient le même ordre, qui respecte le temps réel : une fois qu'une écriture est terminée, chaque lecture qui commence ensuite la voit, ou voit une valeur plus récente. Le système se comporte comme s'il n'y avait qu'une seule copie. Exemple de livraison de repas : deux clients tentent d'utiliser le dernier code de réduction ; avec un stockage linéarisable, un seul des deux réussit et l'autre constate que le code n'est plus disponible, quelle que soit la réplique qu'il atteint.

La cohérence séquentielle, définie par Leslie Lamport en 1979 pour les multiprocesseurs, conserve l'ordre global unique et conserve les opérations de chaque client dans l'ordre où ce client les a émises, mais abandonne l'exigence de temps réel. Un client peut lire une valeur

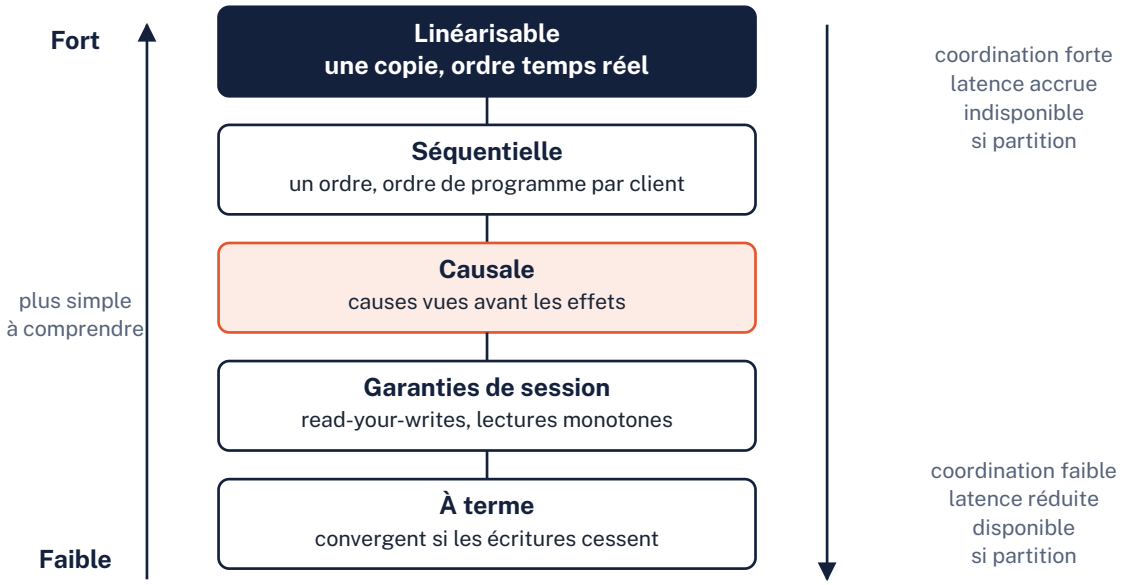
légèrement périmée, tant que tout le monde s'accorde sur l'ordre de toutes les écritures. Elle est rarement proposée sous ce nom dans les bases de données distribuées, mais elle explique l'écart entre « un ordre convenu » et « un ordre convenu qui correspond à l'horloge ».

La cohérence causale exige seulement que les opérations liées causalement soient vues dans le même ordre par tout le monde : si l'écriture de B dépendait du fait d'avoir vu A, personne ne doit voir B sans A. Des écritures concurrentes et sans lien peuvent être vues dans des ordres différents par différents clients. Exemple : un restaurant répond « Oui, c'est sans gluten » à la question d'un client dans la messagerie de la commande ; personne ne doit jamais voir la réponse sans la question. La cohérence causale est intéressante, car la carte des modèles de cohérence de Jepsen, très utilisée, la classe comme atteignable avec une disponibilité dite *sticky* : elle peut continuer à fonctionner pendant une partition, tant que chaque client reste sur la même réplique.

Les garanties de session, décrites par Douglas Terry et ses collègues en 1994, sont des promesses plus faibles, propres à chaque client, qui éliminent les anomalies les plus déroutantes. Lecture de ses propres écritures (read-your-writes) : après qu'un client a modifié l'adresse de livraison, la lecture suivante de ce client affiche la nouvelle adresse, même si d'autres utilisateurs la voient plus tard. Lectures monotones : une fois qu'un client a vu l'état de commande « en cours de livraison », il ne voit jamais plus tard « en préparation » parce qu'une lecture ultérieure a atteint une réplique en retard. (Deux autres garanties, les écritures monotones et les écritures qui suivent les lectures, contraignent l'ordre dans lequel les écritures d'un client sont appliquées.)

La cohérence à terme (eventual consistency), popularisée par Werner Vogels en 2008, est la plus faible des garanties utiles : si aucune nouvelle écriture n'arrive, toutes les répliques finissent par converger vers la même valeur. Elle ne dit rien du temps que cela prend ni de ce qui peut être lu entre-temps, et une implémentation de base qui résout les écritures en conflit en gardant celle qui a l'horodatage le plus récent, appelée last write wins (la dernière écriture l'emporte), écarte silencieusement l'autre. Des formes plus fortes, comme la cohérence à terme forte avec des types de données répliqués sans conflit (CRDT), garantissent que les répliques qui ont reçu les mêmes mises à jour ont le même état, en utilisant des types de données dont les mises à jour concurrentes fusionnent toujours, comme les compteurs et les ensembles.

L'échelle de la cohérence



Chaque échelon vers le bas retire une garantie et une part de coordination ; les garanties de session sont des promesses propres à chaque client, qui se situent entre la cohérence causale et la simple cohérence à terme.

SI VOUS DÉBUTEZ

Si vous débutez

Pensez à une liste de courses familiale tenue sur trois téléphones qui se synchronisent dès qu'ils ont du réseau. Linéarisable signifie que la liste se comporte comme s'il n'y avait qu'une seule feuille de papier sur le réfrigérateur. À terme signifie que les téléphones finissent par être d'accord une fois que chacun a eu du réseau pendant un moment, mais d'ici là, quelqu'un peut acheter du lait que quelqu'un d'autre a déjà acheté. Tout ce qui se trouve entre les deux est un compromis différent entre « toujours d'accord » et « toujours fonctionnel, même dans un tunnel ».

CAP : ce qu'il dit vraiment

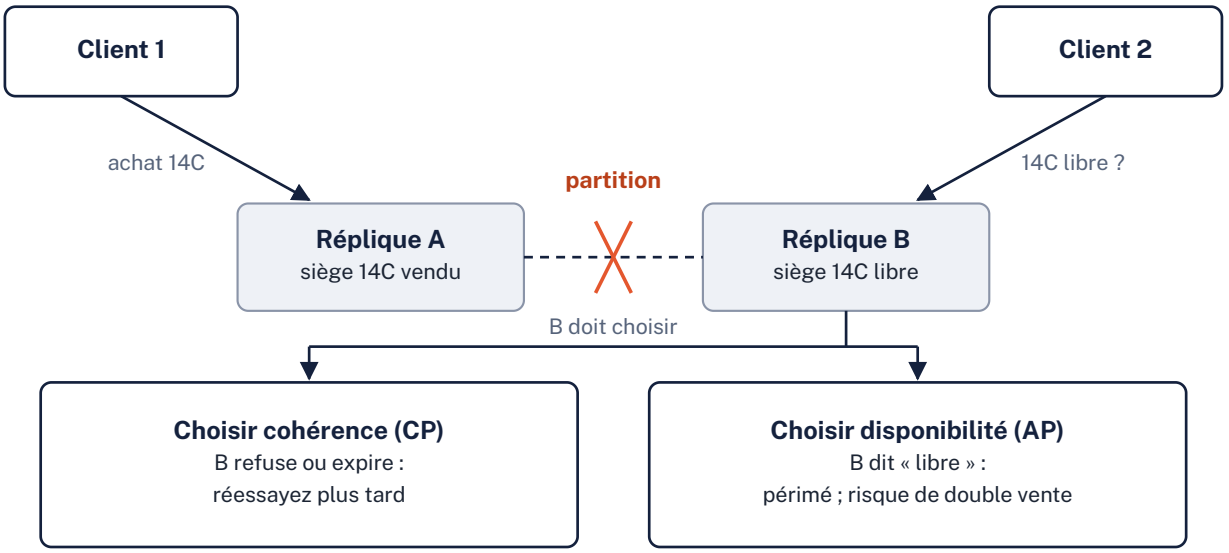
Eric Brewer a présenté l'idée de CAP sous forme de conjecture lors de sa conférence plénière au ACM Symposium on Principles of Distributed Computing (PODC) en 2000. En 2002, Seth Gilbert et Nancy Lynch en ont publié une démonstration formelle, ce qui en a fait un théorème pour leurs définitions précises. La **cohérence** (consistency) désignait la linéarisabilité. La **disponibilité** (availability) signifiait que chaque requête reçue par un nœud qui n'est pas tombé

en panne doit donner lieu à une réponse. La **tolérance aux partitions** (partition tolerance) signifiait que le système tient ses promesses même quand le réseau perd un nombre arbitraire de messages entre les nœuds.

Lu attentivement, le théorème CAP a une portée étroite. Il dit que, lorsqu'une partition réseau se produit, de sorte que certains nœuds ne peuvent pas en joindre d'autres, un système répliqué doit choisir, pour chaque requête, entre cohérence et disponibilité. Un nœud coupé des écritures les plus récentes peut soit refuser de répondre (ou attendre que la partition soit résorbée), ce qui préserve la cohérence, soit répondre avec ce qu'il a, ce qui préserve la disponibilité au risque de renvoyer des données périmées ou d'accepter des écritures conflictuelles. Le diagramme montre ce dilemme pour un système de billetterie qui vend le siège 14C.

Plusieurs lectures répandues sont fausses. **« Choisissez-en deux sur trois »** suggère que la tolérance aux partitions est facultative, mais dans tout système qui communique à travers un réseau, les partitions se produisent, que vous les choisissiez ou non ; le seul vrai choix est la façon de se comporter quand elles surviennent. Un système dit « CA » est un système qui n'a pas décidé quoi faire lorsqu'une partition survient ; ce n'est défendable que là où les partitions sont pratiquement impossibles, comme sur un nœud unique. **« Notre base de données est CP (ou AP) »** est généralement imprécis : la disponibilité au sens de CAP est une propriété très forte que beaucoup de systèmes présentés comme hautement disponibles n'ont pas, et beaucoup de systèmes dits CP ne sont pas non plus linéarisables. Martin Kleppmann a soutenu en 2015 qu'il ne faudrait pas du tout qualifier les bases de données de CP ou d'AP, car la plupart ne correspondent à aucune des deux définitions et leur comportement dépend de leur configuration. **« CAP explique notre latence »** est faux également : CAP ne dit rien de ce qui se passe en l'absence de partition, c'est-à-dire la plupart du temps.

Le choix CAP, de façon concrète



La réplique B ne peut pas apprendre que le siège 14C vient d'être vendu sur la réplique A : elle doit donc soit refuser de répondre, soit risquer de vendre le siège deux fois.

CAP, douze ans après

En 2012, dans un article d'IEEE Computer intitulé « CAP Twelve Years Later: How the Rules Have Changed », Brewer lui-même a expliqué que la formule « deux sur trois » avait induit les gens en erreur. Les partitions sont rares : un système peut donc offrir à la fois la cohérence et la disponibilité presque tout le temps ; le choix ne s'applique que pendant une partition, et seulement aux opérations concernées. Le choix n'a pas non plus à être global : des opérations différentes et des données différentes peuvent faire des choix différents.

Brewer recommandait de concevoir explicitement pour les partitions en trois étapes : **détecter** la partition, entrer dans un **mode partition** explicite dans lequel certaines opérations sont limitées (par exemple, accepter les commandes mais reporter les remboursements), et lancer ensuite un processus de **rétablissement** qui fusionne l'état et compense les erreurs commises pendant la partition. Il notait aussi que le C de CAP est la cohérence à copie unique, un sous-ensemble plus strict de ce que protège le C d'ACID, et que des invariants comme « ne jamais survendre » peuvent devoir être réparés après coup. Ce cadre mène directement aux compensations au niveau métier des sagas : une compagnie aérienne qui survend un vol dédommage le passager plutôt que de refuser toutes les réservations pendant une panne réseau.

Horloges vectorielles et horloges logiques hybrides

Fusionner après une partition exige de savoir quelles écritures sont réellement en conflit, et les horloges murales ne peuvent pas le dire (section 6.1). Une **horloge vectorielle** attache à chaque version une liste de paires nœud-compteur. Si chaque compteur de l'une est inférieur ou égal à ceux d'une autre, la première version est un ancêtre et peut disparaître ; sinon, les deux sont concurrentes et doivent être fusionnées. L'article Dynamo d'Amazon de 2007 s'en servait pour les paniers d'achat : aucun ajout n'est perdu, même si des articles supprimés peuvent réapparaître. Les listes grandissent, et Dynamo les tronquait donc. Une **horloge logique hybride** (Kulkarni et d'autres, 2014) est un unique horodatage compact : un temps proche de l'horloge physique plus un compteur, de sorte qu'un événement qui en a entraîné un autre a toujours la plus petite valeur. Elle ordonne les événements, mais, contrairement à une horloge vectorielle, une valeur plus petite ne prouve pas un lien de causalité. CockroachDB utilise des horloges logiques hybrides.

PACELC : le compromis que vous payez chaque jour

Daniel Abadi a souligné, dans un billet de blog de 2010 puis dans un article d'IEEE Computer de 2012, « Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story », que CAP ignore le compromis auquel les concepteurs sont réellement confrontés la plupart du temps. PACELC se lit ainsi : s'il y a une **P**artition, choisir entre **A**vailability (disponibilité) et **C**onsistency (cohérence) ; **E**lse (sinon), en fonctionnement normal, choisir entre **L**atency (latence) et **C**onsistency (cohérence).

La moitié « else » (sinon) est l'ajout important. Pour qu'une lecture soit cohérente, un système répliqué doit se coordonner : envoyer l'écriture aux autres répliques et attendre leurs accusés de réception, ou diriger chaque lecture vers un leader, ou contacter plusieurs répliques et comparer. Chacune de ces options coûte des allers-retours réseau, et entre régions, un aller-retour peut coûter des dizaines, voire des centaines de millisecondes. Un système qui répond depuis la réplique la plus proche sans se coordonner est plus rapide, mais ses données peuvent être périmées. Ce compromis se paie à chaque requête, qu'il y ait partition ou non.

Des classifications PACELC sont souvent citées, comme PA/EL pour les systèmes réglés pour la vitesse et la disponibilité, et PC/EC pour les systèmes qui préfèrent toujours la cohérence. Considérez-les comme la description d'une configuration, pas d'un produit : la même base de données peut être PA/EL dans un déploiement et PC/EC dans un autre, selon des paramètres choisis par table, par requête ou par appel.

Consensus et coordination : leaders, baux et verrous

Le basculement, les cartes de shards et les tâches planifiées exigent tous que plusieurs machines s'accordent sur une même réponse, par exemple quel nœud est désormais leader, même si certaines sont tombées en panne ou si des messages sont en retard. C'est le problème du **consensus**, et une décision une fois prise est définitive. Paxos, de Leslie Lamport, est l'algorithme classique. Raft (Ongaro et Ousterhout, 2014) offre la même tolérance aux pannes et les mêmes performances, mais a été conçu pour être plus facile à comprendre. Un leader Raft envoie des heartbeats ; un suiveur qui n'en reçoit aucun avant l'expiration de son délai aléatoire lance une élection dans un nouveau *mandat* numéroté, et une majorité de votes en fait le leader. Cinq serveurs continuent de fonctionner avec deux en panne. Au-delà, le cluster cesse de progresser, mais il ne renvoie jamais de résultat erroné. Les clusters ont généralement une taille impaire, car la documentation d'etcd note que quatre membres ne tolèrent pas plus de pannes que trois, et elle conseille de ne pas dépasser sept, puisque chaque écriture doit atteindre davantage de machines.

Peu d'équipes l'implémentent elles-mêmes. Elles utilisent **etcd**, qui repose sur Raft et sert de stockage à toutes les données de cluster de Kubernetes, ou **Apache ZooKeeper**, dont la recette d'élection désigne comme leader le candidat qui détient le nœud temporaire de plus petit numéro. Tous deux contiennent de petites quantités de données de coordination, pas des enregistrements métier.

Un **bail** (lease) est une revendication limitée dans le temps : le détenteur doit le renouveler sans cesse, et s'il meurt, le bail expire et un autre nœud prend le relais. Kubernetes utilise des baux pour les heartbeats des nœuds et l'élection de leader. Le danger est un détenteur qui se met en pause, par exemple pour le ramasse-miettes, puis se réveille sans savoir que son bail a expiré et écrit quand même ; vérifier d'abord n'y change rien, car la pause peut survenir entre la vérification et l'écriture. Un **jeton de fencing** (fencing token) résout ce problème : un numéro qui augmente à chaque attribution et accompagne chaque écriture, de sorte que le stockage rejette tout jeton inférieur à un jeton déjà vu.

Les **verrous distribués** soulèvent la même question. Martin Kleppmann distingue les verrous d'efficacité, dont l'échec se traduit par du travail en double, comme un e-mail répété, et les verrous de correction, dont l'échec corrompt les données. L'algorithme Redlock de Redis prend un verrou sur une majorité de maîtres Redis indépendants. Kleppmann a soutenu en 2016 qu'il dépend d'hypothèses de temporisation et n'émet aucun jeton de fencing, et qu'il ne devrait donc pas protéger la correction. Salvatore Sanfilippo, créateur de Redis, a répondu qu'il est sûr dans le cadre de son modèle déclaré, tout en acceptant l'argument sur les horloges monotones. La documentation de Redis renvoie aux deux textes ; le débat n'est pas tranché, alors lisez-les. Les stockages de consensus demandent le même soin : le test d'etcd 3.4.3 par Jepsen a montré que deux clients pouvaient détenir un même verrou, et a conseillé le fencing. La leçon pratique

est que la ressource protégée doit se défendre elle-même, par une vérification de fencing ou une contrainte de base de données. Si Affectation exécute un seul planificateur d'affectations, une double exécution occasionnelle ne doit causer aucun dommage.

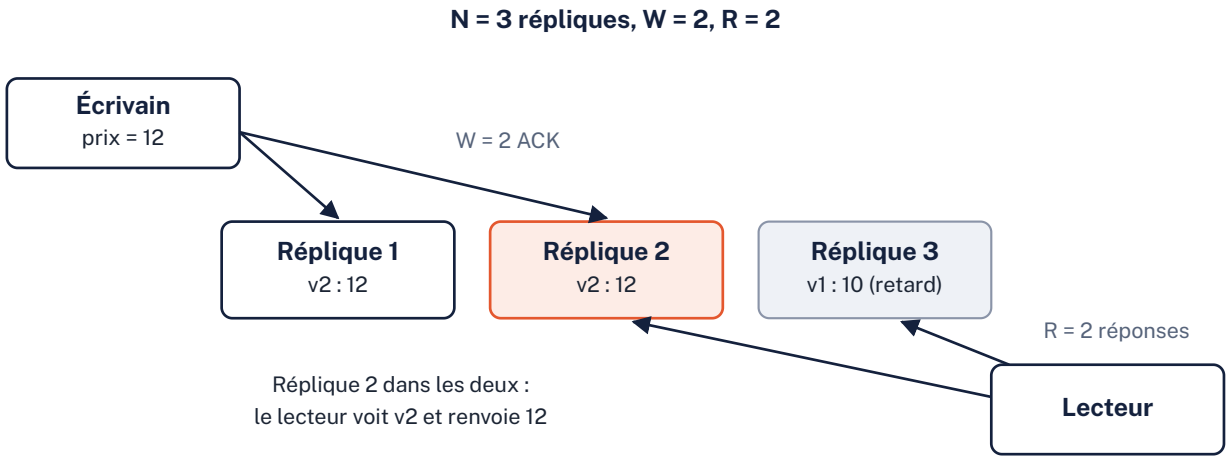
Les quorums : régler la cohérence par l'arithmétique

Les stockages répliqués sans leader, dans le style de l'article Dynamo d'Amazon (2007), permettent à chaque opération de choisir combien de répliques impliquer. Avec *N* répliques d'un élément, une écriture est confirmée quand *W* répliques l'ont stockée, et une lecture recueille les réponses de *R* répliques et renvoie la version la plus récente parmi elles. Si R + W > N, chaque ensemble de lecture recoupe chaque ensemble d'écriture sur au moins une réplique : une lecture inclura donc au moins une copie de la dernière écriture confirmée.

Le choix de *R* et de *W* arbitre entre cohérence, latence et disponibilité. Des valeurs plus basses signifient attendre moins de répliques, et les plus rapides, et tolérer davantage de pannes, au prix de lectures éventuellement périmées. Des valeurs plus élevées signifient l'inverse. Les écritures sont généralement envoyées de toute façon aux *N* répliques ; *W* décide seulement combien d'accusés de réception le client attend.

Le recouvrement des quorums est nécessaire pour des lectures à jour, mais il ne constitue pas, à lui seul, la linéarisabilité. Des écritures concurrentes peuvent être ordonnées par des horodatages issus d'horloges en désaccord, si bien qu'une écriture plus tardive peut perdre avec last-write-wins. Une écriture qui a échoué après avoir atteint certaines répliques peut rester visible sur celles-ci. Les quorums « sloppy » (approximatifs), qui acceptent des écritures sur des nœuds de remplacement pendant les pannes, brisent le recouvrement. Et deux lectures qui recoupent une écriture encore en cours peuvent voir la nouvelle valeur puis l'ancienne, à moins que le lecteur ne répare les répliques en retard avant de répondre. Les stockages qui ont besoin d'une véritable sémantique compare-and-set ajoutent un protocole de consensus, comme les lightweight transactions d'Apache Cassandra, fondées sur Paxos, pour ces seules opérations.

Pourquoi R + W > N fonctionne



L'écriture a été confirmée par les répliques 1 et 2 ; la lecture interroge les répliques 2 et 3 ; le recouvrement sur la réplique 2 signifie que la version la plus récente se trouve dans l'ensemble de lecture.

EXEMPLE COMMENTÉ

Exemple commenté : choisir N, W et R pour le stockage des prix de la carte

La plateforme de livraison de repas stocke les prix de la carte dans un stockage sans leader avec N = 3 répliques, une par zone de disponibilité. Chaque réplique répond en 2 ms environ dans sa zone et en 5 ms environ depuis une autre zone. Le client de quorum attend la W-ième (ou R-ième) réponse la plus rapide.

W = 2, R = 2 (R + W = 4 > 3)	Recouvrement garanti. Les lectures et les écritures attendent chacune la deuxième réplique la plus rapide (environ 5 ms). Toutes deux réussissent encore si une zone est indisponible. Un réglage par défaut équilibré et courant.
W = 3, R = 1 (4 > 3)	Recouvrement garanti et lectures rapides (réplique la plus proche, environ 2 ms), mais chaque écriture échoue si une seule réplique est indisponible : la disponibilité en écriture est le produit de trois disponibilités. Convient aux données rarement écrites et constamment lues, si les écritures échouées peuvent être rejouées plus tard.
W = 1, R = 3 (4 > 3)	Les écritures sont rapides et survivent à deux pannes ; chaque lecture doit atteindre les trois répliques, si bien que les lectures échouent si une zone est injoignable. C'est rarement un bon compromis.
W = 1, R = 1 (2, pas > 3)	Le plus rapide et le plus disponible, mais une lecture peut manquer entièrement la dernière écriture. Acceptable pour un compteur de popularité ; pas pour un prix qui sera facturé au client.
N = 5, W = 3, R = 3 (6 > 5)	Recouvrement garanti tout en tolérant deux répliques en panne, pour les lectures comme pour les écritures, au prix de davantage de stockage et de quorums légèrement plus lents.

Règle générale : avec R + W > N, les écritures tolèrent N − W pannes et les lectures tolèrent N − R pannes.

Le recouvrement ne donne des lectures à jour que pour les écritures confirmées, et seulement avec un versionnage fiable ; voyez les réserves ci-dessus avant de vous fier aux quorums pour des invariants comme les niveaux de stock.

Pour les prix, la plateforme a choisi W = 2, R = 2, et a facturé les clients au prix enregistré dans la commande au moment du paiement, si bien qu'une lecture périmée de la carte ne pouvait jamais modifier le montant facturé.

Comment configurer des bases de données bien connues

Les produits réels se situent rarement en un point unique de l'échelle. Ils exposent des paramètres, et la bonne description est « comment ce déploiement est configuré ». Quatre exemples courants suivent ; consultez la documentation actuelle de la version que vous utilisez, car les valeurs par défaut changent d'une version à l'autre.

La cohérence configurable dans quatre bases de données très utilisées (comportement tel que documenté au moment de la rédaction)

Base	Comportement par défaut typique	Paramètres qui renforcent la cohérence	Points de vigilance
PostgreSQL avec réplication en continu (streaming replication)	Un serveur primaire reçoit toutes les écritures ; la réplication vers les serveurs de secours (standbys) est asynchrone par défaut, si bien que les lectures sur un standby peuvent être périmées.	`synchronous_standby_names` avec `synchronous_commit` (par exemple `remote_apply`) fait attendre une validation jusqu'à ce que les standbys l'aient reçue ; lisez sur le primaire pour obtenir read-your-writes.	La réplication synchrone ajoute de la latence à la validation, et les écritures se bloquent si les standbys requis sont injoignables.
Replica set MongoDB	Les écritures exigent par défaut un accusé de réception de la majorité dans la plupart des déploiements ; les lectures sur le primaire utilisent par défaut le read concern `local`, qui peut renvoyer des données annulées plus tard par un rollback.	Read concern `majority` ou `linearizable` ; sessions à cohérence causale quand les lectures et les écritures utilisent toutes deux majority.	Lire sur les secondaires (read preference) échange la fraîcheur des données contre la latence et la charge.
Apache Cassandra	Le niveau de cohérence est choisi à chaque requête (ONE, QUORUM, LOCAL_QUORUM, ALL et d'autres) ; les conflits sont résolus par horodatages en last-write-wins.	QUORUM (ou LOCAL_QUORUM au sein d'un centre de données) pour les lectures comme pour les écritures, afin que R + W > facteur de réplication ; lightweight transactions pour le compare-and-set.	Le décalage d'horloge affecte last-write-wins ; LOCAL_QUORUM ne donne aucune garantie entre centres de données.
Amazon DynamoDB	Les lectures sont à cohérence à terme par défaut ; les tables globales se répliquent entre régions de façon asynchrone par défaut.	`ConsistentRead=true` sur les tables et les index secondaires locaux ; tables globales avec cohérence forte multirégion (disponible en version générale depuis 2025).	Les lectures fortement cohérentes ne sont pas disponibles sur les index secondaires globaux, consomment deux fois plus de capacité de lecture et peuvent être plus lentes.

Choisir la cohérence selon le cas d’usage

L’habitude la plus utile est de cesser de demander « notre système est-il cohérent ? » et de demander, pour chaque donnée et chaque opération : *que se passe-t-il de travers si un client voit une valeur périmée ou réordonnée, et combien cela coûte-t-il ?* Les mouvements d’argent, les ressources uniques et les invariants justifient généralement une cohérence forte et sa latence. Les compteurs, les fils d’actualité, les recommandations et l’analytique, généralement pas. Beaucoup d’anomalies visibles par les utilisateurs disparaissent avec les seules garanties de session, qui sont peu coûteuses : dirigez les lectures d’un utilisateur vers le primaire pendant quelques secondes après que cet utilisateur a écrit, ou transportez un jeton de version dans la session et attendez que la réplique l’ait rattrapé.

Besoins de cohérence selon le cas d'usage

Cas d'usage	Coût d'une lecture périmée ou réordonnée	Garantie raisonnable	Mécanisme typique
Solde bancaire avant un retrait	Découvert au-delà de la limite ; argent perdu	Linéarisable (et transactions sérialisables)	Leader unique pour le compte, réplication synchrone, ou consensus
Dernière place ou dernier billet à l'ouverture d'une billetterie	Une même place vendue deux fois ; remboursements et clients mécontents	Linéarisable par place	Écriture conditionnelle (compare-and-set) sur l'enregistrement de la place
Nom d'utilisateur unique à l'inscription	Deux comptes portant le même nom	Linéarisable sur la vérification d'unicité	Contrainte d'unicité dans une seule base de données, ou registre adossé à un consensus
Adresse de livraison du client lui-même après modification	Repas envoyé à l'ancienne adresse	Lecture de ses propres écritures (read-your-writes)	Lecture sur le leader après une écriture, ou jeton de version de session
État de la commande affiché au client	L'état semble revenir en arrière ; confusion et appels au support	Lectures monotones plus read-your-writes	Routage sticky vers une seule réplique, ou jetons de version
Messagerie entre le restaurant et le client	Une réponse apparaît avant sa question	Causale	Stockage à cohérence causale, ou ordonnancement par conversation
Panier	Un article retiré réapparaît ; légèrement agaçant	À terme, avec fusion	Toujours accepter les ajouts ; fusionner les versions (l'exemple de l'article Dynamo)
Nombre de likes ou note d'un restaurant	Le nombre est faux de quelques unités pendant quelques secondes	À terme	Réplication asynchrone, compteurs ou CRDT
Position du livreur sur la carte	Point en retard de quelques secondes	À terme, la dernière valeur l'emporte	Écritures à faible latence sur la réplique la plus proche ; anciennes positions écartées

ATTENTION

Erreurs courantes

- Dire « nous avons choisi AP » pour tout un système, au lieu de décider opération par opération et donnée par donnée.
- Supposer qu'une base de données répliquée est linéarisable parce que son marketing dit « fortement cohérente », sans lire ce que couvre la garantie (clés uniques ? une seule région ? index ?).
- Ajouter des répliques en lecture ou des caches à un système en oubliant que chaque chemin de lecture visible par l'utilisateur vient de devenir cohérent à terme.
- Compter sur last-write-wins avec des horloges non synchronisées, et perdre silencieusement des mises à jour.
- Utiliser des quorums avec R + W > N en croyant que cela sécurise les réservations de stock sans compare-and-set.
- Traiter la cohérence à terme comme un détail technique, alors qu'elle est visible par les utilisateurs et exige des décisions produit sur ce qu'ils voient pendant que les données convergent.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Quelle est la différence entre la linéarisabilité et la cohérence séquentielle ?**

Toutes deux exigent un ordre unique des opérations sur lequel tous les clients s'accordent et qui respecte l'ordre propre à chaque client ; la linéarisabilité exige en outre que cet ordre respecte le temps réel, si bien qu'une lecture qui commence après la fin d'une écriture doit la voir.

▼ **Pourquoi une base de données distribuée « CA » n'est-elle pas un choix qui a du sens ?**

Parce que les partitions réseau se produisent dans tout système qui communique à travers un réseau ; un système qui suppose qu'elles ne surviennent jamais n'a tout simplement pas décidé de ce qu'il fera quand elles surviendront.

▼ **Avec N = 5 et W = 2, quel est le plus petit R qui garantit que les ensembles de lecture et d'écriture se recoupent ?**

R = 4, car R + W doit dépasser 5.

▼ **Un utilisateur modifie son profil et la page se recharge en affichant l'ancienne version, alors qu'il importe peu que les autres utilisateurs la voient en retard. Quelle garantie manque-t-il, et quelle est une solution peu coûteuse ?**

Read-your-writes (lecture de ses propres écritures) ; dirigez les lectures de cet utilisateur vers le leader pendant un court moment après une écriture, ou transportez un jeton de version et attendez une réplique qui a rattrapé son retard.

▼ **À vous : Un stockage de prix de places utilise N = 5, W = 2, R = 3, et un collègue affirme que les lectures recoupent toujours la dernière écriture. Vérifiez-le et proposez un correctif. Utiliseriez-vous les quorums seuls pour empêcher deux fans d'acheter une même place ?**

R + W = 5, ce qui n'est pas supérieur à N : une lecture peut donc manquer la dernière écriture. Fixez W = 3 et R = 3, ce qui donne 6 > 5 ; les deux fonctionnent encore avec deux répliques en panne, avec une latence légèrement plus élevée. Choisissez selon l'usage : des prix périmés peuvent être tolérables brièvement. Les quorums seuls ne sont pas linéarisables : ils ne protégeront donc pas une place unique. Vendre une seule place exige une cohérence forte, comme une transaction à leader unique ou une contrainte d'unicité.

SOURCES

- Eric Brewer: CAP Twelve Years Later: How the Rules Have Changed (InfoQ reprint of the IEEE Computer article): https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/
- Daniel Abadi: Problems with CAP, and Yahoo's little known NoSQL system (2010, origin of PACELC): https://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html
- Jepsen: Consistency models: https://jepsen.io/consistency/models
- Martin Kleppmann: Please stop calling databases CP or AP (2015): https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html
- Apache Cassandra documentation: Dynamo (tunable consistency and quorums): https://cassandra.apache.org/doc/latest/cassandra/architecture/dynamo.html
- Raft consensus algorithm site: https://raft.github.io/
- Ongaro and Ousterhout: In Search of an Understandable Consensus Algorithm (2014): https://raft.github.io/raft.pdf
- etcd FAQ: https://etcd.io/docs/v3.5/faq/
- Kubernetes: Operating etcd clusters: https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/
- Kubernetes: Leases: https://kubernetes.io/docs/concepts/architecture/leases/

- Apache ZooKeeper: Recipes and solutions: https://zookeeper.apache.org/doc/current/recipes.html
- Martin Kleppmann: How to do distributed locking (2016): https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html
- Salvatore Sanfilippo: Is Redlock safe?: https://antirez.com/news/101
- Redis: Distributed locks: https://redis.io/docs/latest/develop/clients/patterns/distributed-locks/
- Jepsen: etcd 3.4.3: https://jepsen.io/analyses/etcd-3.4.3
- DeCandia and others: Dynamo: Amazon's Highly Available Key-value Store (2007): https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- Kulkarni and others: Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases: https://cse.buffalo.edu/tech-reports/2014-04.pdf
- CockroachDB: Transaction layer: https://docs.cockroachlabs.com/docs/stable/architecture/transaction-layer

Transactions distribuées, sagas et patron outbox

AVANT DE LIRE

Cette section s'appuie sur :

- 0.5 Stocker les données
- 6.3 Microservices
- 6.4 Pilotée par les événements

Termes déjà vus qu'elle utilise : ACID, Base de données par service, Chorégraphie, Orchestration, Livraison au moins une fois (at-least-once)

L'IDÉE EN BREF

Réserver des vacances, c'est réserver un vol, un hôtel et une voiture de location auprès de trois entreprises différentes. Personne ne peut faire en sorte que les trois se produisent exactement au même instant. Vous les réservez donc l'une après l'autre, et si la voiture de location s'avère indisponible, vous annulez l'hôtel et le vol, en payant peut-être des frais d'annulation. C'est une saga : une série d'étapes distinctes, chacune avec une façon prévue de l'annuler. Un logiciel qui s'étend sur plusieurs services fonctionne de la même manière, et tout l'art consiste à choisir l'ordre des étapes et à prévoir les annulations avant que quoi que ce soit ne tourne mal.

CE QU'IL FAUT SAVOIR

- La validation en deux phases assure l'atomicité entre participants au moyen des phases de préparation et de validation, mais elle bloque tant que le coordinateur est en panne et maintient des verrous à travers le réseau ; XA (X/Open, 1991) la normalise.
- Entre services détenus de manière indépendante, le 2PC multiplie l'indisponibilité, maintient des verrous à distance, couple les déploiements et n'est pas pris en charge par la plupart des API ni par de nombreux courtiers : on l'évite donc en général.
- Une saga (Garcia-Molina et Salem, 1987) est une suite de transactions locales, chacune assortie d'une transaction compensatoire ; soit toutes les étapes aboutissent, soit celles qui ont abouti sont compensées dans l'ordre inverse.
- Les compensations sont de nouvelles transactions qui vont de l'avant, pas des retours arrière : des états intermédiaires ont pu être vus, et les compensations doivent être idempotentes, rejouables et capables d'arriver dans le désordre.

- Les sagas manquent d'isolation ; parmi les contre-mesures décrites par Chris Richardson figurent les verrous sémantiques, les mises à jour commutatives, la vue pessimiste, la relecture de la valeur, les fichiers de versions et le choix par la valeur.
- L'outbox transactionnelle écrit la modification métier et son message dans une même transaction locale, et un relais (par interrogation périodique ou par capture des données modifiées, comme Debezium) le publie au moins une fois ; des récepteurs idempotents dotés d'une inbox éliminent les doublons.
- Préférez une transaction locale unique quand c'est possible, revoyez les frontières quand le découpage est accidentel, et utilisez des sagas quand les étapes peuvent être compensées ou réessayées.

Le problème : une action métier, plusieurs bases de données

Dans une seule base de données, une transaction rend atomiques plusieurs modifications : la ligne de la commande, ses articles et la référence du paiement sont tous enregistrés, ou aucun ne l'est (section 0.5). Dès que les données sont réparties entre des services qui possèdent chacun leur base de données, comme le recommandent les microservices et la base de données par service, une seule action métier, comme passer une commande de repas, touche plusieurs bases de données appartenant à des services différents : celle du service de commandes, celle du service de paiement et celle du service d'affectation. Aucune transaction locale ne peut les couvrir toutes.

Il existe deux familles de réponses. La première consiste à faire agir de nombreux gestionnaires de ressources comme un seul, au moyen d'un protocole de validation atomique distribuée, dont la validation en deux phases est l'exemple classique. La seconde consiste à accepter que l'action soit une suite de transactions locales et à concevoir en fonction des moments intermédiaires : c'est la saga. Une troisième réponse, souvent la meilleure, consiste à remarquer que la frontière de la transaction et celle du service ne coïncident pas, et à déplacer l'une des deux. Cette section couvre les trois et se termine par un guide pour choisir.

La validation en deux phases et XA

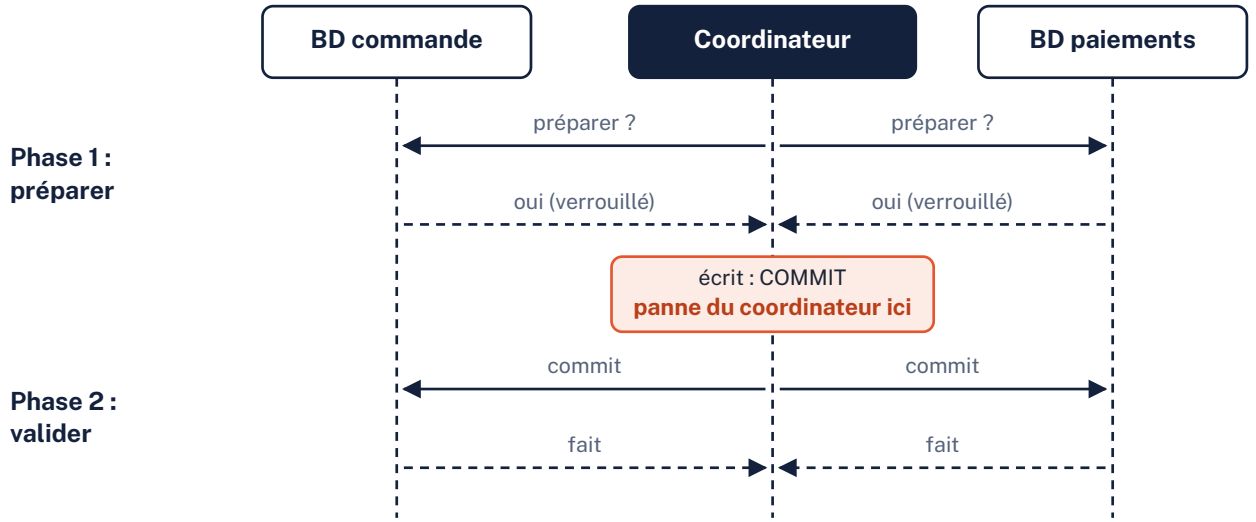
La validation en deux phases (2PC) est un protocole, qui remonte à la fin des années 1970, assurant une atomicité tout-ou-rien entre plusieurs participants, par exemple deux bases de données, ou une base de données et un courtier de messages (message broker). Un **coordinateur** (le gestionnaire de transactions) le pilote. En **phase 1, préparation**, le coordinateur demande à chaque participant s'il peut valider. Un participant qui répond oui doit d'abord rendre les modifications durables et conserver les verrous concernés, car en votant oui il renonce au droit d'annuler de lui-même. Si tous les votes sont oui, le coordinateur écrit sa

décision, COMMIT, dans son propre journal ; si un vote est non ou qu'un participant ne répond pas, la décision est ABORT. En **phase 2, validation**, il communique le résultat à chaque participant, qui l'applique et libère ses verrous.

La faiblesse est que le 2PC est un protocole **bloquant**. Un participant qui a voté oui est *en doute* : il ne connaît pas le résultat et ne peut pas le deviner sans risque, car un autre participant a peut-être déjà validé. Si le coordinateur tombe en panne à ce moment-là et le reste, les participants en doute conservent leurs verrous jusqu'à son rétablissement, et les autres transactions qui touchent ces lignes attendent elles aussi. Les opérateurs peuvent forcer une décision *heuristique* pour libérer les verrous, au risque de rompre l'atomicité.

La spécification XA, publiée par X/Open en 1991, a normalisé l'interface entre un gestionnaire de transactions et des gestionnaires de ressources, de sorte que, par exemple, un serveur d'applications Java puisse valider atomiquement une mise à jour de base de données et un envoi de message. De nombreuses bases de données relationnelles et certains courtiers de messages la prennent en charge.

La validation en deux phases, et là où elle bloque



Si le coordinateur tombe après les votes, ceux qui ont voté oui ne peuvent décider seuls et gardent leurs verrous

Les participants qui ont voté oui conservent leurs verrous jusqu'à ce qu'ils apprennent la décision ; si le coordinateur tombe en panne au point mis en évidence, ils restent bloqués.

Pourquoi on évite le 2PC entre microservices

Le 2PC fonctionne et reste utilisé à l'intérieur des frontières d'un même système. Entre services détenus de manière indépendante, il convient généralement mal, pour des raisons qui se renforcent mutuellement.

La disponibilité se multiplie à la baisse. Chaque participant et le coordinateur doivent être disponibles pour qu'une transaction soit validée : la disponibilité de la transaction est donc le produit des leurs, le même calcul que pour une chaîne d'appels synchrones (voir pourquoi distribuer). **Les verrous sont maintenus à travers le réseau** pendant au moins deux allers-retours, et bien plus longtemps si quelque chose est lent, ce qui réduit le débit et accroît la contention. **Le couplage revient** : tous les participants doivent parler le même protocole, partager un gestionnaire de transactions et s'accorder sur les délais d'expiration, ce qui sape la déployabilité indépendante qui justifiait de séparer les services. **De nombreux participants ne peuvent pas y prendre part** : la plupart des API HTTP, les services tiers comme les prestataires de paiement, de nombreuses bases de données NoSQL et plusieurs courtiers populaires ne prennent pas en charge XA. Et l'API du prestataire de paiement ne maintiendra jamais un verrou pour votre coordinateur.

Le tableau est différent à l'intérieur des bases de données distribuées. Des systèmes comme Google Spanner exécutent la validation en deux phases en interne entre des groupes de répliques eux-mêmes maintenus cohérents par consensus, si bien qu'un coordinateur tombé en panne est remplacé plutôt qu'attendu. Cela rend praticables les transactions entre partitions au sein d'une même base de données ; cela ne rend pas praticables les transactions entre des services détenus séparément, qui reposent sur des technologies différentes.

Les sagas : une suite de transactions locales

Hector Garcia-Molina et Kenneth Salem ont introduit la saga en 1987, dans un article présenté à SIGMOD sur les *transactions de longue durée*, celles qui maintiendraient des verrous de base de données pendant des heures. Leur idée était de découper une telle transaction en une suite de transactions ordinaires T_1, T_2 … T_n, dont chacune est validée séparément, et d'associer à chacune une transaction compensatoire C_i qui l'annule sur le plan sémantique. Le système garantit alors soit que la suite entière aboutit, soit que les étapes ayant abouti sont suivies de leurs compensations dans l'ordre inverse : T_1 … T_j, C_j … C_1.

Les architectures de microservices ont adopté cette idée parce qu'elle correspond exactement à leurs contraintes : chaque étape est une transaction ACID locale dans la base de données propre à un service, et les étapes sont reliées par des messages. Une saga offre l'atomicité au sens métier et à terme (toutes les étapes ont lieu, ou leurs effets sont annulés), mais pas l'isolation : entre deux étapes, d'autres transactions peuvent voir l'état intermédiaire.

Il est utile de classer les étapes, comme le fait Chris Richardson dans *Microservices Patterns* (2018). Les **transactions compensables** peuvent être annulées par une compensation. La **transaction pivot** est le point de non-retour : une fois qu'elle est validée, la saga ira jusqu'au

bout. Les **transactions rejouables** viennent après le pivot et doivent être conçues pour finir par réussir si on les réessaie. Placer avant le pivot les étapes susceptibles d'échouer, et après lui celles qui ne peuvent pas être annulées, est une décision de conception essentielle.

Chorégraphie ou orchestration

Les étapes d'une saga doivent être coordonnées d'une manière ou d'une autre, et les deux options reprennent la chorégraphie et l'orchestration présentées à propos de l'architecture pilotée par les événements.

Dans une **saga chorégraphiée**, il n'y a pas de coordinateur. Chaque service écoute des événements et y réagit en effectuant sa transaction locale et en publiant un nouvel événement : le service de commandes publie `OrderCreated`, le service de paiement réagit et publie `PaymentAuthorised`, le service d'affectation réagit à son tour, et ainsi de suite ; des événements d'échec déclenchent des compensations dans les services qui les écoutent. C'est simple pour deux ou trois étapes, et cela garde les services faiblement couplés. À mesure que les étapes se multiplient, le flux n'existe plus que de façon implicite, éparpillé entre les services, si bien qu'il devient difficile à voir, à tester et à modifier ; et des dépendances cycliques entre services, par le jeu des événements, s'installent insidieusement.

Dans une saga orchestrée, un orchestrateur unique, généralement situé dans le service responsable du processus métier, envoie des commandes aux participants et réagit à leurs réponses. L'orchestrateur est une machine à états persistante : il enregistre dans sa propre base de données l'étape où il en est, ce qui lui permet de reprendre après une panne. Le flux est explicite et testable en un seul endroit, et les participants n'ont pas besoin de se connaître. Le risque est que l'orchestrateur accumule une logique métier qui relève des participants. Des moteurs de workflow comme Temporal, Camunda et AWS Step Functions fournissent une orchestration durable sous forme de plateforme, au prix d'une dépendance supplémentaire. Au-delà de trois étapes environ, ou pour tout flux comportant des branchements et des délais d'expiration, la plupart des praticiens préfèrent l'orchestration.

L'exécution durable : Temporal et Step Functions

L'**exécution durable** signifie qu'un workflow, une fois lancé, s'exécute jusqu'au bout malgré les pannes. Dans Temporal, un workflow est du code ordinaire ; ce qui se produit est stocké dans un historique d'événements, et après une panne, un autre worker rejoue le code sur cet historique pour reconstruire son état et continuer. Le code du workflow doit donc être déterministe, et le travail extérieur va dans des *activités*, qui peuvent faire l'objet de nouvelles tentatives et doivent donc tolérer la répétition. AWS Step Functions définit plutôt une machine à états en Amazon States Language ; les workflows Standard peuvent durer jusqu'à un an et les workflows Express cinq minutes. L'attente, par une saga de commande, de l'acceptation d'un

restaurant peut alors être une simple temporisation. Les coûts sont une nouvelle dépendance critique et la prudence nécessaire pour changer le code de workflows encore en cours. Vérifiez les limites actuelles.

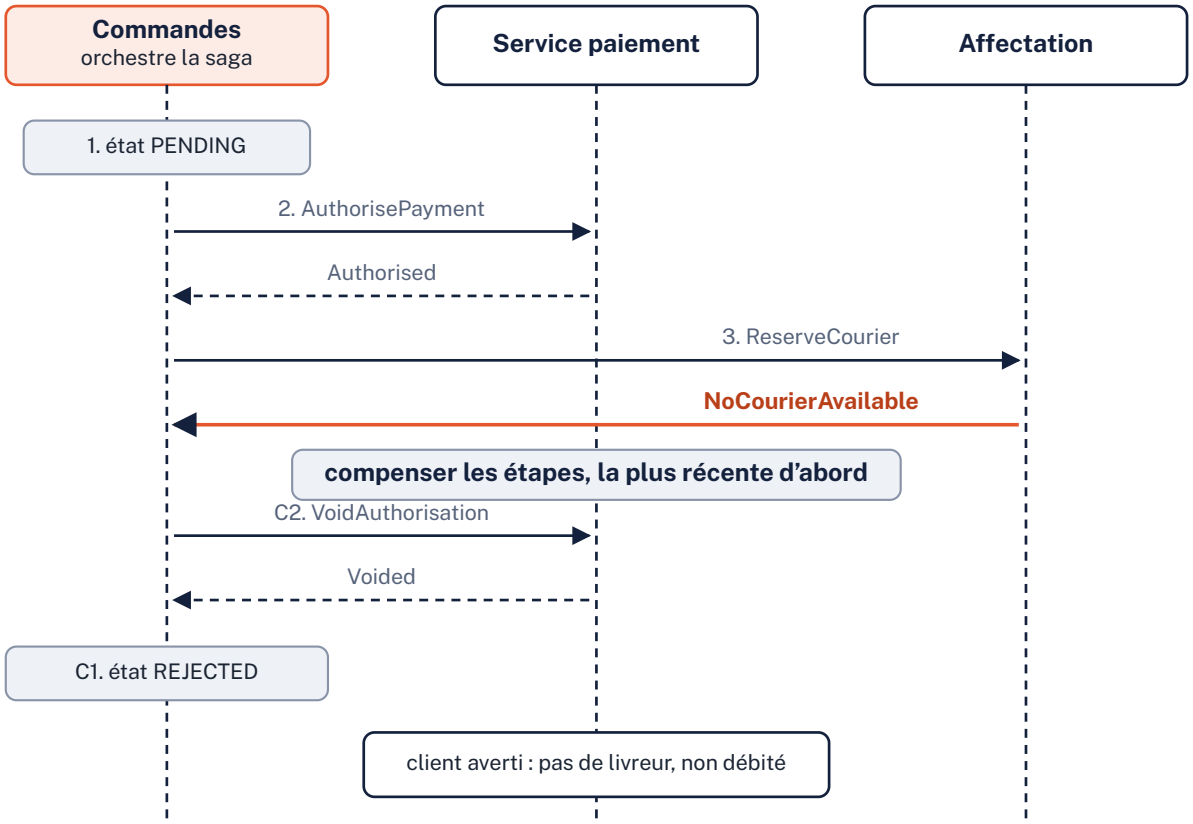
Exemple commenté : la saga de commande

La saga de commande de la plateforme de livraison de repas est orchestrée par le service de commandes. L'ordre des étapes est délibéré : l'autorisation de paiement est un blocage de fonds peu coûteux et facile à annuler, la réservation du livreur est l'étape la plus susceptible d'échouer aux heures de pointe, et la confirmation au restaurant est le pivot, car une fois qu'une cuisine a commencé à préparer la commande, celle-ci ne peut plus simplement être annulée.

La saga de commande, étape par étape

Étape	Service et transaction locale	Type	Compensation si une étape ultérieure échoue
1. Créer la commande	Le service de commandes insère la commande avec l'état `PENDING` (un verrou sémantique).	Compensable	C1 : passer l'état à `REJECTED` et enregistrer le motif. La ligne est conservée pour l'audit, et non supprimée.
2. Autoriser le paiement	Le service de paiement demande au prestataire de carte de bloquer le montant total, en utilisant l'ID de la commande comme clé d'idempotence.	Compensable	C2 : annuler l'autorisation, ce qui libère le montant bloqué. Le client peut brièvement voir un montant en attente dans son application bancaire.
3. Réserver un livreur	Le service d'affectation réserve un livreur proche pour le créneau de retrait.	Compensable	C3 : libérer le livreur pour qu'il puisse accepter d'autres courses.
4. Confirmer la commande	Le service de commandes passe l'état à `CONFIRMED` et envoie le bon de commande au restaurant.	Pivot	Aucune : à partir d'ici, la saga doit aller de l'avant. Les problèmes ultérieurs sont traités par de nouveaux processus métier (remboursements, bons d'achat).
5. Avertir le client ; encaisser le paiement au retrait	Le service de notification envoie « Commande confirmée » ; le service de paiement encaisse le montant bloqué quand le livreur récupère le repas.	Rejouable	Aucune : ces étapes sont réessayées, de façon idempotente, jusqu'à ce qu'elles réussissent ou soient transmises à une personne.

La saga de commande quand aucun livreur n'est disponible



L'étape 3 échoue : l'orchestrateur exécute donc les compensations des étapes 2 et 1, de la plus récente à la plus ancienne, et le client est prévenu avant que quoi que ce soit ait été préparé ou débité.

La compensation n'est pas un retour arrière

Un retour arrière (rollback) de base de données fait comme si la transaction n'avait jamais eu lieu : personne d'autre n'a jamais vu ses modifications. Une compensation est une nouvelle transaction, tournée vers l'avant, qui contrebalance sur le plan sémantique une transaction antérieure qui *a bel et bien* eu lieu et qui a pu être vue. Dans la saga de commande, la banque du client affiche peut-être déjà un montant bloqué, et un e-mail a peut-être annoncé « paiement autorisé ». La compensation ne peut pas annuler l'envoi de l'e-mail ; elle peut seulement lever le blocage et envoyer un autre message. Certaines actions ne peuvent pas du tout être compensées, comme un repas déjà cuisiné ou une notification déjà lue ; elles ont leur place après le pivot, ou nécessitent une compensation métier comme un bon d'achat ou un remboursement.

Les compensations demandent donc le même soin que les étapes qui vont de l'avant. Elles doivent être **idempotentes**, car le message qui les déclenche peut être livré deux fois. Elles doivent être **rejouables jusqu'à ce qu'elles réussissent**, car une compensation qui échoue définitivement laisse le système dans un état que personne n'a conçu ; si elles ne peuvent vraiment pas réussir, elles doivent déclencher une alerte pour qu'une personne règle le

problème. Elles peuvent devoir être **commutatives avec l'étape qu'elles compensent** : si un `VoidAuthorisation` arrive avant l'`AuthorisePayment` qu'il annule (ce qui est possible après des délais d'expiration et des nouvelles tentatives), le service de paiement doit mémoriser l'annulation et refuser l'autorisation tardive plutôt qu'échouer. Enfin, elles doivent être visibles pour le métier : « rejetée, pas de livreur » est une information que l'équipe des opérations veut voir sur un tableau de bord.

Vivre sans isolation : anomalies et contre-mesures

Comme chaque étape est validée séparément, les sagas n'ont pas le I d'ACID, et des sagas concurrentes peuvent interférer. Richardson décrit trois anomalies qui en résultent. Une **mise à jour perdue** se produit quand une saga écrase une modification faite par une autre sans l'avoir lue, par exemple quand une saga d'annulation marque une commande comme annulée et que la saga de commande la passe ensuite à confirmée. Une **lecture sale** se produit quand une saga lit des données qu'une autre saga a modifiées mais risque encore de compenser, comme une vérification de plafond de crédit qui compte un blocage sur le point d'être annulé. Une **lecture floue, ou non reproductible**, se produit quand deux étapes d'une même saga lisent les mêmes données et obtiennent des valeurs différentes parce qu'une autre saga les a modifiées entre-temps.

Il recense aussi des contre-mesures, des techniques de conception qui réduisent ces anomalies, en s'appuyant sur des travaux de recherche antérieurs consacrés aux transactions portant sur plusieurs bases de données. Voici les plus utilisées.

Contre-mesures au manque d'isolation des sagas (d'après Richardson)

Contre-mesure	Idée	Exemple de la livraison de repas
Verrou sémantique	Une étape compensable positionne un indicateur au niveau de l'application, généralement un état `*_PENDING`, qui signale aux autres transactions que l'enregistrement est en cours de changement ; elles attendent, échouent ou le traitent de façon particulière.	Tant qu'une commande est `PENDING`, une demande d'annulation est mise en file d'attente ou refusée avec le message « commande en cours de traitement », au lieu d'être appliquée à mi-parcours.
Mises à jour commutatives	Concevoir les mises à jour pour qu'elles donnent le même résultat dans n'importe quel ordre, ce qui supprime les mises à jour perdues.	Le registre des gains du livreur enregistre les crédits et les débits comme des écritures distinctes au lieu d'écraser un solde.
Vue pessimiste	Réordonner les étapes pour réduire au minimum le risque métier d'une lecture sale, en plaçant les mises à jour risquées dans des étapes rejouables après le pivot.	N'augmenter la « capacité disponible » d'un restaurant que dans une étape rejouable, une fois l'annulation certaine, jamais dans une étape susceptible d'être compensée.
Relecture de la valeur	Avant de mettre à jour, relire l'enregistrement et vérifier qu'il n'a pas changé depuis sa lecture (verrouillage optimiste) ; s'il a changé, abandonner et éventuellement relancer la saga.	Avant de confirmer, l'orchestrateur vérifie que la version de la commande est toujours celle qu'il a créée ; si une annulation l'a modifiée, il compense à la place.
Fichier de versions	Enregistrer les opérations appliquées à un enregistrement, afin que les opérations arrivées dans le désordre puissent être réordonnées ou reconnues.	Le service de paiement enregistre qu'une annulation est arrivée pour une autorisation qu'il n'a pas encore vue, et rejette l'autorisation tardive.
Par la valeur	Choisir le mécanisme de coordination de chaque demande selon son risque métier.	Les commandes ordinaires passent par la saga ; une grosse commande de traiteur pour une entreprise pourrait suivre un processus plus strict et plus lent.

SI VOUS DÉBUTEZ

Pas à pas : le moment entre l'enregistrement et l'envoi

L'idée qui demande le plus d'attention ici est le problème de la double écriture et la façon dont l'outbox transactionnelle le résout. Chaque étape d'une saga enregistre une modification dans sa propre base de données et envoie un message pour que l'étape suivante puisse commencer. La difficulté se trouve dans le moment qui sépare les deux.

- Le service de commandes enregistre la commande 1042 dans sa base de données avec l'état PENDING. Il doit ensuite envoyer un message OrderCreated au courtier, pour que le service de paiement puisse commencer.
- Supposons que le serveur plante après l'enregistrement et avant l'envoi. La commande existe, mais le message n'est jamais parti : le service de paiement n'en entend donc jamais parler, et la commande 1042 attend indéfiniment.
- Procéder dans l'autre sens ne vaut pas mieux : si le service envoie d'abord et que l'enregistrement échoue ensuite, le service de paiement commence à traiter une commande qui n'existe pas.
- Écrire ainsi à deux endroits en étapes séparées s'appelle une double écriture : il y a toujours un moment où l'une a eu lieu et l'autre pas.
- L'outbox transactionnelle supprime ce moment. Le service enregistre la commande et le message sortant, sous la forme d'une ligne dans une table outbox, dans une même transaction de base de données, si bien que les deux sont enregistrés ou aucun ne l'est. Un programme séparé, le relais, lit ensuite l'outbox et envoie chaque message au courtier.
- Si le relais plante après avoir envoyé un message mais avant d'avoir noté qu'il l'a fait, il renvoie ce message : chaque destinataire doit donc gérer les doublons, comme l'explique la fin de cette section.
- **La leçon :** sans validation en deux phases, que cette section déconseille, une base de données et un courtier ne peuvent pas être mis à jour comme un tout. Enregistrez plutôt le message avec la modification, envoyez-le ensuite, et acceptez un doublon de temps à autre.

Publier de façon fiable : l'outbox transactionnelle

Chaque étape de saga fait deux choses : elle met à jour sa propre base de données et elle envoie un message (un événement ou une réponse). Les faire séparément crée le problème de la double écriture (dual write). Si le service valide puis plante avant de publier, le reste de la

saga n'entend jamais parler de l'étape. S'il publie d'abord et que la validation échoue ensuite, d'autres services agissent sur quelque chose qui n'a jamais eu lieu. Envelopper la base de données et le courtier dans un 2PC est précisément ce que nous voulions éviter.

L'outbox transactionnelle (transactional outbox) le résout avec une seule transaction locale. Le service écrit la modification métier et le message sortant, sous la forme d'une ligne dans une table `outbox`, dans la même transaction de base de données ; soit les deux sont validés, soit aucun ne l'est. Un **relais** séparé lit ensuite l'outbox et publie chaque message vers le courtier. Deux conceptions de relais sont courantes. Un **éditeur par interrogation** (polling publisher) interroge régulièrement la table outbox pour y trouver les lignes non publiées et les marque comme envoyées : c'est simple, mais cela ajoute de la charge et de la latence. La **lecture du journal des transactions** (transaction log tailing) utilise la capture des données modifiées (CDC) : un outil lit le journal de réplication de la base de données elle-même (le write-ahead log de PostgreSQL ou le binary log de MySQL) et transforme les nouvelles lignes de l'outbox en messages. Debezium, une plateforme de CDC open source généralement exécutée avec Kafka Connect, en est l'exemple le plus connu ; elle comprend un routeur d'événements outbox (outbox event router) qui achemine chaque ligne vers un topic en fonction d'une colonne comme le type d'agrégat.

Le relais peut planter après avoir publié et avant d'avoir enregistré qu'il l'a fait : il publiera donc parfois un message deux fois. L'outbox assure ainsi une livraison au moins une fois, et les consommateurs doivent gérer les doublons. L'ordre de publication est préservé pour chaque agrégat si le relais publie les lignes dans l'ordre de validation et utilise l'ID de l'agrégat comme clé de message.

```
BEGIN;

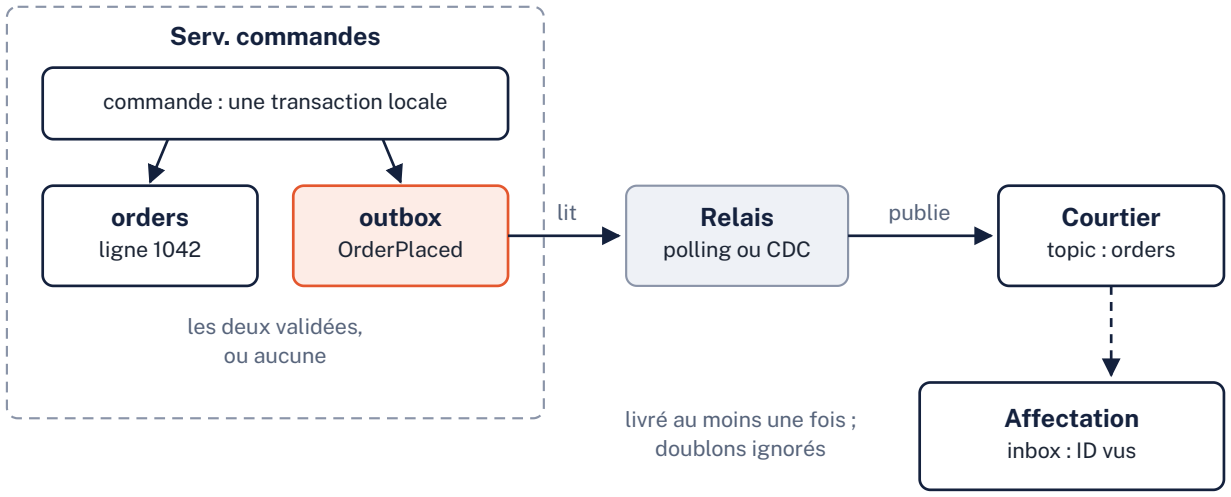
INSERT INTO orders (id, customer_id, state, total_pence)
VALUES ('1042', 'c-77', 'PENDING', 2350);

INSERT INTO outbox (id, aggregate_type, aggregate_id, type, payload)
VALUES ('e-9f1c', 'order', '1042', 'OrderCreated',
        '{"orderId":"1042","customerId":"c-77","totalPence":2350}');

COMMIT;
-- A relay (poller or CDC such as Debezium) later publishes e-9f1c to the broker.
```

Passer une commande avec une outbox transactionnelle : la commande et son événement sont validés ensemble, dans une seule transaction locale.

L'outbox à la sortie, l'inbox à l'entrée



L'outbox supprime la double écriture côté émetteur ; l'inbox, ou tout autre contrôle des doublons, sécurise le côté récepteur en cas de livraison au moins une fois.

Recevoir de façon fiable : récepteurs idempotents et inbox

Comme l'outbox, les nouvelles tentatives du courtier et celles de la saga produisent toutes des doublons, chaque participant doit être un récepteur idempotent (Idempotent Receiver), nom que Gregor Hohpe et Bobby Woolf ont donné à ce patron dans *Enterprise Integration Patterns* (2003). Certaines opérations sont naturellement idempotentes (« passer la commande 1042 à CONFIRMED ») ; d'autres nécessitent un contrôle des doublons. L'implémentation habituelle est une **inbox** : une table des ID de messages traités, écrite dans la même transaction locale que la modification métier, de sorte que « traiter le message » et « se souvenir qu'il a été traité » soient validés ensemble. Un message livré une nouvelle fois trouve son ID déjà présent et est acquitté sans rien faire. Le code côté consommateur figure dans la section sur l'architecture pilotée par les événements.

Ensemble, l'outbox, la livraison au moins une fois et les récepteurs idempotents donnent le résultat pratique que l'on désigne généralement par « exactement une fois » : chaque effet métier se produit une seule fois, même si les messages peuvent circuler plus d'une fois.

GUIDE DE DÉCISION

Guide de décision : une transaction, une saga ou de nouvelles frontières ?

Utilisez-le quand une action métier doit modifier des données à plus d'un endroit.

Q1 **Toutes les données qui doivent changer ensemble appartiennent-elles à un seul service et sont-elles stockées dans une seule base de données ?**

- **Oui** Utilisez une seule transaction ACID locale. C'est l'option la plus simple et la plus solide ; ne la distribuez pas.
- **Non** Passez à la question suivante.

Q2 **Les parties qui doivent changer ensemble changent-elles presque toujours ensemble, appartiennent-elles à une seule équipe et partagent-elles un contexte délimité, si bien que le découpage est accidentel ?**

- **Oui** Revoyez la frontière : fusionnez les services, ou déplacez les données, pour que l'action devienne une seule transaction locale. Les sagas ne corrigent pas un mauvais découpage.
- **Non** Passez à la question suivante.

Q3 **Chaque participant doit-il voir toutes les modifications au même instant, sans qu'aucun état intermédiaire ne soit jamais visible (par exemple, un registre comptable qui doit toujours être équilibré) ?**

- **Oui** Gardez ces enregistrements dans une seule base de données, ou utilisez une base de données distribuée qui fournit en interne des transactions entre partitions. N'utilisez le 2PC qu'à l'intérieur des frontières d'un même système, avec des participants que vous maîtrisez.
- **Non** Passez à la question suivante.

Q4 **Chaque étape peut-elle soit être compensée, soit être placée après un pivot et réessayée jusqu'à ce qu'elle réussisse ?**

- **Oui** Utilisez une saga : orchestrée si elle compte plus de trois étapes environ ou le moindre branchement, chorégraphiée si elle est courte et stable. Ajoutez une outbox, des récepteurs idempotents et des contre-mesures comme les verrous sémantiques.
- **Non** Réordonnez les étapes ou modifiez le processus métier (par exemple, réserver avant de débiter, ou ajouter une vérification manuelle) jusqu'à ce que ce soit le cas ; sinon, reconsidérez les frontières.

ATTENTION

Erreurs courantes

- Publier un événement après la validation, sans outbox, et perdre des événements chaque fois qu'un service redémarre au mauvais moment.
- Écrire les compensations sous la forme « supprimer la ligne », ce qui détruit la piste d'audit et déroute quiconque l'avait déjà vue.
- Oublier que les compensations peuvent échouer ou arriver avant l'étape qu'elles annulent.
- Placer une étape irréversible, comme l'envoi de la commande en cuisine ou le débit de la carte, avant une étape qui échoue souvent.
- Utiliser la chorégraphie pour un flux long et ramifié, jusqu'à ce que plus personne ne sache dire ce qui se passe quand l'étape 5 échoue.
- Utiliser des sagas pour masquer un découpage en services qui traverse une frontière de cohérence ; la solution la moins coûteuse consiste souvent à déplacer la frontière.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Pourquoi un participant à une validation en deux phases ne peut-il pas simplement valider de lui-même si le coordinateur disparaît après qu'il a voté oui ?**

Parce qu'il ne connaît pas la décision : un autre participant a peut-être voté non, et le coordinateur a donc peut-être décidé d'annuler ; deviner pourrait rompre l'atomicité.

▼ **Dans la saga de commande, pourquoi la confirmation de la commande au restaurant est-elle le pivot plutôt que la première étape ?**

Une fois que la cuisine commence à préparer le repas, l'étape ne peut plus être annulée : elle doit donc venir après les étapes susceptibles d'échouer (autorisation de paiement, réservation du livreur), qui peuvent être compensées à peu de frais.

▼ **Quel problème l'outbox transactionnelle résout-elle, et quelle garantie offre-t-elle ?**

Elle supprime la double écriture entre la base de données et le courtier en validant ensemble la modification et le message ; le relais livre ensuite le message au moins une fois, si bien que des doublons sont possibles.

▼ **Citez deux différences entre une compensation et un retour arrière de base de données.**

C'est une nouvelle transaction qui s'exécute après la validation de la transaction d'origine, et d'autres ont peut-être déjà vu celle-ci ou agi en conséquence ; elle peut aussi n'être qu'une annulation sémantique (un remboursement, une annulation d'autorisation) plutôt qu'un retour exact à l'état précédent.

▼ **À vous : Une saga de billetterie comporte ces étapes : autoriser le paiement, bloquer les places, émettre le billet, envoyer un e-mail au fan. Les places s'avèrent indisponibles à la deuxième étape. Que se passe-t-il ? Quelle étape est le pivot, et pourquoi une compensation n'est-elle pas un retour arrière ?**

La saga compense en sens inverse : elle annule l'autorisation de paiement, et le fan est informé que les places sont parties. L'émission du billet est le pivot, car une fois qu'un billet existe, il ne peut pas simplement disparaître, et les étapes suivantes, comme l'e-mail, doivent être retentées jusqu'à leur réussite. Une compensation est une nouvelle transaction vers l'avant, et non un rollback : d'autres ont pu voir le blocage, elle doit donc être idempotente et pouvoir être réessayée sans risque.

SOURCES

- Garcia-Molina and Salem: Sagas (Princeton technical report, 1987): https://www.cs.princeton.edu/research/techreps/598
- Chris Richardson: Pattern: Saga (microservices.io): https://microservices.io/patterns/data/saga.html
- Chris Richardson: Pattern: Transactional outbox (microservices.io): https://microservices.io/patterns/data/transactional-outbox.html
- Microsoft Azure Architecture Center: Saga design pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/saga
- Debezium documentation: Outbox event router: https://debezium.io/documentation/reference/transformations/outbox-event-router.html
- Temporal: Understanding Temporal: https://docs.temporal.io/evaluate/understanding-temporal
- Temporal: Workflows: https://docs.temporal.io/workflows
- AWS Step Functions: Standard and Express workflows: https://docs.aws.amazon.com/step-functions/latest/dg/concepts-standard-vs-express.html

SECTION 6.8 SUR 6.8 · INTERMÉDIAIRE · 28 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Architectures serverless et basées sur l'espace

AVANT DE LIRE

Cette section s'appuie sur :

- 0.7 Serveurs et cloud
- 6.1 Pourquoi distribuer
- 6.4 Pilotée par les événements

Termes déjà vus qu'elle utilise : Service managé, Idempotent, Défaillance partielle, Cohérence à terme (eventual consistency), Réplique en lecture (read replica)

L'IDÉE EN BREF

Pensez à deux façons de faire face à la clientèle. Une station de taxis ne fait payer que chaque trajet : quand personne n'a besoin d'une course, vous ne payez rien, mais lors d'une soirée chargée chaque trajet coûte plus cher que de posséder une voiture, et le premier taxi peut mettre un moment à arriver. C'est le serverless. Imaginez maintenant la billetterie d'un stade avant un énorme concert : au lieu que chaque guichetier fasse la queue pour consulter un registre central unique, chaque guichet garde sa propre copie des places libres et transmet ses ventes au back-office plus tard. C'est l'architecture basée sur l'espace, conçue pour la ruée.

CE QU'IL FAUT SAVOIR

- Le serverless combine les fonctions en tant que service, où le fournisseur exécute du code déclenché par des événements et le met à l'échelle à partir de zéro, et le backend en tant que service, où des services managés remplacent les composants côté serveur ; AWS Lambda (annoncé en 2014) a démocratisé le FaaS.
- Les fonctions sont sans état et de courte durée, facturées par invocation et par Go-seconde, sujettes aux démarrages à froid et à des limites strictes de durée, de mémoire et de taille des données, et souvent déclenchées au moins une fois : les gestionnaires doivent donc être idempotents.
- Tout l'état durable réside dans des services managés : un système serverless est donc par construction un système distribué, et il est fortement couplé à son fournisseur, sauf si la logique métier est maintenue derrière des ports.

- La tarification par requête est moins chère qu'une capacité permanente en dessous d'un volume de bascule, et plus chère au-delà ; ce point de bascule dépend de la durée, de la mémoire, de la forme du trafic et des coûts humains.
- L'architecture basée sur l'espace sert les requêtes depuis des unités de traitement dotées de grilles de données en mémoire répliquées, gérées par un middleware virtualisé, et met à jour la base de données de façon asynchrone au moyen de pompes et de rédacteurs de données, des lecteurs de données servant à préchauffer la grille.
- Son nom vient de l'espace de tuples de Linda, le langage de Gelernter (1985) ; ses principaux risques sont les collisions de données entre répliques, les limites de mémoire, le retard de la base de données, le coût et la complexité.
- N'utilisez l'architecture basée sur l'espace que pour une charge extrême, en pics, que la mise en cache, les répliques et les salles d'attente virtuelles ne peuvent pas absorber, et routez ou partitionnez les données disputées pour qu'elles ne puissent pas entrer en collision.

Deux réponses à la question de la capacité

Tous les styles vus jusqu'ici dans ce chapitre supposent que quelqu'un exploite des serveurs et les dimensionne en fonction de la charge. Cette section présente deux styles qui répondent à la question de la capacité de manières opposées. L'architecture serverless confie entièrement les serveurs à un fournisseur cloud et paie à l'usage, ce qui convient à un travail de petite taille, piloté par les événements et souvent inactif. L'architecture basée sur l'espace garde en mémoire tout ce dont une requête a besoin en période de forte charge, réparti sur de nombreuses unités identiques, afin que la base de données ne soit plus sur le chemin critique, ce qui convient à des pics extrêmes et soudains comme une mise en vente de billets. Ce sont deux outils spécialisés ; tous deux sont souvent choisis pour de mauvaises raisons.

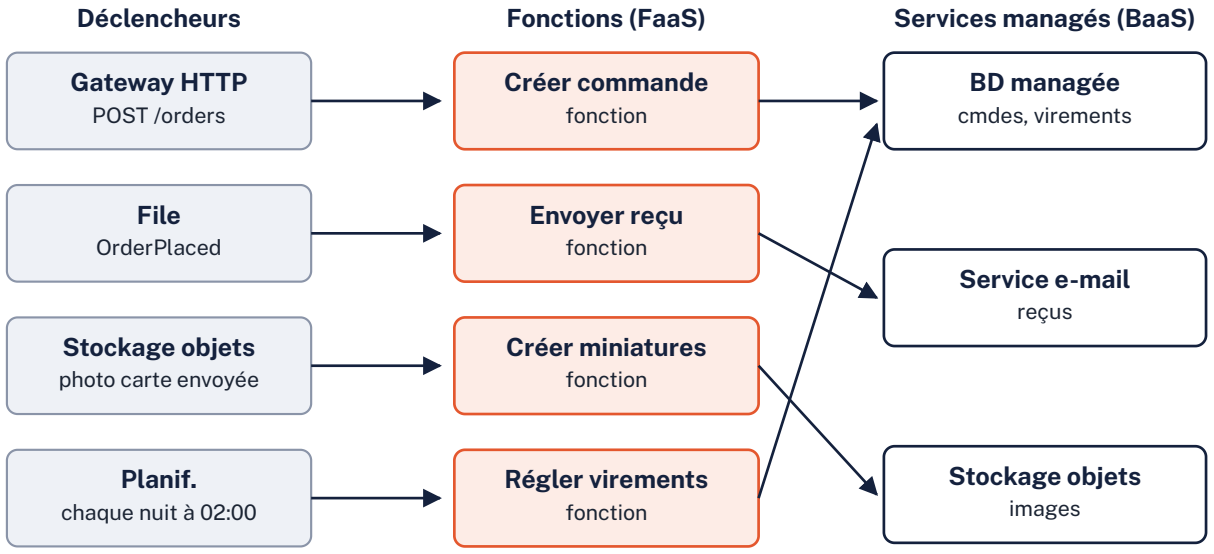
Serverless : fonctions et back-ends managés

Le serverless ne veut pas dire qu'il n'y a pas de serveurs ; cela signifie que l'équipe n'a pas à provisionner, corriger, dimensionner ni payer des serveurs inactifs. L'article de Mike Roberts, publié en 2018 et largement cité, distingue deux ingrédients. Les fonctions en tant que service (FaaS) exécutent de petits morceaux du code propre à l'équipe en réponse à des événements, dans des conteneurs que le fournisseur crée, met à l'échelle et détruit, et qui peuvent ne vivre que le temps d'une seule invocation. AWS Lambda, annoncé en novembre 2014 et disponible pour tous en avril 2015, a démocratisé ce modèle ; Azure Functions, Google Cloud Run functions et Cloudflare Workers en sont d'autres exemples. Le backend en tant que service

(BaaS) remplace des composants côté serveur par des services tiers entièrement managés, que le client ou la fonction appelle directement : authentification (Auth0, Amazon Cognito), bases de données (Firebase, DynamoDB), stockage d'objets, e-mail et files d'attente.

Une application serverless est généralement une combinaison : les fonctions contiennent la logique métier propre à l'organisation, et tout le reste, y compris tout l'état durable, réside dans des services managés. Le fournisseur gère la capacité, la disponibilité entre zones et le système d'exploitation. L'équipe reste responsable de son code, de ses données, de sa configuration et de ses paramètres de sécurité, comme l'explique le modèle de responsabilité partagée présenté dans les serveurs et le cloud.

Les événements déclenchent les fonctions ; l'état réside ailleurs



Chaque instance de fonction est sans état et éphémère ; tout l'état durable réside dans des services managés

Sur une plateforme de livraison de repas, quatre types d'événements déclenchent quatre fonctions, qui lisent et écrivent dans des services managés au lieu de conserver quoi que ce soit elles-mêmes.

Comment se comportent les fonctions : les caractéristiques qui façonnent la conception

Déclencheurs d'événements. Les fonctions s'exécutent en réponse à des événements : une requête HTTP passant par une gateway, un message dans une file ou un flux, un fichier déposé dans un stockage d'objets, une modification de base de données, ou une planification. Les déclencheurs qui invoquent la fonction de façon asynchrone, comme les files et les notifications de stockage, livrent généralement *au moins une fois* et réessaient en cas d'échec : les gestionnaires doivent donc être idempotents, exactement comme dans l'architecture pilotée par les événements.

Mise à l'échelle jusqu'à zéro et à l'horizontale. Sans trafic, aucune instance ne tourne et aucun calcul n'est facturé. À mesure que les événements arrivent, la plateforme démarre davantage d'instances en parallèle, jusqu'à une limite de concurrence fixée pour le compte ou pour la fonction ; sur AWS Lambda, la limite régionale par défaut est de 1000 exécutions simultanées au moment de la rédaction, et peut être relevée sur demande. Ce passage à l'échelle rapide est à la fois un atout et un danger : mille fonctions simultanées peuvent ouvrir mille connexions à la base de données et submerger une base de données relationnelle conçue pour quelques centaines, d'où l'existence des proxys de connexions et des plafonds de concurrence par fonction.

Facturation par invocation. Le coût est généralement une petite somme par requête, plus un montant pour la mémoire multipliée par le temps d'exécution (en gigaoctets-secondes). Un code qui attend sans rien faire une API en aval lente paie donc pour attendre.

Démarrages à froid (cold starts). La première requête adressée à une nouvelle instance doit attendre que la plateforme démarre un conteneur, charge l'environnement d'exécution et exécute le code d'initialisation : de quelques dizaines de millisecondes à plusieurs secondes, selon le langage, la taille du paquet et la configuration réseau. Les instances sont réutilisées pour les requêtes suivantes tant qu'elles restent chaudes, si bien que les démarrages à froid frappent surtout le trafic peu fréquent et les pics soudains. Parmi les parades figurent des paquets de déploiement plus petits, des environnements d'exécution plus légers, l'initialisation en dehors du gestionnaire, et le fait de payer pour garder des instances pré-initialisées (AWS appelle cela la concurrence provisionnée, ou provisioned concurrency), ce qui réintroduit en partie un coût permanent.

Limites. Les fonctions s'exécutent sous des limites strictes de durée d'exécution, de mémoire, de taille des données transmises et de disque temporaire ; pour AWS Lambda, au moment de la rédaction : 15 minutes, de 128 Mo à 10 240 Mo de mémoire (avec un CPU alloué en proportion) et 6 Mo pour une requête ou une réponse synchrone. Les tâches longues doivent être découpées en étapes, et les grosses données transmises par référence au stockage d'objets.

Absence d'état. Rien de ce qui se trouve dans la mémoire ou sur le disque local d'une instance n'est garanti de survivre jusqu'à l'invocation suivante. La mise en cache en mémoire entre deux appels est une optimisation qui peut fonctionner ou non, jamais un endroit où mettre des données importantes.

SI VOUS DÉBUTEZ

Pas à pas : une fonction qui oublie

Le point qui fait le plus souvent trébucher ici est qu'une fonction serverless ne peut compter sur rien conserver d'une exécution à la suivante : chaque information importante doit donc se trouver ailleurs. Une petite erreur montre pourquoi.

- L'équipe de la livraison de repas écrit une fonction qui s'exécute chaque fois qu'une commande est passée. Pour afficher en direct le nombre de commandes du jour, un développeur garde un compteur dans la mémoire de la fonction et y ajoute un à chaque fois.
- Par une matinée calme, cela semble fonctionner : une seule copie de la fonction traite les commandes l'une après l'autre, et le compteur passe à 1, 2, 3.
- À l'heure du déjeuner, les commandes affluent et la plateforme démarre 50 copies de la fonction à la fois, chacune avec sa propre mémoire. Chaque copie tient son propre compte séparé, si bien qu'aucune ne connaît le vrai total.
- Quand le trafic retombe, la plateforme supprime discrètement les copies dont elle n'a plus besoin, et leurs comptes disparaissent avec elles. La première commande du lendemain peut repartir de 1, ou de la valeur dont une copie survivante se souvient par hasard.
- La solution consiste à garder le compteur dans un service managé partagé par toutes les copies, comme une base de données ou un cache, et à utiliser la commande « ajouter un » propre à ce service, pour que deux copies qui ajoutent en même temps ne puissent pas s'écraser l'une l'autre.
- **La leçon :** écrivez chaque fonction pour qu'elle soit sans état. Considérez tout ce qui se trouve dans sa mémoire comme un brouillon qui peut disparaître à tout moment, et décidez où se trouve chaque information avant d'écrire la fonction ; c'est l'objet de la partie suivante de cette section.

Où va l'état

Comme les fonctions ne conservent rien, le travail d'architecture se déplace vers le choix de l'endroit où réside chaque type d'état. Les données métier durables vont dans une base de données managée, souvent une base qui gère bien de nombreuses connexions de courte durée (bases clé-valeur ou orientées documents, ou base de données relationnelle derrière un proxy de connexions). Les gros objets vont dans le stockage d'objets et passent d'une fonction à l'autre par leur clé. L'état partagé de courte durée, comme les compteurs de limitation de débit, va dans un cache managé. La progression d'un processus en plusieurs étapes va dans un

service de workflow (AWS Step Functions, Azure Durable Functions ou équivalent), qui joue le rôle d'orchestrateur de saga durable, comme décrit dans la section sur les sagas. Les sessions utilisateur vont dans des jetons conservés par le client ou dans un magasin de sessions.

La conséquence est qu'un système serverless est par construction un système distribué, avec toutes les préoccupations de ce chapitre : latence réseau entre chaque fonction et ses données, défaillance partielle, événements en double, cohérence à terme et besoin de traçage distribué. Une seule action métier peut traverser une gateway, trois fonctions, une file et deux bases de données. L'observabilité (voir observabilité) n'est pas facultative.

Couplage au fournisseur : le prix de ne pas exploiter de serveurs

Le serverless couple un système à son fournisseur plus étroitement que la location de machines virtuelles ou l'exécution de conteneurs. Les formats d'événements, la configuration des déclencheurs, les modèles d'identité et de permissions, les définitions de workflows et les services managés eux-mêmes sont tous propres au fournisseur. Migrer une application serverless mature vers un autre cloud tient davantage de la réécriture que de la migration.

Les praticiens ne s'accordent pas sur l'importance de ce point. Selon un premier point de vue, la dépendance au fournisseur (lock-in) est le prix d'une forte réduction du travail d'exploitation, et la plupart des organisations ne changent jamais de cloud : optimiser pour la portabilité serait donc un effort perdu. Selon l'autre, les changements de tarifs, les exigences réglementaires ou les acquisitions imposent bel et bien des migrations, et un couplage profond retire tout pouvoir de négociation. Une voie médiane consiste à accepter le couplage en périphérie et à le contenir dans le cœur : garder la logique métier dans des modules ordinaires derrière des ports, avec de minces gestionnaires de fonctions jouant le rôle d'adaptateurs (l'approche de l'architecture hexagonale), gérer l'infrastructure en tant que code, et consigner le couplage comme une décision explicite. Le coût et la dépendance au fournisseur sont approfondis dans FinOps et dépendance au fournisseur.

EXEMPLE COMMENTÉ

Exemple commenté : quand une capacité toujours active devient-elle moins chère ?

L'API de carte des restaurants de la plateforme de livraison de repas pourrait tourner sous forme de fonctions ou sur deux machines virtuelles toujours actives. **Tous les prix sont des chiffres ronds illustratifs choisis pour faciliter le calcul, et non les prix catalogue actuels d'un quelconque fournisseur.** Chaque requête s'exécute en moyenne pendant 200 ms avec 0,5 Go de mémoire : elle consomme donc 0,1 Go-seconde.

Prix serverless illustratifs	Calcul : £0,00002 par Go-seconde ; £0,20 par million d'invocations ; £1,00 par million de requêtes passant par la gateway HTTP.
Coût serverless par million de requêtes	Calcul : 1 000 000 × 0,1 Go-s × £0,00002 = £2,00. Invocations : £0,20. Gateway : £1,00. Total : £3,20 par million.
Coût illustratif d'une capacité toujours active	Deux petites VM à £40 par mois chacune (deux pour la redondance), plus un équilibreur de charge à £20 = £100 par mois, dimensionnées pour des pics d'environ 300 requêtes par seconde.
1 million de requêtes par mois (0,4 par seconde en moyenne)	Serverless : £3,20 ; toujours actif : £100. Le serverless est environ 30 fois moins cher.
10 millions par mois (environ 4 par seconde)	Serverless : £32 ; toujours actif : £100.
Point de bascule	£100 ÷ £3,20 ≈ 31 millions de requêtes par mois, soit environ 12 requêtes par seconde en moyenne (31,25 millions ÷ 2,59 millions de secondes en 30 jours).
100 millions par mois (environ 39 par seconde)	Serverless : £320 ; toujours actif : toujours £100, car deux VM ont encore de la marge.
Effet d'une réduction de moitié de la durée, à 100 ms	Le calcul tombe à £1,00, le total à £2,20 par million, et le point de bascule passe à environ 45 millions. En serverless, l'optimisation est directement rentable.

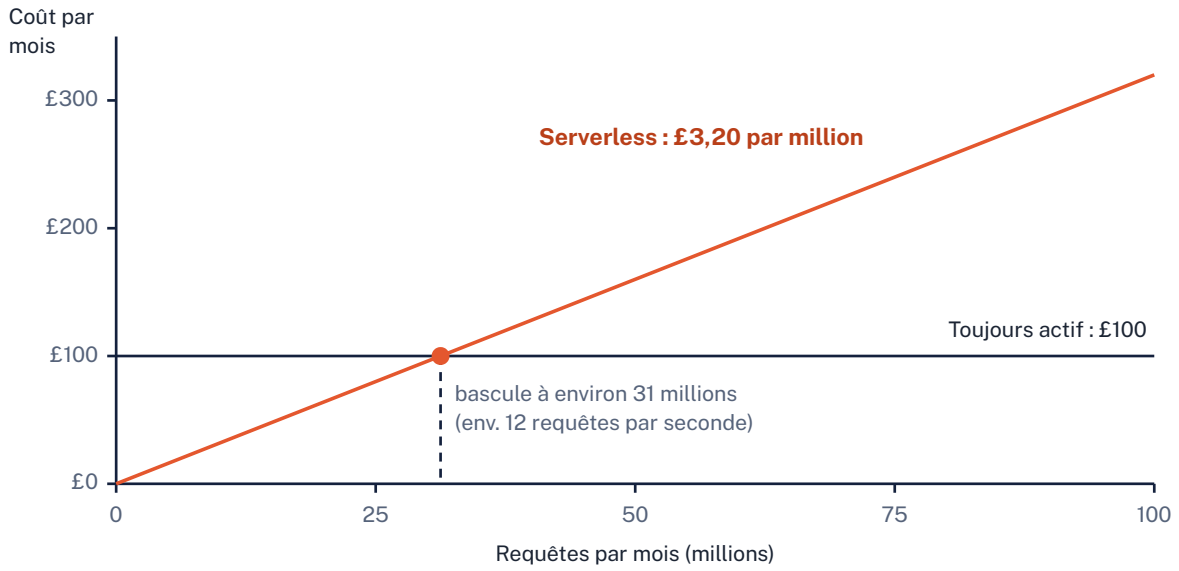
En dessous du point de bascule, le serverless est moins cher parce que le temps d'inactivité est gratuit ; au-dessus, un service constamment chargé paie davantage pour le même travail. Un trafic en rafales, avec de longues périodes d'inactivité, favorise le serverless même à des volumes plus élevés.

La comparaison brute laisse de côté le coût le plus important pour beaucoup d'équipes : les personnes. Les correctifs, la mise à l'échelle et l'astreinte pour des VM peuvent facilement coûter plus que la différence, ce qui repousse vers le haut le point de bascule

réel. Les frais de transfert de données, de journalisation et d'observabilité s'appliquent aux deux et sont souvent plus élevés que l'une ou l'autre facture de calcul.

Comme le résultat dépend de la durée, de la mémoire, de la forme du trafic et des prix en vigueur, refaites le calcul avec des prix réels et un trafic mesuré avant de décider, puis à nouveau à mesure que le trafic augmente.

Le point de bascule, en graphique



Avec ces prix illustratifs, le coût serverless augmente de £3,20 par million de requêtes et rejoint le coût fixe de £100 de la capacité toujours active à environ 31 millions de requêtes par mois.

Quand le serverless convient, et quand il ne convient pas

Le serverless convient aux charges de travail **en pics, imprévisibles ou de faible volume**, pour lesquelles payer des serveurs inactifs est du gaspillage ; au **code de liaison piloté par les événements**, par exemple pour réagir à des envois de fichiers, à des messages de file et à des modifications de base de données ; aux **tâches planifiées** et à l'automatisation du back-office ; aux **prototypes et nouveaux produits**, pour lesquels l'équipe doit consacrer son temps au produit plutôt qu'à l'infrastructure ; et aux équipes sans capacité de plateforme ni d'exploitation.

Il convient mal à un trafic **régulier et à fort volume**, pour lequel la tarification par requête dépasse le coût d'une capacité toujours active (comme dans l'exemple commenté) ; aux processus **de longue durée ou avec état**, comme le transcodage vidéo au-delà de la limite de durée ou les serveurs WebSocket qui maintiennent de nombreuses connexions ; aux exigences **strictes de faible latence de queue** que les démarrages à froid ne permettraient pas de

respecter ; aux charges de travail nécessitant du matériel spécial ou un contrôle fin de l'environnement d'exécution ; et aux conceptions à grain fin dans lesquelles de nombreuses fonctions s'appellent les unes les autres de façon synchrone, ce qui ajoute à chaque étape des sauts réseau, des coûts et des points de défaillance. Le chapitre 1 a décrit comment Amazon Prime Video, en 2023, a fait passer un service de supervision d'étapes serverless distribuées à un processus unique, et a fait état d'une baisse de plus de 90 % du coût d'infrastructure (pourquoi l'architecture compte) : ce n'est pas un verdict sur le serverless, mais un rappel que l'adéquation compte plus que la mode.

L'architecture basée sur l'espace : sortir la base de données du chemin critique

La plupart des architectures font facilement passer à l'échelle les niveaux web et applicatif, mais finissent par buter sur la base de données, que chaque requête sollicite. Ajouter des serveurs en amont ne fait que déplacer la file d'attente vers la base de données. L'architecture basée sur l'espace (space-based), telle que la décrivent Mark Richards et Neal Ford dans *Fundamentals of Software Architecture*, retire complètement la base de données du chemin synchrone. Chaque requête est traitée entièrement en mémoire par l'une des nombreuses unités de traitement identiques, et la base de données est mise à jour de façon asynchrone, en coulisses.

Le nom vient de l'espace de tuples, une mémoire associative partagée introduite en 1985 par Linda, le langage de coordination de David Gelernter : les processus communiquent en écrivant des tuples dans un espace partagé et en y lisant ou en y prélevant les tuples correspondants, sans se connaître les uns les autres. JavaSpaces, de Sun, a apporté l'idée à Java à la fin des années 1990, et des plateformes commerciales ont bâti sur elle des espaces distribués et répliqués, qui ont évolué vers les grilles de données en mémoire actuelles comme Hazelcast, Apache Ignite et Oracle Coherence.

Les composants d'un système basé sur l'espace

1. **Unité de traitement**
   Unité déployable contenant la logique applicative et une grille de données en mémoire (in-memory data grid) qui détient les données dont cette logique a besoin, ainsi qu'un moteur de réplication. Les unités sont identiques : en ajouter une ajoute de la capacité. Dans l'exemple de la billetterie, chaque unité détient le plan des places et les réservations temporaires en cours pour les événements qu'elle sert.

2

Middleware virtualisé

L'infrastructure qui gère les unités. La **grille de messagerie** reçoit les requêtes et achemine chacune vers une unité disponible. La **grille de données** réplique les modifications entre les copies en mémoire des unités. La **grille de traitement**, facultative, coordonne les requêtes qui nécessitent plusieurs types d'unités. Le **gestionnaire de déploiement** démarre et arrête des unités à mesure que la charge augmente et diminue.

3

Pompes de données

Canaux asynchrones, généralement des files de messages persistantes, qui transportent chaque modification faite dans une unité vers le côté base de données. Si la base de données est lente ou indisponible, les modifications attendent dans la file ; les acheteurs n'en sont pas affectés.

4

Rédacteurs de données

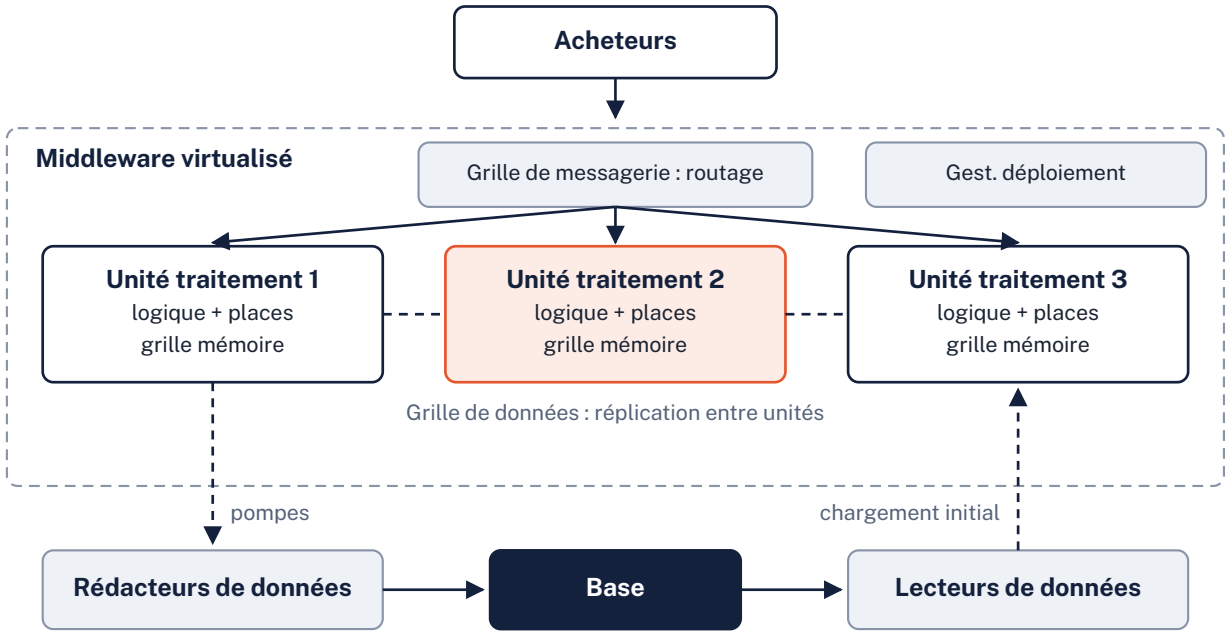
Composants qui consomment les messages des pompes et écrivent les modifications dans la base de données. Ils peuvent être mis à l'échelle et mis en pause indépendamment des unités.

5

Lecteurs de données

Composants qui chargent les données de la base de données dans les unités quand la grille démarre à froid, par exemple après le redémarrage ou le redéploiement de toutes les unités. Ils ne servent pas au traitement normal des requêtes.

L'architecture basée sur l'espace pour une mise en vente de billets



Les requêtes sont servies depuis la mémoire par des unités de traitement identiques ; la base de données est mise à jour de façon asynchrone par des pompes et des rédacteurs de données, et n'est lue que pour préchauffer la grille.

Collisions de données et autres coûts

Conserver des copies répliquées dans de nombreuses unités transpose en mémoire les problèmes de cohérence vus dans les sections précédentes. Avec un cache **répliqué**, chaque unité dispose d'une copie complète et les modifications sont copiées de façon asynchrone, ce qui permet des lectures locales rapides, mais deux unités peuvent mettre à jour le même élément avant que la réplication ne rattrape son retard. Richards et Ford appellent cela une collision de données et en estiment le taux comme le nombre d'unités multiplié par le carré du taux de mise à jour, divisé par le nombre d'éléments, multiplié par la latence de réplication. Avec 5 unités, 20 mises à jour par seconde, 50 000 éléments et 100 ms de latence, on obtient 5 × 400 ÷ 50 000 × 0,1 = 0,004 par seconde, soit environ 115 collisions sur une journée de huit heures ; réduire la latence à 1 ms ramène ce chiffre à environ une.

Pour les données qui ne doivent jamais entrer en collision, comme la dernière place d'une rangée, les alternatives sont un cache **distribué**, dans lequel chaque élément réside dans une seule partition, où chaque unité le lit et l'écrit (c'est cohérent, mais chaque accès est un appel réseau et les partitions de la grille doivent rester disponibles), ou l'acheminement de toutes les requêtes portant sur un même événement ou une même section vers la même unité, pour

qu’une seule unité modifie jamais ces places. La plupart des conceptions réelles combinent les deux : partitionner par clé les données chaudes et disputées, et répliquer les données surtout lues, comme les plans de salle et les prix.

Les autres coûts sont importants. Les données conservées en mémoire sont limitées par la taille de la mémoire et doivent être reconstruites après un redémarrage complet. La base de données est cohérente à terme avec la grille : les rapports et les autres systèmes voient donc les modifications avec retard, et la perte de modifications en file d’attente est un risque réel que la durabilité des pompes doit couvrir. Les tests à une charge de pointe réaliste sont coûteux et difficiles. Et ce style est complexe et coûteux à construire : Richards et Ford lui attribuent une note faible en simplicité et en testabilité, et élevée en coût. Il n’en vaut la peine que lorsque la charge ne peut vraiment pas être absorbée par la mise en cache, les répliques en lecture et une base de données bien réglée.

EXEMPLE COMMENTÉ

Exemple commenté : une mise en vente pour un stade

Les billets d'une tournée des stades sont mis en vente à 10:00. Environ 400 000 personnes attendent ; pendant les dix premières minutes, la demande représente des centaines de fois le niveau normal, puis elle retombe. Le système de billetterie ne doit pas vendre deux fois la même place et ne doit pas s'effondrer.

Avant la mise en vente	Les lecteurs de données chargent dans la grille le plan des places de chaque spectacle. Le gestionnaire de déploiement démarre suffisamment d'unités de traitement pour le pic prévu.
Routage	La grille de messagerie achemine les requêtes par spectacle et par section, si bien que toutes les réservations temporaires pour la section 112 du spectacle du samedi sont traitées par la même unité. Les réservations temporaires n'entrent jamais en collision.
Un acheteur sélectionne deux places	L'unité les marque HELD pour cet acheteur pendant 8 minutes, en mémoire, en quelques microsecondes, et répond. Les plans des places des autres acheteurs se mettent à jour par réplication.
La réservation temporaire devient une vente	L'étape de paiement s'exécute (un appel externe, avec une clé d'idempotence) ; en cas de succès, l'unité marque les places SOLD et une pompe de données met la vente en file d'attente.
La base de données	Les rédacteurs de données écrivent les ventes à un rythme régulier que la base de données peut soutenir. Pendant le pic, la file grossit ; elle se vide quelques minutes après la baisse de la demande. Aucune requête d'acheteur n'attend la base de données.
Une réservation temporaire expire	L'unité libère les places en mémoire ; la pompe met en file d'attente un enregistrement de libération.

De nombreux systèmes de billetterie placent aussi une salle d'attente virtuelle devant la vente, qui admet les acheteurs à un rythme contrôlé. C'est beaucoup plus simple, et souvent suffisant à soi seul ; l'architecture basée sur l'espace est faite pour les cas où la charge admise reste elle-même trop élevée pour une conception centrée sur la base de données.

Le routage par section est la décision de conception qui fait passer le problème des collisions de « rare mais possible » à « impossible » pour les données les plus importantes.

Caractéristiques comparées (qualitatives, pour des conceptions typiques)

Caractéristique	Serverless (FaaS et BaaS)	Basée sur l'espace
Élasticité	Très élevée, automatique, y compris la réduction à zéro	Très élevée pour les pics, mais les unités mettent du temps à démarrer et à charger les données
Performances	Bon niveau à chaud ; les démarrages à froid et les sauts réseau ajoutent de la latence	Niveau très élevé : requêtes servies depuis la mémoire
Profil de coût	Paiement à l'usage ; excellent en cas d'inactivité ou de pics, cher en cas de charge constante	Élevé : nombreuses unités, beaucoup de mémoire, middleware complexe, toujours actif
Simplicité	Simple par fonction ; le système dans son ensemble peut devenir difficile à suivre	Faible : réplication, collisions, pompes et chargement au démarrage à froid
Cohérence	Ce que fournissent les magasins de données managés	Réplication en mémoire à terme, sauf si les données disputées sont partitionnées ; la base de données est en retard
Testabilité	Les unités sont faciles à tester ; l'intégration nécessite des environnements cloud	Difficile : les tests de pointe réalistes coûtent cher
Effort d'exploitation	Faible pour l'infrastructure ; élevé pour l'observabilité et les limites	Élevée
Portabilité	Faible : déclencheurs et services propres au fournisseur	Moyenne : dépend du produit de grille de données

Liste de contrôle : avant de choisir le serverless ou l'architecture basée sur l'espace

Répondez-y avec des chiffres mesurés quand vous le pouvez.

- [ ] **Forme du trafic**
  La charge est-elle en pics, souvent inactive ou régulière ? Le serverless récompense l'inactivité ; l'architecture basée sur l'espace récompense les pics extrêmes ; une charge régulière favorise des services ordinaires toujours actifs.
- [ ] **Point de bascule des coûts**
  Avez-vous comparé le coût par requête au coût d'une capacité toujours active avec des prix, des durées et un trafic réels, en incluant la gateway, le transfert de données et les personnes ?
- [ ] **Budget de latence**
  Le temps de réponse le plus lent acceptable peut-il absorber un démarrage à froid, ou le saut réseau jusqu'à l'endroit où réside l'état ?
- [ ] **Limites**
  Des tâches dépassent-elles les limites de durée d'exécution, de mémoire ou de taille des données ? Peuvent-elles être découpées en étapes ou leurs données transmises par référence ?
- [ ] **État et connexions**
  Où réside chaque type d'état, et les magasins de données peuvent-ils supporter des rafales soudaines de connexions simultanées ?
- [ ] **Idempotence**
  Chaque gestionnaire peut-il s'exécuter deux fois sans risque pour le même événement ?
- [ ] **Contention**
  Pour les conceptions basées sur l'espace, quelles données sont disputées, et seront-elles partitionnées ou routées de façon à ne pas pouvoir entrer en collision ?
- [ ] **Durabilité des écritures en attente**
  Si la grille ou les pompes tombent en panne, que perd-on ? Les pompes sont-elles persistantes, et le retard de la base de données est-il acceptable pour les rapports et les autres systèmes ?
- [ ] **Alternatives plus simples**
  La mise en cache, des répliques en lecture, une file d'attente devant la base de données ou une salle d'attente virtuelle répondraient-elles au besoin avec moins de complexité ?

- [ ] **Décision sur la dépendance au fournisseur**
  Le couplage au fournisseur ou au produit de grille est-il consigné comme une décision consciente, avec la logique métier maintenue derrière des ports ?

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi une fonction déclenchée par une file doit-elle être idempotente ?

Parce que les déclencheurs asynchrones livrent généralement au moins une fois et réessaient en cas d'échec : le même événement peut donc invoquer la fonction plus d'une fois.

▼ Avec les prix illustratifs de l'exemple commenté, combien coûteraient environ 50 millions de requêtes par mois en serverless, et quelle option est la moins chère ?

50 × £3,20 = £160 par mois : l'option toujours active à £100 est donc moins chère en coût brut d'infrastructure, avant de compter les personnes et les autres frais.

▼ Dans l'architecture basée sur l'espace, que font les lecteurs de données, et quand ?

Ils chargent les données de la base de données dans les grilles en mémoire des unités de traitement quand la grille démarre à froid, par exemple après un redémarrage complet ou un redéploiement ; ils ne servent pas pour les requêtes normales.

▼ En quoi l'acheminement de toutes les requêtes d'une même section du stade vers une seule unité de traitement aide-t-il ?

Seule cette unité modifie jamais ces places : deux unités ne peuvent donc pas mettre à jour la même place avant que la réplication ne rattrape son retard, ce qui élimine les collisions de données pour ces données.

▼ À vous : Prix indicatifs : calcul £0,00002 par Go-seconde, £0,20 par million d'invocations, £1 par million de requêtes de passerelle. Chaque requête prend 300 ms à 1 Go. Deux serveurs toujours allumés coûtent £288 par mois. Trouvez le point de bascule. Que choisiriez-vous à 5 millions de requêtes ?

Chaque requête utilise 0,3 Go-seconde, soit £6 par million, et le total est donc de £7,20 par million. Point de croisement : 288 / 7,20 = 40 millions de requêtes par mois. À 5 millions, le serverless coûte £36 contre £288 : choisissez donc les fonctions. La forme du trafic et les coûts humains comptent aussi, et les démarrages à froid peuvent l'exclure si la latence est serrée. Si le volume approche 40 millions, le calcul s'inverse : gardez donc la logique derrière des ports pour faciliter le changement.

SOURCES

- Mike Roberts: Serverless Architectures (martinfowler.com, 2018): https://martinfowler.com/articles/serverless.html
- AWS: Introducing AWS Lambda (announcement, 13 November 2014): https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-aws-lambda/
- AWS Lambda Developer Guide: Lambda quotas: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html
- Adrian Colyer: Generative Communication in Linda (summary of Gelernter, 1985): https://blog.acolyer.org/2015/02/17/generative-communication-in-linda/

Lectures complémentaires

- *Fundamentals of Software Architecture (2nd edition)*, Mark Richards and Neal Ford (2025), O'Reilly Media. Des chapitres clairs sur les styles à base de services, pilotés par les événements, basés sur l'espace et microservices, avec des évaluations des caractéristiques, et la liste étendue des illusions de l'informatique distribuée.
- *Software Architecture: The Hard Parts*, Neal Ford, Mark Richards, Pramod Sadalage and Zhamak Dehghani (2021), O'Reilly Media. L'analyse des compromis pour les systèmes distribués : quanta d'architecture, granularité des services (facteurs de désintégration et d'intégration), propriété des données et workflows distribués.
- *Building Microservices (2nd edition)*, Sam Newman (2021), O'Reilly Media. Le guide de référence des praticiens sur les microservices, construit autour de la déployabilité indépendante, avec des conseils équilibrés sur les cas où il vaut mieux ne pas les utiliser.
- *Microservices*, James Lewis and Martin Fowler (2014), martinfowler.com. L'article qui a défini le terme pour la majeure partie du secteur, avec ses neuf caractéristiques communes.
  https://martinfowler.com/articles/microservices.html
- *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Gregor Hohpe and Bobby Woolf (2003), Addison-Wesley. Le vocabulaire de la messagerie (canaux, messages d'événement et de commande, récepteurs idempotents, canaux de lettres mortes) qu'utilise encore toute conception pilotée par les événements.
  https://www.enterpriseintegrationpatterns.com/
- *Designing Data-Intensive Applications (2nd edition)*, Martin Kleppmann and Chris Riccomini (2026), O'Reilly Media. La meilleure explication de la réplication, de la cohérence, des journaux, du traitement de flux et des difficultés propres aux systèmes distribués.
  https://martin.kleppmann.com/2026/03/24/designing-data-intensive-applications-2e.html
- *Building Event-Driven Microservices (2nd edition)*, Adam Bellemare (2025), O'Reilly Media. Les flux d'événements comme colonne vertébrale d'une architecture de services : conception des événements, schémas, propriété et traitement de flux.

Glossaire

API gateway (passerelle d'API)
Point d'entrée unique placé devant les services, qui gère des préoccupations transversales comme TLS, l'authentification, la limitation de débit et le routage.

Architecture à base de services (service-based)
Style distribué comprenant quelques services de domaine à gros grain, déployés séparément (généralement de 4 à 12), qui partagent souvent une même base de données ; nommé par Mark Richards et Neal Ford.

Architecture basée sur l'espace (space-based)
Style architectural qui sert les requêtes à partir de nombreuses unités de traitement identiques conservant les données dans des grilles en mémoire répliquées, et qui met à jour la base de données de façon asynchrone afin qu'elle soit hors du chemin des requêtes.

Architecture orientée services (SOA)
Approche, dominante dans les grandes entreprises à partir du début des années 2000, qui expose les capacités métier sous forme de services partagés dotés de contrats explicites, généralement reliés par un bus de services d'entreprise.

Architecture pilotée par les événements
Style dans lequel les composants communiquent en publiant des événements sur ce qui s'est produit et en réagissant aux événements publiés par d'autres, généralement par l'intermédiaire d'un courtier.

AsyncAPI
Spécification lisible par machine pour décrire des API pilotées par les messages : les canaux, les opérations qui y envoient ou y reçoivent des messages et les schémas de messages.

Backend en tant que service (BaaS)
Services tiers entièrement managés, comme l'authentification, les bases de données, le stockage et la messagerie, qui remplacent des composants côté serveur qu'une organisation devrait sinon construire et exploiter elle-même.

Backend for frontend (BFF)
Backend léger conçu pour un type de client, comme l'app mobile, qui appelle les services et met en forme leurs données pour ce client.

Base de données par service
Règle selon laquelle chaque service possède ses données et aucun autre service ne les lit ni ne les écrit directement ; les autres utilisent l'API du propriétaire ou ses événements.

Bus de services d'entreprise (ESB)
Middleware au centre de la SOA classique qui achemine, transforme et sert d'intermédiaire pour les messages entre services, et qui exécute souvent aussi des processus métier.

Capture des données modifiées (CDC)
Lecture du journal de modifications ou de réplication propre à une base de données pour diffuser en flux chaque insertion, mise à jour et suppression validée vers d'autres systèmes. Debezium est une plateforme CDC open source très utilisée.

Changement rétrocompatible
Modification d'une interface ou d'un schéma que les consommateurs existants peuvent supporter sans être modifiés, comme l'ajout d'un champ facultatif. Supprimer ou renommer un champ est un changement cassant.

Chorégraphie
Coordination d'un processus en plusieurs étapes par des participants qui réagissent aux événements des uns et des autres, sans qu'aucun composant unique ne dirige l'ensemble.

Clé d'idempotence
Valeur unique qu'un client envoie avec une requête non idempotente, comme POST, afin que le serveur puisse reconnaître une nouvelle tentative de la même opération et renvoyer le résultat d'origine au lieu d'effectuer le travail deux fois.

Cohérence à terme (eventual consistency)
Garantie que des copies de données qui ne sont plus mises à jour convergeront vers la même valeur au bout d'un certain temps, les lecteurs pouvant voir des valeurs périmées dans l'intervalle.

Cohérence causale
Modèle de cohérence dans lequel les opérations liées causalement (l'une a pu influencer l'autre) sont vues dans le même ordre par tous ; les opérations concurrentes sans lien peuvent être vues dans des ordres différents.

Cohérence séquentielle
Modèle de cohérence dans lequel tous les clients voient les opérations dans un ordre unique convenu qui préserve l'ordre propre à chaque client, mais cet ordre n'a pas à correspondre au temps réel.

Collision de données
Dans un cache en mémoire répliqué, situation où deux unités mettent à jour le même élément avant que la réplication ait propagé la première mise à jour, de sorte qu'une mise à jour écrase l'autre.

Commande
Message demandant à un gestionnaire précis de faire quelque chose, comme AuthorisePayment. Contrairement à un événement, elle peut être refusée.

Communication asynchrone
Interaction dans laquelle l'émetteur confie un message à un intermédiaire, comme un courtier de messages (message broker), et poursuit son travail sans attendre ; le destinataire le traite quand il est prêt.

Communication synchrone
Interaction dans laquelle l'appelant envoie une requête et attend la réponse avant de continuer, comme lors d'un appel téléphonique. Les deux parties doivent être disponibles au même moment.

Consommateur idempotent
Destinataire de messages qui enregistre quels messages il a traités, dans la même transaction que leurs effets, de sorte que les doublons ne changent rien.

Contrat piloté par le consommateur (consumer-driven contract)
Contrat constitué à partir de ce que chaque consommateur utilise réellement de l'API d'un fournisseur ; le fournisseur exécute les attentes de tous ses consommateurs dans son propre build pour savoir quelles modifications casseraient qui.

Contre-mesure de saga
Technique de conception qui réduit les anomalies dues au manque d'isolation des sagas, comme un verrou sémantique (un état PENDING), des mises à jour commutatives ou la relecture d'une valeur avant de la mettre à jour.

Contrôles hypermédias (HATEOAS)
Liens dans une réponse d'API qui indiquent au client quelles actions il peut effectuer ensuite, de sorte que le client suit des liens plutôt que de construire lui-même les URL. Niveau 3 du modèle de maturité de Richardson.

CQRS
Command query responsibility segregation (séparation des responsabilités de commande et de requête) : utilisation de modèles distincts, et souvent de stockages distincts, pour mettre à jour les données et pour les lire.

Découverte de services
Mécanisme, comme un registre ou le DNS, par lequel les clients trouvent les adresses réseau actuelles des instances de service en bonne santé.

Défaillance en cascade
Défaillance qui se propage : un composant lent ou défaillant fait attendre ses appelants, qui épuisent leurs propres ressources et défaillent à leur tour, et ainsi de suite en remontant la chaîne.

Défaillance partielle
Situation dans laquelle certaines parties d'un système distribué fonctionnent tandis que d'autres sont en défaillance, lentes ou injoignables, souvent sans que les parties qui fonctionnent puissent savoir lesquelles.

Démarrage à froid (cold start)
Délai supplémentaire lorsqu'une plateforme serverless doit créer et initialiser une nouvelle instance de fonction avant de traiter une requête, de quelques dizaines de millisecondes à plusieurs secondes.

Déployabilité indépendante
Capacité à modifier un service et à le mettre en production sans modifier, tester ou mettre en production un autre service en même temps.

Double écriture (dual write)
Mise à jour de deux systèmes, comme une base de données et un courtier de messages, par des opérations séparées ; une panne entre les deux les laisse incohérents.

Espace de tuples
Mémoire associative partagée dans laquelle des processus écrivent des tuples et dans laquelle ils lisent ou prélèvent les tuples correspondants, sans se connaître les uns les autres. Introduit par le langage Linda de David Gelernter (1985).

Événement
Enregistrement immuable d'un fait survenu, nommé au passé, comme OrderPlaced. Un nombre quelconque de destinataires peuvent y réagir.

Event sourcing
Stockage de chaque modification d'une entité sous forme d'événement, de sorte que le journal des événements soit la source de vérité et que l'état courant soit obtenu en le rejouant.

File (canal point à point)
Canal de messagerie dans lequel chaque message est remis à exactement un des consommateurs qui y lisent, de sorte que plusieurs instances peuvent se partager le travail.

Garanties de session
Promesses de cohérence propres à chaque client, décrites par Terry et ses collègues en 1994 : lecture de ses propres écritures (read-your-writes), lectures monotones, écritures monotones et écritures après lectures (writes-follow-reads).

GraphQL
Langage de requête et environnement d'exécution pour les API, publié par Facebook en 2015, dans lequel le client envoie une requête nommant exactement les champs qu'il veut d'un schéma typé et reçoit une réponse de même forme.

Grille de données en mémoire (in-memory data grid)
Logiciel qui conserve des données dans la mémoire de nombreuses machines sous la forme d'un seul stockage logique, avec partitionnement et réplication, pour un accès très rapide. Hazelcast, Apache Ignite et Oracle Coherence en sont des exemples.

Groupe de consommateurs
Ensemble de consommateurs qui se partagent le travail de lecture d'un topic : chaque partition est lue par un seul membre, et chaque groupe conserve ses propres offsets.

gRPC
Framework d'appel de procédure à distance, rendu open source par Google en 2015, qui envoie des messages Protocol Buffers sur HTTP/2 et génère du code client et serveur typé. Il prend en charge le streaming dans un sens, dans l'autre ou dans les deux.

Idempotence
Propriété d'une opération qui a le même effet qu'elle soit exécutée une ou plusieurs fois, ce qui rend les nouvelles tentatives sûres. Généralement obtenue grâce à un identifiant unique de requête ou d'événement que le destinataire mémorise.

Illusions de l'informatique distribuée
Huit hypothèses fausses sur les réseaux recensées chez Sun Microsystems par L Peter Deutsch et ses collègues, James Gosling ayant ajouté la huitième : par exemple, que le réseau est fiable et que la latence est nulle.

Last write wins (la dernière écriture l'emporte)
Règle de résolution des conflits qui conserve l'écriture portant l'horodatage le plus récent et écarte les autres. Simple, mais les mises à jour concurrentes sont perdues sans avertissement, et des écarts d'horloge peuvent désigner le mauvais gagnant.

Lecture de ses propres écritures (read-your-writes)
Garantie qu'après qu'un client a écrit une valeur, les lectures ultérieures de ce même client voient cette écriture ou une valeur plus récente, même si d'autres clients ne la voient pas encore.

Lectures monotones
Garantie qu'une fois qu'un client a vu une valeur, ses lectures ultérieures ne renvoient jamais une valeur plus ancienne, de sorte que les données ne semblent jamais remonter dans le temps.

Linéarisabilité
Le modèle de cohérence le plus fort pour un objet unique : chaque opération semble se produire en un instant situé entre son début et sa fin, dans un ordre partagé par tous les clients et qui respecte le temps réel, comme s'il n'existait qu'une seule copie.

Livraison au moins une fois (at-least-once)
Garantie que chaque message est traité, éventuellement plus d'une fois ; les doublons doivent être gérés par le destinataire.

Message
Toute donnée envoyée d'un programme à un autre par un canal ; elle peut porter un événement, une commande, une requête ou une réponse.

Microservices
Style dans lequel une application est construite sous forme de petits services, chacun dans son propre processus, organisé autour d'une capacité métier, déployable indépendamment et communiquant par des mécanismes légers.

Modèle de cohérence
Contrat qu'un stockage de données offre sur les valeurs que les lectures peuvent renvoyer compte tenu des écritures effectuées, et donc sur les résultats périmés ou réordonnés auxquels les clients doivent être préparés.

Modèle de données canonique
Représentation unique, à l'échelle de l'entreprise, de chaque entité métier, vers laquelle et depuis laquelle tous les messages sont traduits. Convient rarement bien à toutes les parties d'une entreprise.

Notification d’événement
Événement qui ne fait que signaler un changement, en ne transportant souvent qu’un identifiant, de sorte que les destinataires qui ont besoin de détails doivent interroger l’émetteur.

Offset (décalage)
La position séquentielle d’un événement au sein d’une partition ; les consommateurs enregistrent l’offset qu’ils ont atteint afin de pouvoir reprendre après un redémarrage.

OpenAPI
Format standard, lisible par machine, pour décrire les API HTTP (chemins, opérations, paramètres, schémas), anciennement la spécification Swagger. Il sert de base à la documentation, à la génération de code et aux tests.

Orchestrateur de saga
Composant qui coordonne une saga en envoyant des commandes aux participants, en suivant la progression dans une machine à états persistante et en déclenchant les compensations en cas de défaillance.

Orchestration
Coordination d’un processus en plusieurs étapes par un composant unique qui dit à chaque participant quoi faire et décide de la suite.

Outbox transactionnelle (transactional outbox)
Patron dans lequel un service écrit les messages sortants dans une table outbox au sein de la même transaction locale que sa modification métier ; un relais les publie ensuite, ce qui assure une livraison au moins une fois (at-least-once) sans transaction distribuée.

PACELC
L’extension de CAP proposée par Daniel Abadi : en cas de Partition, arbitrer entre disponibilité (Availability) et cohérence (Consistency) ; sinon (Else), en fonctionnement normal, arbitrer entre latence (Latency) et cohérence (Consistency).

Pagination par curseur
Parcours d’une collection page par page en transmettant un token opaque qui marque le dernier élément vu, plutôt qu’un offset numérique. Elle reste rapide et stable lorsque des éléments sont ajoutés pendant la pagination.

Partition (d’un topic)
Dans Kafka et les journaux similaires, une partie ordonnée d’un topic, en ajout seul (append-only). Les événements de même clé vont dans la même partition ; l’ordre n’est garanti qu’à l’intérieur de celle-ci.

Partition réseau
Faute réseau qui divise un système en groupes de nœuds incapables de communiquer entre eux, bien que chaque groupe puisse encore fonctionner et rester joignable par certains clients.

Problem details (RFC 9457)
Format JSON standard pour les erreurs des API HTTP, avec les membres type, title, status, detail et instance, servi en application/problem+json. La RFC 9457 (2023) a remplacé la RFC 7807.

Problème des requêtes N+1
Défaut de performance dans lequel le code effectue une requête pour récupérer une liste de N éléments, puis une requête supplémentaire par élément, soit N+1 allers-retours là où une ou deux suffiraient.

Protocol Buffers
Format binaire compact et indépendant du langage pour les données structurées, défini dans des fichiers .proto où chaque champ porte un numéro permanent. Utilisé par gRPC et de nombreux systèmes de messagerie.

Publication-abonnement (publish-subscribe)
Patron de messagerie dans lequel un éditeur envoie un message à un topic et chaque abonné en reçoit sa propre copie, sans que l'éditeur sache qui ils sont.

Quorum (R + W > N)
Dans un système de stockage à N répliques, une écriture attend W accusés de réception et une lecture R réponses ; si R + W > N, chaque ensemble de lecture recoupe chaque ensemble d'écriture sur au moins une réplique.

Saga
Transaction métier mise en œuvre sous la forme d'une suite de transactions locales, chacune associée à une transaction compensatoire, de sorte que soit toutes les étapes aboutissent, soit les étapes terminées sont annulées sur le plan sémantique. Introduite par Garcia-Molina et Salem en 1987.

Service de domaine
Dans une architecture à base de services, service déployé séparément qui couvre tout un domaine d'activité, comme les commandes ou les paiements, de l'API à l'accès aux données.

Service mesh (maillage de services)
Couche d'infrastructure faite de proxys réseau, configurés de manière centralisée, qui assure le chiffrement entre services, les nouvelles tentatives, les délais d'expiration, le contrôle du trafic et la télémétrie.

Système distribué
Ensemble de programmes s'exécutant sur des ordinateurs différents, qui coopèrent en s'envoyant des messages par un réseau afin que les utilisateurs voient un seul service.

Tests de contrat
Test de chaque côté d'une intégration de manière isolée, par rapport à une description partagée de leurs interactions, au lieu de déployer les deux ensemble pour des tests de bout en bout.

Théorème CAP
Résultat, conjecturé par Eric Brewer en 2000 et démontré par Gilbert et Lynch en 2002, selon lequel, pendant une partition réseau, un système répliqué ne peut pas être à la fois linéarisable et disponible pour chaque requête.

Topologie à courtier (broker)
Topologie pilotée par les événements sans coordinateur central : les processeurs réagissent aux événements et publient d'autres événements, de sorte que le flux de travail émerge de leurs réactions.

Topologie à médiateur
Topologie pilotée par les événements dans laquelle un médiateur reçoit un événement initial et dirige le flux de travail en envoyant des commandes à des processeurs dans un ordre défini.

Transaction compensatoire
Nouvelle transaction qui annule sémantiquement l'effet d'une transaction antérieure déjà validée, comme l'annulation d'une autorisation de carte. Contrairement à un retour arrière, l'original a pu déjà être vu.

Transfert d'état porté par les événements (event-carried state transfer)
Événements qui transportent suffisamment de données pour que les destinataires conservent leur propre copie, de sorte qu'ils n'ont pas besoin d'appeler l'émetteur.

Validation en deux phases (two-phase commit, 2PC)
Protocole qui fait valider ou annuler ensemble plusieurs participants : un coordinateur demande d'abord à tous de se préparer et de voter, puis leur communique à tous le résultat. Les participants qui ont voté oui restent bloqués si le coordinateur tombe en panne.

XA
L'interface standard X/Open (1991) entre un gestionnaire de transactions et des gestionnaires de ressources comme les bases de données et les courtiers de messages, utilisée pour exécuter une validation en deux phases entre eux.