BE SOFTWARE ARCHITECTURE WISE · CHAPITRE 9

Performances, scalabilité, résilience et exploitabilité

Ce chapitre transforme les scénarios de performances et de disponibilité du chapitre 2 en systèmes qui fonctionnent. Il traite de la scalabilité verticale et horizontale et du cube de scalabilité, de l'équilibrage de charge et des services sans état, des files d'attente, du back pressure et de la limitation de débit, des patrons de résilience qui empêchent les défaillances de se propager (délais d'expiration, nouvelles tentatives, disjoncteurs, cloisonnement), des SLI, des SLO et des budgets d'erreur, de l'observabilité avec les journaux, les métriques et les traces, de la planification de la capacité et des tests de performances, ainsi que de l'ingénierie du chaos et de l'apprentissage tiré des incidents. Il s'adresse aux développeurs et aux architectes qui connaissent les styles distribués du chapitre 6 et qui doivent garder un système rapide et opérationnel sous une charge réelle.

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

EN UN COUP D'ŒIL

Vous apprendrez à

- Choisir entre scalabilité verticale et horizontale, et entre les axes X, Y et Z du cube de scalabilité, et dimensionner un parc de serveurs pour un pic connu à l'aide de la loi de Little et d'une marge de capacité.
- Choisir des signaux et des réglages d'autoscaling qui évitent le décalage et les oscillations (flapping), et expliquer pourquoi le CPU est souvent la mauvaise métrique.
- Comparer l'équilibrage de charge de couche 4 et de couche 7 ainsi que les principaux algorithmes, concevoir les contrôles de santé et le drainage des connexions, et externaliser l'état de session.
- Appliquer les files d'attente, les tampons bornés, le back pressure, le délestage de charge et la limitation de débit (y compris le token bucket) pour garder la surcharge sous contrôle.
- Combiner délais d'expiration, échéances, nouvelles tentatives avec backoff et jitter, budgets de nouvelles tentatives, disjoncteurs, cloisonnement et solutions de repli pour contenir les défaillances.
- Définir des SLI et des SLO du point de vue de l'utilisateur, calculer les budgets d'erreur et les taux de consommation, et concevoir des alertes qui s'appuient sur eux.
- Instrumenter un système avec des journaux, des métriques et des traces à l'aide d'OpenTelemetry, et utiliser les méthodes RED et USE pour diagnostiquer les problèmes.
- Planifier la capacité et les tests de performances, mener des expériences de chaos en toute sécurité et animer des revues post-incident sans reproche.

Avant de commencer

Le chapitre 2, en particulier les sections sur les performances et la scalabilité et sur la disponibilité et la fiabilité ; le chapitre 6, en particulier « Pourquoi distribuer » et « Communication et API » ; et les sections du chapitre 7 sur la réplication, le partitionnement et la mise en cache.

Durée

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

Termes clés

cube de scalabilité · autoscaling · service sans état · équilibreur de charge de couche 7 · puissance de deux choix aléatoires · back pressure (contre-pression) · délestage de charge · token bucket (seau à jetons) · disjoncteur (circuit breaker) · cloisonnement (bulkhead) · budget d'erreur · taux de consommation (burn rate) · traçage distribué · ingénierie du chaos

Des qualités sur le papier aux qualités en production

Le chapitre 2 a rendu précises les performances et la disponibilité : percentiles de latence, débit, loi de Little, taux d'utilisation et « neuf », formulés sous forme de scénarios d'attribut de qualité testables. Ce chapitre porte sur la manière d'atteindre ces scénarios dans un système en fonctionnement et de les garder vrais le jour le plus chargé de l'année. Une conception qui

atteint ses objectifs sur un diagramme peut tout de même s'effondrer un vendredi à 12 h 30, quand tous les bureaux de la ville commandent leur déjeuner, ou à 10 h 00 le matin où les billets d'un concert sont mis en vente. La différence entre les deux tient rarement à du code astucieux. Elle tient à un ensemble de décisions architecturales : comment la capacité est ajoutée, comment la charge est répartie, comment le travail attend, comment les défaillances sont contenues, et comment les personnes qui exploitent le système savent ce qu'il fait.

Deux exemples récurrents traversent le chapitre. La **plateforme de livraison de repas** connaît chaque jour un pic marqué à midi et le soir, compte des dizaines de services et dépend de tiers comme les prestataires de paiement et les API de cartographie. La **mise en vente de billets** est le cas extrême : une demande qui passe de presque rien à plusieurs fois la capacité du système en quelques secondes, pour un stock fixe de places, sous les yeux de tout le pays. À eux deux, ils mettent en jeu toutes les techniques du chapitre.

Gérer la charge : passage à l'échelle, équilibrage et attente

Les quatre premières sections portent sur la charge. Stratégies de passage à l'échelle compare la scalabilité verticale et la scalabilité horizontale, utilise le cube de scalabilité d'Abbott et Fisher pour distinguer le clonage, le découpage par fonction et le découpage par données, et explique pourquoi l'autoscaling sur le CPU est souvent un mauvais choix et comment dimensionner un parc pour un pic connu avec la loi de Little et une marge de capacité. Équilibrage de charge, absence d'état et sessions traite de ce qui se trouve devant les instances : les équilibreurs de charge de couche 4 et de couche 7, les algorithmes qu'ils utilisent (dont la puissance de deux choix aléatoires), des contrôles de santé qui ne retirent pas tout un parc d'un coup, le drainage des connexions, et les raisons pour lesquelles les sessions persistantes doivent céder la place à un état de session externalisé.

Asynchronisme, files d'attente et back pressure traite de ce qui se passe quand le travail arrive plus vite qu'il ne peut être fait. Les files d'attente transforment les pics en arriérés, ce qui n'est utile que si l'arriéré est borné et se résorbe dans un délai acceptable. Le back pressure, le délestage de charge et la limitation de débit décident quel travail est accepté, lequel attend et lequel est refusé poliment avec un HTTP 429. Délais d'expiration, nouvelles tentatives, disjoncteurs, cloisonnement et dégradation gracieuse traite ensuite des défaillances entre services : délais d'expiration et échéances, nouvelles tentatives avec backoff et jitter et budgets qui empêchent les tempêtes de nouvelles tentatives, disjoncteurs, cloisonnement, solutions de repli, idempotence et requêtes de couverture. Son exemple commenté montre comment un seul délai d'expiration manquant permet à une seule dépendance lente de faire tomber tout ce qui l'appelle.

Exploiter le système : objectifs, observabilité, tests et apprentissage

La seconde moitié passe des mécanismes à l'exploitation. SLI, SLO, SLA et budgets d'erreur montre comment exprimer la fiabilité du point de vue de l'utilisateur, comment un budget d'erreur transforme un objectif de disponibilité en une quantité de défaillances autorisées que les équipes peuvent dépenser en changements, et comment les alertes sur le taux de consommation ne sollicitent les personnes d'astreinte que lorsque ce budget est réellement menacé. Observabilité traite des journaux, des métriques et des traces qui permettent aux ingénieurs de poser de nouvelles questions à un système en production, des méthodes RED et USE et des quatre signaux d'or, du traçage distribué avec OpenTelemetry, et du coût des données à forte cardinalité.

Planification de la capacité et tests de performances boucle la boucle entre prévisions et preuves : tests de charge, de stress, d'endurance, de pic et de point de rupture, modèles de charge ouverts et fermés, omission coordonnée, et un plan de capacité pour une mise en vente de billets. Ingénierie du chaos, incidents et apprentissage sans reproche traite des défaillances qui surviennent malgré tout : expérimenter délibérément en production pour trouver les faiblesses avant les clients, gérer les incidents avec des rôles clairs, et mener des revues sans reproche dont les actions correctives changent le système plutôt que les personnes.

Comment les sections s'articulent

Les sections forment une chaîne. Le passage à l'échelle et l'équilibrage de charge ajoutent de la capacité ; les files d'attente et le back pressure décident de ce qui se passe quand la capacité vient malgré tout à manquer ; les patrons de résilience empêchent un problème local de devenir global. Les SLO disent ce qui est assez bon, l'observabilité montre si le système atteint ce niveau et, sinon, pourquoi, les tests et la planification de la capacité prédisent quand il cessera de l'atteindre, et l'ingénierie du chaos et les revues d'incident trouvent les lacunes de tout ce qui précède. La plupart de ces mécanismes coûtent peu à ajouter lors de la conception et cher à ajouter après coup ; c'est pourquoi ils relèvent de l'architecture plutôt que du détail d'exploitation : qu'un service soit sans état, qu'un appel soit idempotent, qu'une file soit bornée et qu'une requête porte une échéance, tout cela se décide bien avant le premier incident.

Le chapitre suppose connus les styles distribués du chapitre 6 et l'architecture des données du chapitre 7. Il y renvoie pour les clés d'idempotence, les garanties de livraison, la réplication et la mise en cache plutôt que de les répéter, et renvoie au chapitre 10 pour les plateformes cloud, les fonctionnalités de Kubernetes et les conceptions multirégions sur lesquelles ces systèmes tournent généralement.

L’idée à retenir

Toute ressource est finie, donc tout système a un point de rupture ; le travail de l’architecte est de décider à l’avance de ce qui se passe quand il est atteint. Un système bien conçu soumis à une charge excessive ralentit en douceur, se déleste d’abord du travail le moins important, refuse clairement plutôt que d’expirer en silence, isole la partie défaillante et se rétablit de lui-même quand la pression retombe. Un système mal conçu met en file sans limite, réessaie jusqu’à s’effondrer et entraîne ses voisins dans sa chute. La différence tient à de petits choix délibérés (un délai d’expiration ici, une file bornée là, un budget de nouvelles tentatives, une sonde de disponibilité) et à l’habitude de mesurer, de tester et d’apprendre du système tel qu’il fonctionne réellement.

Stratégies de passage à l'échelle : verticale, horizontale et cube de scalabilité

AVANT DE LIRE

Cette section s'appuie sur :

- 0.7 Serveurs et cloud
- 2.4 Performances et scalabilité
- 6.1 Pourquoi distribuer

Termes déjà vus qu'elle utilise : Équilibreur de charge (load balancer), Loi de Little, Loi d'Amdahl, Taux d'utilisation, Réplique en lecture (read replica)

L'IDÉE EN BREF

Pensez à un café très fréquenté. Vous pouvez lui permettre de servir plus de clients en achetant une machine à café plus grande et plus rapide (scalabilité verticale), ou en ouvrant davantage de comptoirs avec davantage de personnel (scalabilité horizontale). La scalabilité horizontale va plus loin, mais ne fonctionne que si n'importe quel comptoir peut servir n'importe quel client : aucune commande ne doit donc rester dans la tête d'un seul barista. Vous pouvez aussi répartir le travail, avec un comptoir pour les boissons et un pour la nourriture, ou un café par quartier. Et comme le personnel supplémentaire met du temps à arriver, un gérant avisé le planifie avant le coup de feu de midi plutôt que de l'appeler quand la file déborde déjà sur le trottoir.

CE QU'IL FAUT SAVOIR

- La scalabilité verticale ajoute des ressources à une seule machine sans modifier le logiciel, mais se heurte à un plafond de taille et de prix et laisse un seul domaine de défaillance ; la scalabilité horizontale ajoute des instances, va plus loin et tolère les défaillances, mais exige un travail sans état et divisible.
- Le cube de scalabilité (Abbott et Fisher) distingue trois façons de passer à l'échelle horizontalement : X clone des copies identiques, Y découpe par fonction en services, et Z découpe par données pour que chaque copie serve une tranche des clients ou des enregistrements.
- L'absence d'état est la condition préalable à la scalabilité sur l'axe X : les sessions, les fichiers, les caches servant de source de vérité, les tâches planifiées, les verrous et les compteurs doivent passer dans des stockages partagés ou voyager avec la requête.

- L'autoscaling peut être réactif, planifié ou prédictif ; la scalabilité réactive suit la charge avec plusieurs minutes de retard, donc les changements brusques exigent une marge de capacité, des files d'attente ou une capacité planifiée à l'avance.
- Passez à l'échelle sur un signal lié à la ressource qui vient à manquer, généralement les requêtes en cours par instance ou l'arriéré de la file par worker ; le CPU est trompeur pour les services limités par les E/S, et la latence l'est quand une dépendance partagée est lente.
- Évitez le flapping en augmentant vite et en réduisant lentement, avec des fenêtres de stabilisation, des bandes de tolérance et des nombres minimal et maximal d'instances, et dimensionnez les parcs pour le pic avec une zone perdue, à l'aide de la loi de Little.
- Faire passer le niveau applicatif à l'échelle reporte la pression sur le niveau des données ; vérifiez les connexions et les lignes chaudes, et recourez à l'optimisation des requêtes, à la mise en cache, aux répliques, aux poolers de connexions et à la scalabilité verticale avant le partitionnement.

Ce que signifie passer à l'échelle en pratique

La section 2.4 a défini la scalabilité comme la capacité à absorber davantage de charge en ajoutant des ressources, et a montré avec la loi d'Amdahl et la loi universelle de scalabilité pourquoi ajouter des ressources n'ajoute pas toujours de la capacité. Cette section porte sur les choix pratiques : *quelles* ressources ajouter, *où*, *quand* et *avec quel degré d'automatisation*. Le point de départ est toujours la même question : quel est le paramètre de charge, et quelle ressource vient à manquer en premier ? À midi, l'API de commandes d'une plateforme de livraison de repas est limitée par les requêtes par seconde ; son service d'affectation, par le nombre de livreurs actifs dont les positions doivent être rapprochées toutes les quelques secondes ; sa base de données, par les transactions d'écriture par seconde et par les connexions. Chacun appelle une réponse différente.

Faire passer un système à l'échelle, c'est en réalité faire passer à l'échelle son goulot d'étranglement actuel, puis trouver le suivant. Doubler le niveau web d'un système dont la base de données est occupée à 95 % n'apporte rien d'autre qu'une facture plus lourde et une base de données encore plus sollicitée. Toute décision de passage à l'échelle commence donc par une mesure : taux d'utilisation et saturation de chaque ressource (la méthode USE de Brendan Gregg, présentée dans Observabilité), et un test de charge qui trouve à partir de quand la latence commence à grimper (Planification de la capacité et tests de performances).

Scalabilité verticale : une machine plus grosse

La scalabilité verticale (scaling up) consiste à donner à la machine existante plus de cœurs de CPU, plus de mémoire, des disques plus rapides ou un réseau plus rapide. Sa grande vertu est que rien ne change dans le logiciel. Une base de données relationnelle unique sur un serveur plus grand conserve ses transactions, ses jointures et sa cohérence simple ; une application conserve ses caches en mémoire et ses fichiers locaux. Dans le cloud, le redimensionnement est un appel d'API suivi d'un redémarrage. Les plus grandes machines virtuelles du cloud offrent désormais des centaines de CPU virtuels et des téraoctets de mémoire, bien plus que ce dont la plupart des systèmes de gestion auront jamais besoin, et certains sites web très fréquentés ont décrit publiquement comment ils servent un trafic énorme à partir d'un petit nombre de serveurs de bases de données puissants.

Les limites sont pourtant réelles. Il existe une machine plus grande que toutes les autres, et le prix par unité de capacité augmente fortement en haut de la gamme. Une machine constitue un seul domaine de défaillance : quand elle tombe en panne, ou qu'elle est redémarrée pour être redimensionnée ou corrigée, tout ce qu'elle héberge s'arrête ; la scalabilité verticale seule n'apporte donc rien à la disponibilité. Le logiciel ne peut souvent pas exploiter pleinement une machine plus grosse : un processus monothread, un verrou global ou une ligne chaude de la base de données limitent l'accélération exactement comme le prédit la loi d'Amdahl. Et la capacité vient par grandes marches lentes, si bien que vous payez le pic jour et nuit. La scalabilité verticale reste néanmoins le bon premier geste bien plus souvent que la mode ne le laisse croire, en particulier pour les bases de données, où la scalabilité horizontale coûte cher en effort de conception.

Scalabilité horizontale : plus de machines

La scalabilité horizontale (scaling out) consiste à exécuter davantage d'instances et à répartir le travail entre elles, généralement avec un équilibreur de charge en amont. Elle va beaucoup plus loin, par petites marches peu coûteuses, et la redondance qui ajoute de la capacité permet aussi de tolérer les défaillances : avec huit instances, en perdre une coûte un huitième de la capacité plutôt que sa totalité. Les plateformes cloud permettent d'ajouter une instance en quelques secondes ou minutes, ce qui est le fondement de l'autoscaling.

Le prix à payer est que le logiciel doit être conçu pour cela. N'importe quelle instance doit pouvoir traiter n'importe quelle requête, ce qui signifie que les instances ne doivent rien conserver dont dépendrait une requête ultérieure (voir la section suivante). Le travail doit être divisible : une tâche qui exige un verrou global unique ne s'exécute pas plus vite sur dix machines. Les données doivent être accessibles depuis chaque instance, ce qui reporte la pression sur le niveau de données partagé. Et le système devient distribué, avec les défaillances partielles, les délais d'expiration et les nouvelles tentatives que décrit la section

6.1. La plupart des systèmes réels combinent les deux : les niveaux applicatifs sans état passent à l'échelle horizontalement, tandis que les bases de données passent à l'échelle verticalement aussi longtemps que c'est raisonnable, puis horizontalement avec des répliques et des partitions.

Scalabilité verticale et horizontale comparées

Préoccupation	Verticale (scaling up)	Horizontale (scaling out)
Modifications du logiciel	Aucune	Instances sans état et travail divisible
Limite supérieure	La plus grande machine disponible	Généralement le niveau de données partagé ou les coûts de coordination
Courbe des coûts	Augmente fortement pour les plus grandes tailles	À peu près linéaire, par petites marches
Défaillance (failure)	Un seul domaine de défaillance ; le redimensionnement exige un redémarrage	Perdre une instance fait perdre une fraction de la capacité
Vitesse de changement	Quelques minutes plus un redémarrage	De quelques secondes à quelques minutes, peut être automatique
Cohérence et transactions	Inchangées et simples	Inchangées pour les niveaux sans état ; difficiles pour les données
Usage typique	Bases de données, logiciels hérités et avec état, première réponse à la croissance	Niveaux web et API, workers, tout ce qui est sans état

Le cube de scalabilité : trois façons différentes de passer à l'échelle horizontalement

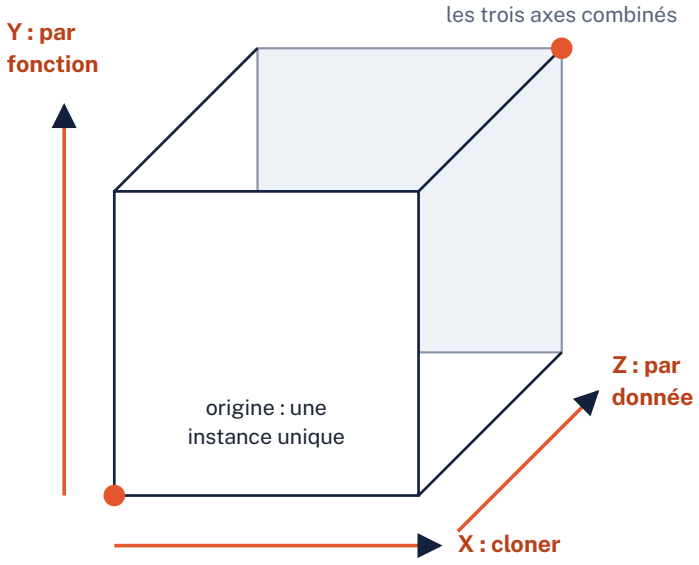
Martin Abbott et Michael Fisher, qui ont dirigé la technologie chez eBay et PayPal, décrivent la scalabilité horizontale à l'aide du cube de scalabilité dans *The Art of Scalability* (première édition 2009, deuxième édition 2015). Il distingue trois façons indépendantes de découper un système, chacune représentée par un axe d'un cube dont l'origine est une instance unique qui fait tout.

L'**axe X** est la duplication horizontale, ou clonage : exécuter N copies identiques de toute l'application ou du service et répartir les requêtes entre elles. C'est l'axe le plus simple et le moins coûteux, et il convient bien au traitement de requêtes sans état et au trafic de lecture. Il

n’apporte rien pour la taille des données ni pour le volume d’écriture, car chaque clone parle toujours aux mêmes données, et rien non plus pour la taille de la base de code ou le nombre d’équipes qui y travaillent.

L’**axe Y** découpe le système par fonction : des services différents pour les cartes, les commandes, l’affectation et les paiements, chacun n’exécutant que le code et les données de sa capacité. Chaque service peut alors passer à l’échelle, tomber en panne et être déployé séparément, et chacun est plus petit et plus facile à comprendre. C’est l’axe qui sous-tend les microservices (section 6.3), avec tous leurs coûts. L’**axe Z** découpe par données : chaque copie exécute le même code, mais chacune n’est responsable que d’une tranche des clients, des locataires, des régions ou des enregistrements, choisie selon une clé comme la ville ou l’identifiant client. Le sharding de base de données est l’axe Z appliqué au stockage (section 7.4) ; une architecture *cellulaire* l’applique à des piles entières, de sorte qu’une faute ou un mauvais déploiement dans une cellule n’affecte que les clients de cette cellule.

Le cube de scalabilité



Axe X : cloner
N copies identiques derrière un équilibreur, chacune fait tout : 14 copies de l’API de commandes

Axe Y : par fonction
Services par capacité : cartes, commandes, paiements et affectation

Axe Z : par données
Même code, chaque copie gère une part des clients ou données : cellule ou shard par ville

Le cube de scalabilité d’Abbott et Fisher : chaque axe est une façon différente de découper un système, et les systèmes réels combinent généralement les trois.

Les trois axes dans la plateforme de livraison de repas

Axe	Ce qui est découpé	Exemple	Résout	Coûts et limites
X : cloner	Les requêtes, entre des copies identiques	14 instances de l'API de commandes derrière un seul équilibreur de charge	Débit de requêtes, défaillances d'instances	Exige l'absence d'état ; tous les clones partagent une seule base de données
Y : fonction	Code et données, par capacité	Services distincts pour les cartes, les commandes, l'affectation et les paiements	Besoins différents en passage à l'échelle et en défaillances ; taille des équipes	Appels réseau, données distribuées, charge d'exploitation
Z : données	Clients ou enregistrements, par clé	Une cellule par groupe de villes, ou commandes partitionnées par ville	Taille des données, volume d'écriture, rayon d'impact	Couche de routage, requêtes entre tranches, rééquilibrage, clés chaudes

Les axes se combinent. Le service de commandes de la plateforme (un découpage sur l'axe Y) s'exécute sous forme de nombreux clones (X) dans chaque cellule régionale (Z). Le conseil pratique d'Abbott et Fisher est d'utiliser l'axe le moins coûteux qui résout le problème réel : X pour le volume de requêtes, Y quand des parties du système ont des besoins réellement différents en matière de passage à l'échelle, de défaillances ou d'équipes, et Z quand un jeu de données ou un rayon d'impact est devenu trop grand. Choisir Z parce que cela fait impression apporte le routage, le rééquilibrage et les requêtes entre partitions bien avant qu'on en ait besoin.

L'absence d'état : la condition préalable au clonage

La scalabilité sur l'axe X ne fonctionne que si n'importe quel clone peut servir n'importe quelle requête. Un service sans état ne conserve rien entre les requêtes dans sa propre mémoire ni sur son propre disque ; tout ce dont une requête ultérieure a besoin se trouve dans un stockage partagé ou voyage avec la requête. L'état qui se cache habituellement dans une instance est facile à recenser dès qu'on le cherche : sessions utilisateur conservées en mémoire, fichiers

envoyés écrits sur le disque local, caches en mémoire traités comme la source de vérité, tâches planifiées qui doivent s'exécuter sur exactement une machine, verrous internes au processus protégeant une ressource partagée, et compteurs de limitation de débit.

Chacun a un remède classique : les sessions vont dans un stockage partagé ou dans un jeton signé (section 9.2) ; les fichiers vont dans un stockage objet ; les caches deviennent une optimisation de performances qui peut être perdue à tout moment, ou passent dans un cache partagé (section 7.6) ; les tâches planifiées utilisent un ordonnanceur avec élection de leader ou une file d'attente ; les verrous passent dans la base de données ou dans un service de coordination ; les compteurs de limitation de débit passent dans un stockage partagé. La règle de l'application douze facteurs selon laquelle les processus sont sans état et ne partagent rien (section 10.3) exprime cette même idée. L'absence d'état coûte peu à intégrer dès la conception et cher à ajouter après coup, car l'état s'infiltre dans le code en de nombreux petits endroits.

Autoscaling : réactif, planifié et prédictif

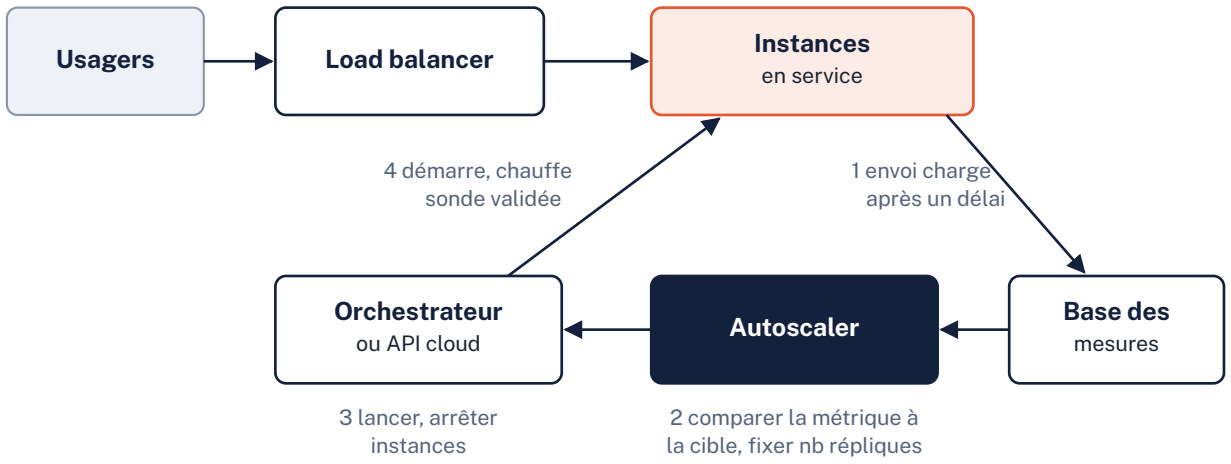
L'autoscaling ferme une boucle de rétroaction : mesurer la charge, la comparer à une cible, modifier le nombre d'instances. L'autoscaling **réactif** le fait en continu à partir d'une métrique. Le suivi de cible (target tracking) en est la forme courante : maintenir le CPU moyen à 60 %, ou les requêtes en cours par instance à 30. Le Horizontal Pod Autoscaler de Kubernetes calcule `desired = ceil(current replicas × current metric ÷ target)` : six pods à 45 requêtes en cours en moyenne pour une cible de 30 deviennent ceil(6 × 45 ÷ 30) = 9. Par défaut, il réévalue la situation toutes les 15 secondes et ignore les écarts de moins de 10 % par rapport à la cible.

L'autoscaling **planifié** modifie la capacité à des heures connues : relever le minimum à 18 instances à 11 h 15 chaque jour de semaine avant le pic de midi, le rabaisser à 14 h 00. C'est rudimentaire mais très efficace pour des schémas prévisibles, et c'est le seul moyen d'avoir la capacité en place *avant* un changement brusque comme une mise en vente de billets à 10 h 00 précises. L'autoscaling **prédictif**, proposé par les principaux fournisseurs de cloud, prévoit la charge à partir de l'historique (cycles quotidiens et hebdomadaires) et passe à l'échelle en avance, avec l'autoscaling réactif comme filet de sécurité pour les surprises.

La propriété cruciale de tout schéma réactif est le décalage. Les métriques sont collectées et moyennées sur des dizaines de secondes ; l'autoscaler agit lors de sa prochaine évaluation ; une nouvelle machine virtuelle met une minute ou plus à démarrer, une image de conteneur doit être téléchargée, un service JVM ou Python doit démarrer, chauffer ses caches et ses pools de connexions et passer sa sonde de disponibilité avant que l'équilibreur de charge ne lui envoie du trafic. Dans un cluster, il faut parfois d'abord ajouter les nœuds eux-mêmes, au moyen d'un second autoscaler, plus lent. Entre le moment où la charge augmente et celui où la nouvelle capacité devient utile, il s'écoule couramment de deux à dix minutes. L'autoscaling

réactif gère donc les changements progressifs ; les changements soudains doivent être absorbés par une marge de capacité, par des files d'attente (section 9.3) ou par une capacité planifiée à l'avance.

La boucle de rétroaction de l'autoscaling et l'origine du décalage



Chaque étape de la boucle ajoute du délai : l'autoscaling réactif suit donc la charge avec plusieurs minutes de retard ; la marge de capacité et la capacité planifiée comblent l'écart.

Sur quelle métrique passer à l'échelle, et pourquoi le CPU est souvent un mauvais choix

Le taux d'utilisation du CPU est le signal par défaut de la plupart des plateformes parce qu'il est toujours disponible. Ce n'est le bon signal que lorsque le service est limité par le CPU : redimensionnement d'images, chiffrement, sérialisation lourde. La plupart des services métier ne le sont pas. Une API de commandes passe l'essentiel de chaque requête à attendre la base de données, le prestataire de paiement et d'autres services ; son CPU peut rester à 25 % alors que tous les threads de travail sont occupés à attendre et que les nouvelles requêtes s'accumulent. Passer à l'échelle sur le CPU ne déclenche alors rien avant qu'il ne soit bien trop tard. La défaillance inverse se produit aussi : une tempête de ramasse-miettes ou un préchauffage JIT fait monter le CPU sans aucune hausse de la charge, et l'autoscaler ajoute des instances inutiles.

Un meilleur signal est un signal directement lié à la ressource qui vient à manquer. Pour les services requête-réponse, il s'agit généralement de la **concurrence** : le nombre de requêtes en cours par instance, comparé au nombre de threads de travail ou de connexions dont elle dispose. La loi de Little rend ce lien exact (en cours = taux d'arrivée × temps de réponse), et ce signal réagit à la fois à un trafic accru et à des réponses plus lentes. Pour les workers de file

d'attente, c'est l'**arriéré par worker** ou l'âge du plus ancien message, qui exprime le temps que la file mettra à se résorber. La latence est tentante mais dangereuse comme signal de passage à l'échelle : elle augmente pour des raisons que davantage d'instances ne peuvent pas corriger, comme une base de données lente, et ajouter des instances à un service dont une dépendance est en difficulté envoie encore plus de charge à cette dépendance. Utilisez la latence pour les alertes, et passez à l'échelle sur une mesure du travail.

Les signaux de passage à l'échelle et quand ils trompent

Signal	Convient à	Trompe quand
Taux d'utilisation du CPU	Travail limité par le CPU : traitement d'images, chiffrement, sérialisation lourde	Le travail attend des E/S, des threads ou des connexions ; le ramasse-miettes ou le préchauffage gonflent le CPU
Requêtes par seconde par instance	Services dont les requêtes coûtent à peu près la même chose	La répartition des requêtes change, par exemple plus de recherches et moins de lectures de la carte
Requêtes en cours par instance	La plupart des services requête-réponse ; correspond aux limites de threads et de connexions	Une dépendance en aval ralentit ; ajouter des instances lui ajoute alors de la charge, donc plafonnez le maximum
Arriéré de la file par worker, ou âge du plus ancien message	Workers asynchrones	Les workers sont limités par une limite de débit en aval plutôt que par leur nombre
Latence	Alertes et SLO (section 9.5)	La cause est une dépendance partagée que davantage d'instances ne peuvent pas corriger
Métrique métier (commandes par minute, sessions actives)	Charge ayant une relation stable et mesurée avec la capacité	La relation dérive à mesure que le produit évolue ; recalibrez avec des tests de charge

SI VOUS DÉBUTEZ

Pas à pas : threads occupés, processeur inactif

Le point qui fait le plus souvent trébucher ici est qu'un serveur peut être plein alors que son processeur semble presque inactif ; le taux d'utilisation du CPU peut donc être le mauvais signal pour passer à l'échelle.

- Une instance de l'API d'historique des commandes dispose de 48 threads de travail : elle peut donc traiter au plus 48 requêtes à la fois. Chaque requête passe l'essentiel de son temps à attendre : les requêtes à la base de données et les réponses du service Carte.
- Un thread qui attend n'utilise presque pas de temps processeur, mais il reste occupé : il ne peut pas commencer une autre requête avant que la réponse revienne.
- Chaque requête occupe un thread pendant environ 0,14 seconde. D'après la loi de Little, 250 requêtes par seconde occupent environ 35 threads (250 × 0,14). À 450 par seconde, il en faudrait 63, plus que les 48 disponibles : les requêtes s'accumulent alors et les temps de réponse grimpent.
- Pourtant, le processeur peut rester à environ 28 %. Un autoscaler chargé de maintenir le CPU à 65 % ne voit aucun problème et n'ajoute aucune instance pendant que les clients attendent.
- Compter les requêtes en cours par instance montre directement le problème : le compte grimpe vers 48. Passer à l'échelle sur ce compte, avec une cible d'environ 36 (les trois quarts des threads), commence à ajouter des instances alors que certains threads sont encore libres.
- Le CPU reste le bon signal pour un travail qui est réellement du calcul, comme le redimensionnement d'images. Et le nombre de requêtes en cours grimpe aussi quand la base de données elle-même est en difficulté ; davantage d'instances ne font alors que lui envoyer plus de charge, et c'est pourquoi la section fixe aussi un maximum.
- **La leçon :** passez à l'échelle sur la ressource qui vient réellement à manquer. Pour la plupart des services métier, ce sont les threads ou les connexions : mesurez donc le travail en cours, pas l'effort que fournit le processeur.

Cooldowns, stabilisation et flapping

Un autoscaler qui réagit à chaque soubresaut va osciller (flapping) : ajouter des instances quand une rafale arrive, les retirer une minute plus tard quand elle est passée, puis les rajouter, en payant chaque fois le coût du démarrage et en perturbant les caches et les pools de connexions. Il peut aussi courir après sa propre queue : les nouvelles instances sont froides et lentes, donc la latence ou le CPU par requête augmente brièvement après l'ajout d'instances, ce qui ressemble à davantage de charge.

Les défenses classiques sont asymétriques. Ajoutez des instances rapidement, car un manque de capacité pénalise les utilisateurs ; retirez-les lentement, car un excès de capacité ne coûte que de l'argent. Un cooldown, ou fenêtre de stabilisation, fait attendre l'autoscaler après une action, ou lui fait retenir la recommandation la plus élevée sur une fenêtre récente avant de réduire : par défaut, l'autoscaler de Kubernetes utilise une fenêtre de cinq minutes (300 secondes) pour la réduction et aucune pour l'augmentation. Une bande de tolérance ignore les petits écarts. Des nombres minimal et maximal d'instances bornent la boucle : le minimum conserve assez de capacité pour survivre à la perte d'une zone et aux premières minutes d'une hausse soudaine, et le maximum protège la base de données, les services en aval et le budget contre un autoscaler qui prendrait un problème pour de la charge. La réduction doit aussi se faire en douceur : une instance choisie pour être retirée doit cesser de recevoir de nouvelles requêtes et terminer celles en cours (drainage des connexions, section 9.2).

EXEMPLE COMMENTÉ

Exemple commenté : dimensionner l'API de commandes pour le pic de midi

L'API de commandes de la plateforme de livraison de repas doit absorber le déjeuner en semaine. Au cours des huit dernières semaines, la minute la plus chargée a atteint 2050 requêtes par seconde (rps) vers 12 h 30 ; le marketing prévoit une croissance de 15 % ce trimestre. Un test de charge sur une instance (4 vCPU, 32 threads de travail) a mesuré un temps de réponse moyen de 120 ms et une latence qui commence à grimper vers 260 rps. Le service tourne dans trois zones de disponibilité et doit survivre à la perte d'une zone au pic.

Pic prévu λ	2050 × 1,15 ≈ 2360, arrondi à 2400 rps
Loi de Little pour le parc	L = λ × W = 2400 × 0,12 s = 288 requêtes en cours au pic
Capacité d'une instance	32 threads ÷ 0,12 s ≈ 267 rps, ce qui concorde avec le coude trouvé par le test de charge (environ 260 rps)
Taux d'utilisation cible	70 % de 32 threads = 22,4 requêtes en cours par instance (environ 187 rps), ce qui laisse 30 % de marge pour les rafales
Instances pour le pic	288 ÷ 22,4 = 12,9, donc 13 instances (2400 ÷ 13 ≈ 185 rps chacune)
Survivre à la perte d'une zone	13 instances réparties en 5/4/4 : en perdre 5 laisse 8 instances avec 288 ÷ 8 = 36 requêtes en cours chacune, plus que 32 threads : surcharge. Autoriser jusqu'à 85 % (27,2 en cours) pendant la perte d'une zone exige 288 ÷ 27,2 = 10,6, donc 11 dans deux zones : 6 par zone, 18 au total
Montée en charge et décalage de l'ajout d'instances	La demande passe de 600 rps à 11 h 30 à 2400 à 12 h 15 : 40 rps par minute. L'ajout réactif d'instances prend environ 5 minutes (1 minute pour détecter, 4 pour démarrer et chauffer), pendant lesquelles la demande augmente de 200 rps ; la capacité disponible à 70 % est d'environ 80 rps par instance, donc l'autoscaling réactif seul pourrait suivre cette montée progressive ; ce qu'il ne peut pas apporter, c'est une protection contre la perte d'une zone
Décision	L'autoscaling planifié relève le minimum à 18 (6 par zone) à 11 h 15 et le ramène à 6 (2 par zone) à 14 h 00 ; l'autoscaling réactif sur les requêtes en cours par instance (cible 22) reste en place comme filet de sécurité, avec un maximum de 30
Vérification de la base de données	18 instances × un pool de 10 connexions = 180 ; au maximum de 30 instances, 300. La base de données en autorise 200 : l'équipe ajoute donc un pooler de connexions avant de relever le maximum

La loi de Little transforme un débit prévu et un temps de réponse mesuré en un chiffre de concurrence que l'on peut comparer directement aux limites de threads et de connexions.

C'est l'exigence de survie à la perte d'une zone, et non la charge moyenne, qui fixe la taille du parc au pic : 18 instances au lieu de 13. L'autoscaling réactif ne peut pas apporter cette protection, car remplacer une zone perdue prend plusieurs minutes.

L'ajout d'instances au niveau applicatif a multiplié la charge sur les connexions de la base de données partagée, qui est le prochain goulot d'étranglement. Tout exercice de dimensionnement doit se terminer par une vérification du niveau inférieur.

Faire passer le niveau des données à l'échelle

Les niveaux applicatifs sans état sont faciles à faire passer à l'échelle ; c'est tout l'intérêt de les rendre sans état. La pression retombe alors sur les bases de données, les caches et les courtiers qu'ils partagent, et c'est pourquoi le niveau des données est généralement la véritable limite d'un système. Les outils ont été présentés au chapitre 7 et sont résumés ici dans l'ordre où les professionnels y ont généralement recours. Premièrement, réduire le travail : index, meilleures requêtes, moins d'allers-retours par requête. Deuxièmement, absorber les lectures avec la mise en cache (section 7.6) et des répliques en lecture, en acceptant le retard de réplication là où le cas d'usage le permet. Troisièmement, protéger les connexions : chaque instance applicative détient un pool, donc le total croît avec le parc, et un pooler de connexions comme PgBouncer permet à des milliers de connexions clientes de partager quelques centaines de connexions à la base (la valeur par défaut de `max_connections` dans PostgreSQL n'est que de 100). Quatrièmement, faire passer le serveur primaire à l'échelle verticalement, ce qui est souvent moins coûteux que toute refonte. Seulement ensuite, partitionner les écritures selon l'axe Z par une clé comme la ville ou le client (section 7.4), ou passer à une base de données SQL distribuée.

Deux mises en garde s'imposent. Premièrement, l'autoscaling du niveau applicatif peut nuire au niveau des données : chaque nouvelle instance ouvre des connexions et chauffe son cache à partir de la base de données, si bien qu'un ajout massif d'instances peut à lui seul surcharger la base. Deuxièmement, certaines limites sont logiques et non physiques : une seule ligne chaude (le stock des 500 derniers billets d'un concert, ou un numéro de séquence global) sérialise les écritures quelle que soit la taille du serveur. Ces limites se suppriment par la conception, par exemple en répartissant le stock sur plusieurs compteurs ou en réservant les places par lots, et non par le matériel.

GUIDE DE DÉCISION

Guide de décision : comment ajouter de la capacité à un composant

Parcourez les questions dans l'ordre pour le composant qui limite le système aujourd'hui.

Q1 **Le goulot d'étranglement a-t-il été mesuré (taux d'utilisation, saturation et mise en file d'attente pour le CPU, la mémoire, les connexions, les verrous et les appels en aval) plutôt que deviné ?**

- **Oui** Continuez avec le composant qui sature en premier.
- **Non** Faites d'abord un test de charge et mesurez. Faire passer à l'échelle le mauvais niveau ajoute du coût sans ajouter de capacité, et peut reporter davantage de charge sur le véritable goulot d'étranglement.

Q2 **Le travail lui-même peut-il être rendu moins coûteux (un index, un cache, moins d'allers-retours, une charge utile plus petite, moins de journalisation) ?**

- **Oui** Faites-le d'abord. Cela réduit les coûts à toutes les échelles et supprime souvent tout besoin de passer à l'échelle.
- **Non** Continuez : il faut ajouter de la capacité.

Q3 **Le composant est-il sans état, ou son état (sessions, fichiers, verrous en mémoire, planifications) peut-il être déplacé vers des stockages partagés ?**

- **Oui** Passez à l'échelle horizontalement sur l'axe X derrière un équilibreur de charge, avec un autoscaling sur un signal de charge et une marge suffisante pour couvrir le décalage de l'ajout d'instances.
- **Non** Passez à l'échelle verticalement pour l'instant, et planifiez le travail d'externalisation de l'état ; une instance unique avec état est aussi un point de défaillance unique.

Q4 **La limite se trouve-t-elle dans le niveau des données : débit d'écriture, taille de stockage ou nombre de connexions d'une base de données ?**

- **Oui** Utilisez la mise en cache et des répliques en lecture pour les lectures, un pooler de connexions pour les connexions, et le partitionnement sur l'axe Z (shards ou cellules) pour les écritures et le stockage.
- **Non** Continuez.

Q5 **Certaines parties du composant ont-elles des besoins très différents en matière de charge, de passage à l'échelle ou de défaillances ?**

Oui Envisagez un découpage sur l'axe Y pour que chaque partie passe à l'échelle et tombe en panne séparément, en acceptant les coûts de la distribution décrits au chapitre 6.

Non Restez sur le clonage selon l'axe X : c'est l'axe le moins coûteux à exploiter.

ATTENTION

Erreurs courantes

- Faire passer à l'échelle le niveau facile à faire évoluer plutôt que celui qui est saturé, typiquement en ajoutant des serveurs web devant une base de données qui est déjà le goulot d'étranglement.
- Appliquer l'autoscaling sur le CPU à des services limités par les E/S, si bien que la capacité n'arrive qu'après l'effondrement de la latence.
- Compter sur l'autoscaling réactif pour des changements brusques comme les mises en vente, les diffusions ou les campagnes marketing, qui arrivent plus vite que les instances ne peuvent démarrer.
- Ne fixer aucun nombre maximal d'instances, si bien qu'une dépendance lente est prise pour de la charge et que l'autoscaler multiplie les connexions vers elle.
- Dimensionner pour une journée moyenne plutôt que pour le pic avec une zone ou une instance perdue.
- Supposer que les instances sont sans état sans vérifier l'absence de sessions en mémoire, de fichiers locaux, de tâches planifiées sur une seule instance et de verrous internes au processus.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Le CPU d'une API de commandes est à 30 %, mais la latence p99 a triplé et tous les threads de travail sont occupés. Pourquoi l'autoscaler basé sur le CPU n'a-t-il pas réagi, et quel signal l'aurait fait ?

Le service attend des E/S : les threads sont donc épuisés alors que le CPU est inactif. Le nombre de requêtes en cours par instance, comparé au nombre de threads, aurait augmenté avec la charge.

▼ Quel axe du cube de scalabilité répond au cas d'une base de données qui a dépassé la capacité d'écriture d'un seul serveur ?

L'axe Z : partitionner les données selon une clé pour que chaque shard ou cellule traite une tranche des écritures.

▼ Pourquoi les autoscalers ajoutent-ils généralement des instances rapidement mais en retirent-ils lentement ?

Une capacité trop faible pénalise immédiatement les utilisateurs alors qu'une capacité trop grande ne coûte que de l'argent, et réduire trop vite provoque des oscillations quand la charge revient.

▼ Pourquoi l'autoscaling réactif ne peut-il pas protéger une mise en vente de billets qui commence à 10 h 00 précises ?

La demande bondit en quelques secondes, mais détecter la hausse, démarrer des instances et les chauffer prend plusieurs minutes : la capacité doit donc être planifiée avant le début de la vente.

▼ À vous : La minute la plus chargée a atteint 1500 rps, et une croissance de 20 % est attendue. Vous prévoyez 200 rps par instance, avec 120 ms de réponse moyenne et 32 threads. Trois zones, en survivant à la perte de l'une d'elles. Combien d'instances ? Les threads suffisent-ils ?

Cible de pointe : 1500 x 1,2 = 1800 rps, soit 9 instances occupées. Deux zones doivent tout porter : donc 4,5 par zone, arrondi à 5, soit 15 au total ; dix instances donnent alors 2000 rps. Selon la loi de Little, chaque instance a 200 x 0,12 = 24 requêtes en cours, dans la limite de ses 32 threads. Vérifiez les connexions à la base de données, et planifiez la capacité avant le déjeuner, car la mise à l'échelle réactive a du retard.

SOURCES

- Kubernetes documentation: Horizontal Pod Autoscaling: https://kubernetes.io/docs/concepts/workloads/autoscaling/horizontal-pod-autoscale/
- Chris Richardson: The Scale Cube (microservices.io): https://microservices.io/articles/scalecube.html
- The Twelve-Factor App: VI. Processes: https://12factor.net/processes
- Brendan Gregg: The USE method: https://brendangregg.com/usemethod.html

SECTION 9.2 SUR 9.8 · INTERMÉDIAIRE · 27 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Équilibrage de charge, absence d'état et sessions

AVANT DE LIRE

Cette section s'appuie sur :

- 0.4 Comment les ordinateurs communiquent
- 7.4 Réplication et partitionnement
- 9.1 Passage à l'échelle

Termes déjà vus qu'elle utilise : Hachage cohérent (consistent hashing), DNS, TCP, gRPC

L'IDÉE EN BREF

Imaginez un supermarché où un membre du personnel oriente les clients vers les caisses. Envoyer les gens vers chaque caisse à tour de rôle est équitable mais naïf : un client avec un chariot rempli pour une famille de six personnes bloque une caisse pendant que la suivante est inoccupée. Envoyer chaque client vers la caisse où la file est la plus courte fonctionne bien mieux. La personne qui oriente doit aussi savoir quelles caisses sont ouvertes, doit cesser d'envoyer des clients vers une caisse sur le point de fermer et laisser sa file se terminer, et travaille au mieux si n'importe quelle caisse peut servir n'importe quel client, pour que personne ne soit prié de revenir vers le même caissier, celui qui détient sa carte de fidélité.

CE QU'IL FAUT SAVOIR

- Un équilibreur de charge répartit les connexions ou les requêtes sur les instances en bonne santé, masque les changements d'instances derrière une adresse unique et assure souvent la terminaison TLS ; il peut se trouver en périphérie ou à l'intérieur des clients et des sidecars.
- Les équilibreurs de couche 4 transmettent les connexions selon l'adresse et le port, rapidement et sans rien savoir du protocole ; les équilibreurs de couche 7 lisent chaque requête HTTP et peuvent router, réessayer et limiter le débit, ce qu'exigent les protocoles à équilibrer par requête comme gRPC sur HTTP/2.
- Le round robin ignore le coût des requêtes ; « moins de connexions » et la puissance de deux choix aléatoires (Mitzenmacher) s'adaptent à la charge, et deux choix aléatoires évitent l'effet de troupeau que provoquent de nombreux équilibreurs lorsqu'ils se ruent tous sur l'instance la moins chargée.

- Un échec de la sonde de vivacité redémarre une instance, et un échec de la sonde de disponibilité la retire seulement de la rotation ; laissez les dépendances partagées hors des sondes de vivacité pour qu'un incident passager de la base de données ne puisse pas redémarrer tout le parc.
- Le drainage des connexions arrête les nouvelles requêtes vers une instance qui s'en va et laisse le travail en cours se terminer, si bien que les déploiements et les retraits d'instances ne provoquent aucune erreur.
- Les sessions persistantes provoquent une charge inégale et des sessions perdues en cas de défaillance ; déplacez l'état de session vers des jetons signés, un stockage partagé ou la base de données pour que n'importe quelle instance puisse servir n'importe quelle requête.
- L'équilibrage de charge global oriente les utilisateurs vers les régions par des réponses DNS, limitées par la mise en cache et les TTL, ou par anycast, qui annonce une même adresse depuis de nombreux sites et laisse le routage choisir le plus proche.

Ce que fait un équilibreur de charge

Un équilibreur de charge se place entre les clients et un pool d'instances interchangeables et décide, pour chaque connexion ou requête, quelle instance la reçoit. La répartition n'est qu'une partie du travail. Un équilibreur de charge vérifie aussi quelles instances sont en bonne santé et cesse d'envoyer du trafic à celles qui ne le sont pas ; il donne aux clients une adresse stable unique pendant que les instances vont et viennent au gré des déploiements et de l'autoscaling ; il assure souvent la terminaison TLS, de sorte que les certificats sont gérés en un seul endroit ; et en couche 7, il peut router, réessayer, limiter le débit et enregistrer chaque requête. C'est donc le composant qui rend la scalabilité horizontale invisible pour les clients.

L'équilibrage de charge intervient à plusieurs niveaux. Un équilibreur de charge cloud managé ou une paire de proxys comme HAProxy, NGINX ou Envoy se trouve généralement en périphérie. Entre les services, l'équilibrage se fait de plus en plus côté client : une bibliothèque cliente gRPC ou un proxy sidecar de service mesh (section 6.3) détient la liste des instances en bonne santé fournie par la découverte de services et en choisit une lui-même, ce qui supprime un saut réseau et un composant central. Quelle que soit la forme, les mêmes décisions reviennent : ce que l'équilibreur peut voir, quel algorithme il utilise, comment il décide qu'une instance est en bonne santé, et comment les instances partent sans perdre de travail.

Équilibrage de charge de couche 4 et de couche 7

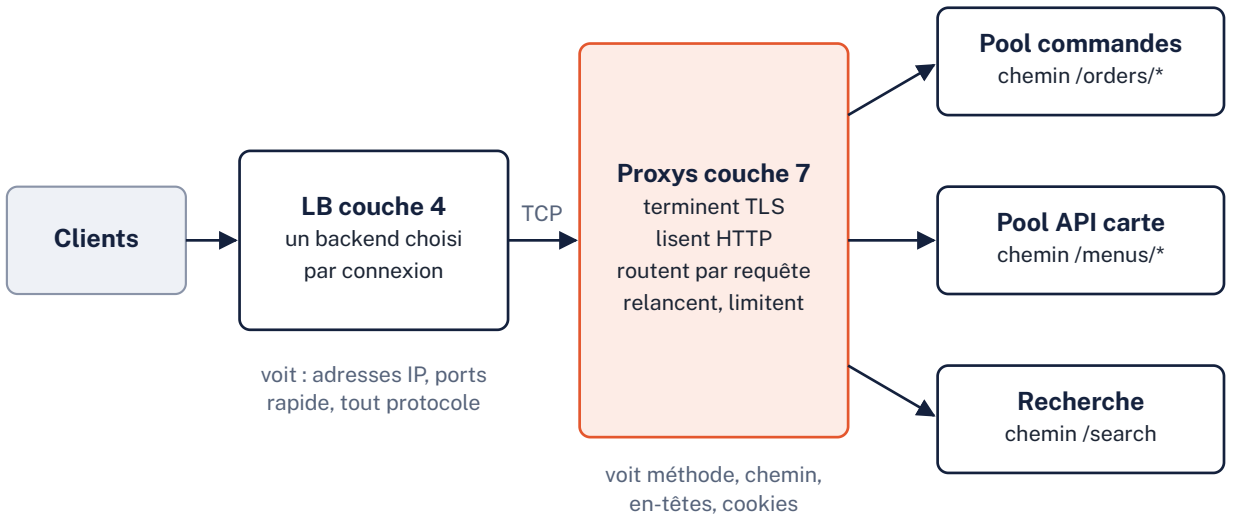
Les numéros de couche viennent du modèle réseau OSI. Un équilibreur de charge de couche 4 travaille au niveau de la couche transport : il voit les adresses IP et les ports TCP ou UDP, choisit un backend à l'ouverture d'une connexion, puis transmet les paquets de cette connexion sans regarder à l'intérieur. Comme il en fait très peu, il est extrêmement rapide et fonctionne

avec n'importe quel protocole, y compris les bases de données, les courtiers de messages et le trafic chiffré qu'il ne peut pas lire. Maglev de Google et les équilibreurs de charge réseau des principaux clouds sont des conceptions de couche 4 qui traitent des millions de paquets par seconde.

Un équilibreur de charge de couche 7 est un proxy inverse qui comprend le protocole applicatif, presque toujours HTTP. Il termine la connexion du client (et généralement TLS), lit chaque requête, et ouvre ou réutilise ses propres connexions vers les backends. Cela lui permet de router selon le chemin, l'hôte, un en-tête ou un cookie (`/orders/*` vers le pool de commandes, `/menus/*` vers le pool de la carte), d'équilibrer chaque requête plutôt que chaque connexion, de réessayer une requête échouée sur une autre instance, d'appliquer des limites de débit, d'ajouter des en-têtes de traçage et de journaliser chaque requête avec son statut et sa latence. Le prix à payer est davantage de traitement par requête, un composant qui doit détenir les clés TLS, et un endroit de plus où les délais d'expiration et les nouvelles tentatives doivent être configurés de manière cohérente avec le reste du système (section 9.4).

Une différence prend de nombreuses équipes au dépourvu. HTTP/2 et gRPC multiplexent de nombreuses requêtes sur une seule connexion de longue durée. Un équilibreur de couche 4 choisit un backend une fois par connexion, si bien qu'un client qui ouvre une seule connexion envoie toutes ses requêtes à une seule instance, quel que soit le nombre d'instances : les nouvelles instances ajoutées par l'autoscaling ne reçoivent rien tant que les clients ne se reconnectent pas. Équilibrer le trafic gRPC exige donc un proxy de couche 7 ou un équilibrage côté client par requête.

Une organisation typique à deux niveaux



Un niveau de couche 4 répartit les connexions à faible coût ; un niveau de couche 7 lit chaque requête et la route selon son chemin vers le bon pool.

Les algorithmes d’équilibrage de charge

L’algorithme décide quelle instance en bonne santé reçoit la connexion ou la requête suivante. Les plus simples ignorent à quel point les instances sont occupées ; les meilleurs utilisent une mesure de la charge ; ceux fondés sur le hachage envoient délibérément les requêtes liées au même endroit. Le tableau résume les choix courants.

Algorithmes d’équilibrage de charge courants

Algorithme	Comment il choisit	Points forts	Faiblesses
Round robin (à tour de rôle)	Chaque instance à tour de rôle	Simple, prévisible, aucun état sur la charge	Aveugle au coût des requêtes et aux instances lentes
Round robin pondéré	À tour de rôle, proportionnellement à des poids configurés	Mélange des tailles d’instances ; augmente progressivement la part des nouvelles versions (canaris)	Les poids sont des estimations statiques ; reste aveugle à la charge réelle
Aléatoire	Uniformément au hasard	Ne nécessite aucun état partagé : de nombreux équilibreurs peuvent donc l’utiliser indépendamment	Inégal par hasard, surtout avec peu de requêtes
Moins de connexions (least connections, ou moins de requêtes en attente)	L’instance qui a le moins de requêtes en cours	S’adapte aux coûts de requête inégaux et aux instances lentes	Exige des comptes exacts ; chacun des nombreux équilibreurs ne voit que les siens ; les nouvelles instances peuvent être submergées
Pondéré par la latence (par exemple EWMA du temps de réponse)	L’instance qui a la meilleure latence récente, souvent combinée à la charge	S’écarte rapidement des instances dégradées	Peut osciller ; une instance qui échoue vite paraît attrayante si les erreurs ne sont pas pénalisées
Hachage (IP source, en-tête, hachage cohérent)	Une même clé correspond toujours à la même instance	Affinité pour les caches et les sessions ; le hachage cohérent déplace peu de clés quand le pool change	Les clés chaudes surchargent une instance ; distribution inégale des clés
Puissance de deux choix aléatoires (power of two choices)	Choisir deux instances au hasard ; envoyer à la moins chargée	Presque aussi bon que « moins de connexions », sans effet de troupeau, avec peu de coordination	Légèrement moins bon qu’un choix global parfait fondé sur une information parfaite

EXEMPLE COMMENTÉ

Exemple commenté : pourquoi « moins de connexions » bat le round robin quand les coûts des requêtes diffèrent

Deux instances, A et B, traitent chacune une requête à la fois (les instances réelles ont davantage de threads, mais l'effet est le même à plus grande échelle). Huit requêtes arrivent à 10 ms d'intervalle. La première est un rapport de ventes de restaurant qui coûte 100 ms ; les sept autres sont des lectures de la carte qui coûtent 10 ms chacune.

Round robin : répartition	A reçoit les requêtes 1, 3, 5, 7 ; B reçoit 2, 4, 6, 8, quel que soit leur coût
Round robin sur A	R1 s'exécute de 0 à 100 ms. R3 (arrivée à 20) attend et se termine à 110 : 90 ms. R5 (40) se termine à 120 : 80 ms. R7 (60) se termine à 130 : 70 ms
Round robin sur B	R2, R4, R6 et R8 se terminent chacune 10 ms après leur arrivée ; B est inactive la moitié du temps
Résultat du round robin	Lectures de la carte : 90, 80, 70, 10, 10, 10, 10 ms, moyenne 290 ÷ 7 ≈ 41 ms, pire cas 90 ms. Les huit requêtes : moyenne (100 + 290) ÷ 8 ≈ 49 ms
« Moins de connexions » : répartition	R1 va à A (les deux sont inactives). À 10 ms, A a 1 requête en cours et B en a 0 : R2 va donc à B ; B est toujours de nouveau libre avant l'arrivée suivante, donc R3 à R8 vont toutes à B
Résultat de « moins de connexions »	Chaque lecture de la carte prend 10 ms ; le rapport prend 100 ms. Moyenne (100 + 70) ÷ 8 ≈ 21 ms, pire lecture de la carte 10 ms
Pourquoi	Le round robin compte les requêtes, pas le travail. « Moins de connexions » compte le travail en cours, qui, d'après la loi de Little (en cours = débit × temps), augmente sur l'instance la plus lente ou la plus occupée, si bien que les nouvelles requêtes s'en détournent automatiquement

Le même effet apparaît chaque fois que les coûts des requêtes varient fortement (recherches contre consultations, exports contre lectures) ou qu'une instance est plus lente que ses pairs à cause d'un voisin bruyant, d'une pause du ramasse-miettes ou d'un cache froid.

« Moins de connexions » exige un compte qui ait du sens. Avec HTTP/2 ou gRPC, de nombreuses requêtes partagent une même connexion : les équilibreurs comptent donc plutôt les requêtes en attente ; avec des connexions WebSocket de longue durée, le compte mesure des sessions, pas la charge.

Une instance qui vient de démarrer a zéro connexion et peut être submergée avant que ses caches ne soient chauds ; de nombreux équilibreurs proposent une période de démarrage progressif (slow start) qui augmente sa part en une minute environ.

La puissance de deux choix aléatoires

Si « moins de connexions » fonctionne si bien, pourquoi ne pas toujours choisir l'instance la moins chargée globalement ? Parce que, dans un grand système, il n'existe aucune vue unique et actuelle de la charge. Des dizaines d'équilibreurs de charge ou des milliers d'équilibreurs côté client ne voient chacun que leurs propres requêtes, ou un relevé de charge vieux de quelques secondes. S'ils choisissent tous l'instance qui semblait la moins chargée, ils choisissent tous la *même*, qui est aussitôt submergée, puis ils passent tous à la suivante : un troupeau qui poursuit la charge à travers le pool.

La thèse de doctorat de Michael Mitzenmacher (1996), publiée sous le titre *The Power of Two Choices in Randomized Load Balancing* (IEEE Transactions on Parallel and Distributed Systems, 2001) et s'appuyant sur des travaux antérieurs d'Azar, Broder, Karlin et Upfal, a analysé une alternative d'une simplicité frappante : choisir deux instances au hasard et envoyer la requête à la moins chargée des deux. Dans le modèle classique des balles et des urnes, placer n requêtes au hasard sur n serveurs donne un serveur le plus chargé avec environ log n ÷ log log n requêtes ; avec deux choix aléatoires, ce nombre tombe à environ log log n ÷ log 2, une amélioration exponentielle, tandis qu'un troisième choix n'apporte que peu de chose de plus. Le hasard empêche l'effet de troupeau, et la comparaison évite les pires instances.

La puissance de deux choix aléatoires est désormais l'option par défaut ou une option standard dans des proxys très répandus : l'équilibreur least-request d'Envoy compare par défaut deux hôtes choisis au hasard, et NGINX propose `random two least_conn`. C'est un bon choix par défaut pour l'équilibrage côté client dans tout système comportant de nombreux équilibreurs.

Hachage et affinité

Parfois, le but est d'envoyer des requêtes liées à la *même* instance : à un nœud de cache qui détient déjà la carte d'un restaurant, ou à l'instance qui détient une session WebSocket. Hacher une clé (identifiant du restaurant, identifiant de l'utilisateur, IP source) vers une instance permet d'y parvenir. Un simple `hash mod N` réaffecte la plupart des clés chaque fois que N change, ce qui vide tous les caches au pire moment, pendant un ajout d'instances. Le hachage cohérent (présenté à la section 7.4) ne déplace qu'environ 1 ÷ N des clés quand une instance arrive ou part. Sa faiblesse, ce sont les clés chaudes : le restaurant le plus populaire à midi

tombe sur une seule instance. Des variantes comme le hachage cohérent à charge bornée plafonnent la part de chaque instance et déversent l'excédent sur l'instance suivante de l'anneau.

Contrôles de santé : vivacité et disponibilité

Un équilibreur de charge doit savoir quelles instances peuvent recevoir du trafic. Un contrôle de santé actif sonde chaque instance périodiquement, par exemple `GET /healthz` toutes les cinq secondes, la marque comme hors service après quelques échecs consécutifs et de nouveau en service après quelques succès. Les contrôles passifs, souvent appelés détection des valeurs aberrantes (outlier detection), observent le trafic réel et éjectent une instance qui renvoie, par exemple, cinq erreurs serveur consécutives. Les deux se complètent : les contrôles actifs trouvent les instances qui ne reçoivent pas de trafic, et les contrôles passifs réagissent aux défaillances qu'une sonde triviale ne met pas à l'épreuve.

Kubernetes a rendu explicite une distinction importante. Une sonde de vivacité (liveness probe) demande si le processus est bloqué sans possibilité de rétablissement (interblocage, mémoire épuisée dans une boucle) ; si elle échoue, l'instance est redémarrée. Une sonde de disponibilité (readiness probe) demande si l'instance doit recevoir du trafic maintenant ; si elle échoue, l'instance est retirée de l'équilibreur de charge mais continue de tourner, par exemple pendant qu'elle chauffe ses caches au démarrage, ou pendant qu'elle déleste de la charge. Une troisième, la sonde de démarrage (startup probe), met les deux autres en attente pendant qu'une application lente démarre.

L'erreur classique consiste à vérifier des dépendances partagées dans ces sondes. Si la sonde de vivacité de chaque instance appelle la base de données, une brève panne de la base fait échouer toutes les sondes, l'orchestrateur redémarre toutes les instances à la fois, et la tempête de redémarrages (caches froids, milliers de nouvelles connexions) prolonge la panne bien après le retour de la base de données. Des contrôles de disponibilité qui incluent une dépendance partagée peuvent de même marquer tout le parc comme non disponible, transformant des erreurs partielles en panne totale. Gardez la sonde de vivacité locale et minimale ; laissez la sonde de disponibilité refléter la capacité propre de l'instance à servir ; et gérez les défaillances des dépendances avec les patrons de la section 9.4. Les équilibreurs de charge se prémunissent contre le cas extrême en échouant en mode ouvert : un Application Load Balancer d'AWS route vers toutes les cibles si toutes échouent à leurs contrôles de santé, et Envoy ignore l'état de santé quand la part d'hôtes en bonne santé tombe sous un seuil de panique (50 % par défaut), au motif qu'envoyer du trafic à des instances douteuses vaut mieux que de ne l'envoyer nulle part.

SI VOUS DÉBUTEZ

Pas à pas : un contrôle qui redémarre tout

L'idée la plus difficile de cette section est que les deux contrôles de santé font des choses différentes, et que placer une dépendance partagée dans le mauvais peut transformer un incident passager en panne. Un échec de la sonde de vivacité fait redémarrer une instance ; un échec de la sonde de disponibilité la retire seulement de la rotation de l'équilibreur de charge.

- Supposons que l'API de commandes tourne sur 30 instances, chacune avec une sonde de vivacité qui ne signale un bon état que si elle peut joindre le courtier de messages partagé.
- Le courtier est injoignable pendant environ une minute, le temps de remplacer un nœud. Toutes les sondes échouent, alors que les instances elles-mêmes vont bien.
- Après quelques échecs consécutifs, l'orchestrateur conclut que toutes les instances sont bloquées sans possibilité de rétablissement et redémarre les 30 ensemble.
- Le courtier revient, mais 30 instances toutes neuves s'y reconnectent maintenant en même temps et remplissent des caches vides à partir de la base de données des commandes. Personne ne peut commander pendant qu'elles chauffent, et la tempête de redémarrages frappe les deux, transformant un incident d'une minute en longue panne.
- Si la sonde de vivacité n'avait vérifié que l'instance elle-même, rien n'aurait redémarré : les instances auraient conservé leurs caches chauds et continué à servir ; seul le travail nécessitant le courtier aurait attendu.
- La sonde de disponibilité demande elle aussi de la prudence. Si le contrôle de disponibilité de chaque instance appelle aussi ce courtier, tout le parc est marqué non disponible en même temps et l'équilibreur de charge n'a plus nulle part où envoyer quoi que ce soit. C'est pourquoi certains équilibreurs de charge échouent en mode ouvert : quand toutes les instances semblent en mauvaise santé, ils leur envoient malgré tout du trafic à toutes.
- **La leçon :** gardez la sonde de vivacité locale et minimale, laissez la sonde de disponibilité refléter la capacité de cette instance à servir maintenant, et gérez une dépendance partagée défaillante avec les patrons de la section 9.4, pas en redémarrant le parc.

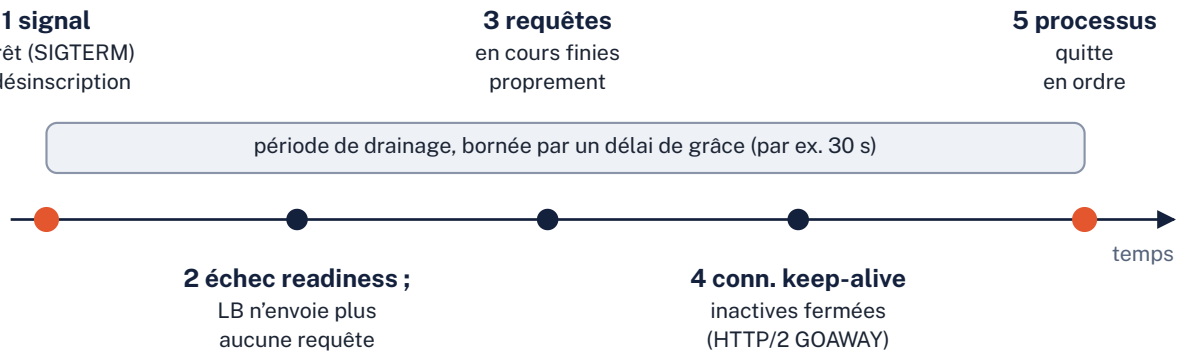
Drainage des connexions et arrêt en douceur

Les instances quittent le pool en permanence : chaque déploiement les remplace, l'autoscaling les retire, et la plateforme les déplace lors des maintenances. Si une instance est simplement tuée, chaque requête qu'elle traitait échoue, et les clients voient des erreurs pendant ce qui devrait être une mise en production de routine. Le drainage des connexions (appelé

deregistration delay chez AWS) retire l'instance de la rotation de l'équilibreur pour qu'elle ne reçoive plus de *nouvelles* requêtes, attend que les requêtes déjà en cours se terminent, et ne la laisse s'arrêter qu'ensuite.

L'instance doit coopérer. À la réception de son signal d'arrêt, elle doit faire échouer sa sonde de disponibilité, continuer à servir les requêtes en cours, dire aux clients sur des connexions keep-alive d'aller ailleurs (un en-tête `Connection: close` en HTTP/1.1, une trame GOAWAY en HTTP/2), terminer ou rendre le travail d'arrière-plan, et se terminer avant la fin du délai de grâce. Dans Kubernetes, le retrait d'un pod de l'équilibreur de charge et l'envoi de SIGTERM se produisent en parallèle : une requête peut donc encore arriver juste après le signal ; un court délai pré-arrêt (pre-stop) de quelques secondes couvre cette situation de concurrence. Le délai de grâce par défaut est de 30 secondes, après quoi le processus est tué. Les requêtes de longue durée (gros envois de fichiers, réponses en streaming, WebSockets) exigent soit un délai de grâce plus long, soit un protocole qui permet aux clients de se reconnecter ailleurs sans perdre de travail.

L'arrêt en douceur en cinq étapes



Le drainage transforme le retrait d'une instance, source d'erreurs, en non-événement : aucun nouveau travail n'arrive, le travail existant se termine, puis le processus s'arrête.

Les sessions persistantes et leurs problèmes

Les applications web plus anciennes conservaient la session de chaque utilisateur (qui est connecté, ce que contient le panier) dans la mémoire du serveur qui l'avait créée. Derrière un équilibreur de charge, cela ne fonctionne que si toutes les requêtes d'un utilisateur reviennent vers ce serveur ; les équilibreurs proposent donc des sessions persistantes (sticky sessions, ou affinité de session), généralement en posant un cookie qui désigne l'instance. La persistance est un moyen rapide de placer une application avec état derrière un équilibreur, et elle a un usage légitime comme indication de performance, par exemple pour garder un utilisateur sur une instance dont le cache local contient déjà ses données.

Comme garantie de bon fonctionnement, en revanche, elle pose des problèmes. La charge devient inégale, car quelques gros utilisateurs ou un grand bureau derrière une seule adresse IP restent attachés à une instance quelle que soit sa charge. Quand une instance tombe en panne, ses utilisateurs perdent leur session : les paniers disparaissent et les gens sont déconnectés. Les retraits d'instances et chaque déploiement ont le même effet, sauf si les sessions sont drainées, ce qui peut prendre autant de temps que la plus longue session. Les nouvelles instances ajoutées par l'autoscaling ne reçoivent que les nouveaux utilisateurs : elles soulagent donc lentement les instances surchargées. Et le système n'est plus sans état au sens de la section 9.1, si bien que chacun de ces problèmes réapparaît lors d'un basculement entre zones ou entre régions.

Externaliser l'état de session

Le remède consiste à sortir l'état de session de l'instance, pour que n'importe quelle instance puisse servir n'importe quelle requête. Il existe trois options principales, souvent combinées. Un **jeton côté client** transporte l'état avec chaque requête : un cookie signé (et si nécessaire chiffré) ou un jeton comme un JWT contenant l'identité de l'utilisateur et quelques données. Les serveurs n'ont besoin d'aucun stockage, mais les jetons doivent rester petits, ne peuvent pas facilement être révoqués avant leur expiration et doivent être protégés contre la falsification et le vol (section 8.3). Un **stockage de sessions partagé**, typiquement Redis ou Memcached, conserve les sessions dans un stockage externe rapide, indexé par un identifiant de session placé dans le cookie ; il gère un état plus volumineux et la déconnexion immédiate, au prix d'un appel réseau par requête et d'une dépendance qui doit elle-même être hautement disponible. Une **base de données** convient à l'état qui doit survivre à tout, comme un panier que l'utilisateur s'attend à retrouver le lendemain, qui relève en réalité des données métier plutôt que des données de session.

Un découpage pratique pour la plateforme de livraison de repas : l'identité dans un jeton signé de courte durée, le panier dans la base de données du service de commandes (les utilisateurs passent du téléphone à l'ordinateur portable), et l'état transitoire de l'interface dans le client. Aucune instance ne conserve quoi que ce soit dont une requête ultérieure a besoin : l'équilibreur de charge peut donc utiliser un équilibrage « moins de requêtes » sans affinité, et les instances peuvent être ajoutées, retirées et remplacées à volonté.

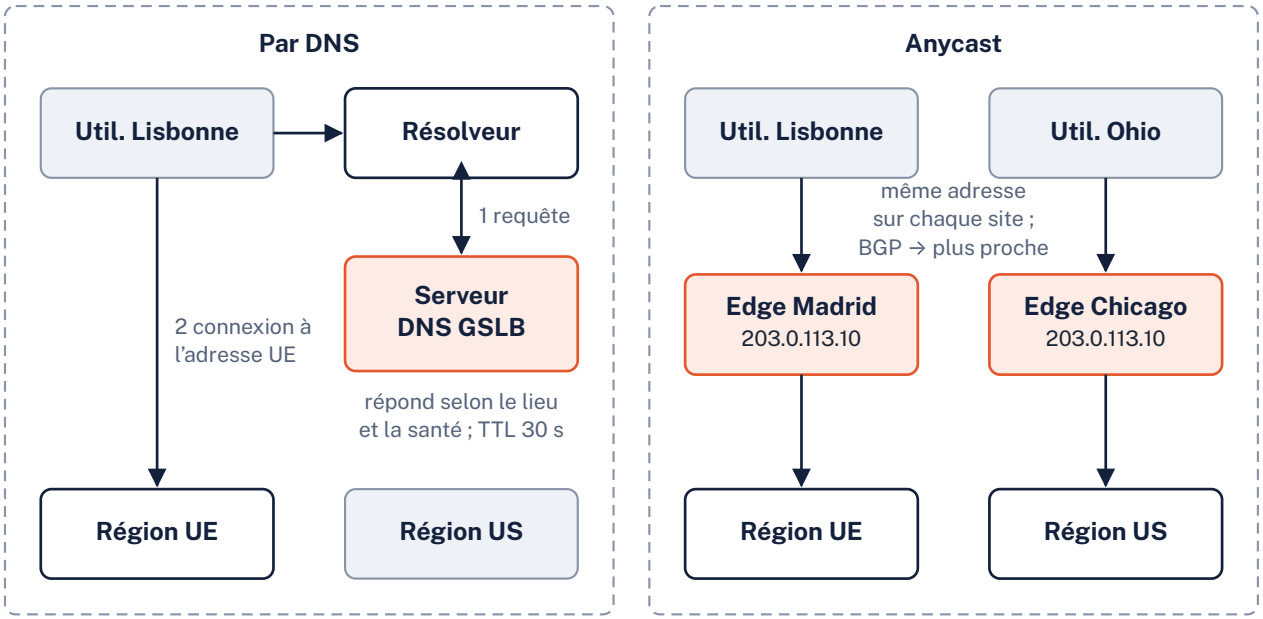
Équilibrage de charge global : DNS et anycast

Dès qu'un système tourne dans plusieurs régions, quelque chose doit envoyer chaque utilisateur vers une région adaptée : la région en bonne santé la plus proche, ou celle qui détient ses données. L'équilibrage de charge global se fait généralement de l'une de deux façons, souvent combinées.

L'équilibrage de charge global des serveurs **fondé sur le DNS** utilise un serveur de noms faisant autorité qui répond au même nom par des adresses différentes selon l'origine de la requête, la santé de chaque région et des poids configurés : un utilisateur à Lisbonne reçoit l'adresse de la région UE, un utilisateur dans l'Ohio celle de la région US. Il ne nécessite aucun réseau particulier, mais il est grossier et lent à réagir. Les réponses sont mises en cache par les résolveurs au moins pendant la durée de vie de l'enregistrement (souvent 30 à 300 secondes), et parfois plus longtemps : après la défaillance d'une région, certains utilisateurs continuent donc d'y arriver pendant plusieurs minutes. Le serveur de noms voit l'emplacement du résolveur plutôt que celui de l'utilisateur, même si l'extension EDNS Client Subnet (RFC 7871) permet aux résolveurs de transmettre une partie de l'adresse du client.

Avec l'anycast, la même adresse IP est annoncée depuis de nombreux sites par le protocole de routage d'internet (BGP), et les paquets de chaque utilisateur sont livrés au site le plus proche au sens du routage. Le basculement consiste à retirer l'annonce d'un site défaillant, ce qui prend effet rapidement sans attendre les caches DNS. L'anycast est utilisé depuis longtemps pour les serveurs racines du DNS et les services de résolution ; les CDN et les équilibreurs de charge globaux des grands fournisseurs de cloud l'utilisent pour amener les utilisateurs vers un site périphérique proche, où des proxys de couche 7 terminent la connexion et transmettent les requêtes, via le réseau du fournisseur, à une région en bonne santé. La conception multirégion, le placement des données et le basculement sont le sujet de la section 10.6.

Équilibrage de charge global par DNS et par anycast



Le DNS oriente les utilisateurs en répondant par des adresses différentes ; l'anycast donne à tous la même adresse et laisse le routage d'internet les livrer au site le plus proche.

ATTENTION

Erreurs courantes

- Placer du trafic gRPC ou HTTP/2 derrière un équilibreur de couche 4 et se demander pourquoi les nouvelles instances ne reçoivent aucune requête.
- Utiliser le round robin pour des services dont les requêtes varient fortement en coût, ou dont certaines instances sont plus lentes que d'autres.
- Appeler la base de données depuis les sondes de vivacité, si bien qu'un incident passager de la base redémarre tout le parc.
- Tuer des instances sans drainage, si bien que chaque déploiement et chaque retrait d'instances produisent une rafale d'erreurs.
- Compter sur les sessions persistantes pour le bon fonctionnement, puis perdre des paniers et des connexions chaque fois qu'une instance tombe en panne ou est remplacée.
- Supposer que le basculement DNS est instantané ; les clients et les résolveurs peuvent continuer d'utiliser une ancienne réponse pendant plusieurs minutes.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi un serveur gRPC nouvellement ajouté derrière un équilibreur de charge de couche 4 pourrait-il ne recevoir aucun trafic ?

L'équilibrage de couche 4 choisit un backend par connexion, et les clients gRPC gardent une seule connexion HTTP/2 de longue durée : les clients existants ne lui envoient donc aucune requête tant qu'ils ne se reconnectent pas.

▼ Quelle est la différence d'effet entre l'échec d'une sonde de vivacité et l'échec d'une sonde de disponibilité ?

L'échec de la sonde de vivacité amène l'orchestrateur à redémarrer l'instance ; l'échec de la sonde de disponibilité la retire seulement de l'équilibreur de charge jusqu'à ce qu'elle la réussisse de nouveau.

▼ Pourquoi de nombreux équilibreurs de charge indépendants qui choisissent l'instance la moins chargée obtiennent-ils de mauvais résultats ?

Ils agissent sur la même vue périmée et choisissent tous la même instance, qu'ils surchargent tour à tour ; choisir la meilleure de deux instances tirées au hasard évite cet effet de troupeau.

▼ **Citez deux problèmes causés par la conservation des sessions dans la mémoire des instances avec des sessions persistantes.**

Les utilisateurs perdent leur session quand l'instance tombe en panne ou est remplacée, et la charge devient inégale parce que les gros utilisateurs restent attachés à une seule instance.

▼ **À vous : La sonde de vivacité d'une API de commandes exécute une requête sur la base de données. Lors d'une brève panne de 30 secondes de la base, l'orchestrateur redémarre toutes les instances, et l'interruption dure dix minutes. Expliquez pourquoi et reconcevez les sondes. La sonde de disponibilité doit-elle inclure la base de données ?**

Un échec de vivacité signifie un redémarrage : un seul incident passager sur une dépendance partagée a donc redémarré toute la flotte, et les démarrages à froid ont duré bien plus longtemps que l'incident. La sonde de vivacité doit seulement demander si ce processus fonctionne, sans dépendances partagées. La sonde de disponibilité retire une instance de la rotation sans la redémarrer : elle peut donc inclure l'essentiel, mais si toutes les instances y échouent ensemble, le service disparaît complètement. Il peut être préférable de continuer à servir et de renvoyer rapidement des erreurs ou une réponse dégradée.

SOURCES

- Michael Mitzenmacher: The Power of Two Choices in Randomized Load Balancing (IEEE TPDS, 2001): https://www.eecs.harvard.edu/~michaelm/postscripts/tpds2001.pdf
- Kubernetes documentation: Liveness, Readiness, and Startup Probes: https://kubernetes.io/docs/concepts/configuration/liveness-readiness-startup-probes/
- Envoy documentation: Supported load balancers: https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/load_balancing/load_balancers
- NGINX documentation: HTTP load balancing: https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/

Asynchronisme, files d'attente et back pressure

AVANT DE LIRE

Cette section s'appuie sur :

- 2.4 Performances et scalabilité
- 6.4 Pilotée par les événements
- 6.5 Communication et API

Termes déjà vus qu'elle utilise : Loi de Little, Idempotent, Couplage temporel, Débit (throughput)

L'IDÉE EN BREF

Une boulangerie très fréquentée distribue, à l'heure de l'ouverture, des tickets numérotés : les clients attendent dans une file ordonnée au lieu de se presser au comptoir, et les boulangers travaillent à un rythme régulier. C'est une file d'attente. Mais une file ne peut pas s'allonger indéfiniment avant que le pain vienne à manquer ou que les gens abandonnent ; une bonne boulangerie cesse donc aussi de distribuer des tickets quand la file est trop longue, sert d'abord les précommandes, limite chaque client à un nombre raisonnable de pains, et dit honnêtement aux retardataires de revenir dans dix minutes plutôt que de les laisser attendre pour rien.

CE QU'IL FAUT SAVOIR

- Une file d'attente entre producteur et consommateur transforme un pic en un arriéré que les consommateurs résorbent à leur propre rythme, ce qui permet de les dimensionner pour la moyenne ; le prix à payer est un délai pour chaque élément placé derrière l'arriéré.
- Les consommateurs concurrents font passer le travail d'arrière-plan à l'échelle par l'ajout de workers ; la livraison se fait au moins une fois, donc les consommateurs doivent être idempotents, et les délais de visibilité et le préchargement doivent correspondre aux temps de traitement.
- Une file ne peut pas absorber un excès durable ; bornez chaque file à l'aide de la loi de Little (longueur maximale = débit de traitement × attente acceptable) et surveillez l'âge du plus ancien message en plus de la longueur.
- Le back pressure fait ralentir les étapes plus rapides ; quand une file bornée est pleine, le système doit bloquer le producteur, rejeter le nouveau travail ou abandonner une partie du travail, et il devrait choisir délibérément.

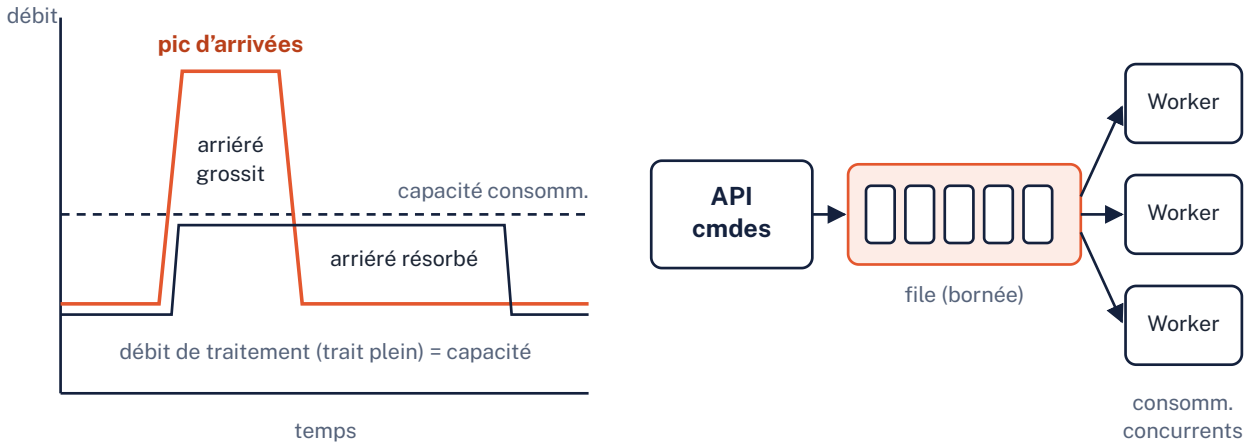
- Le délestage de charge rejette une partie du travail tôt et à moindre coût, par priorité et par ancienneté, afin que le goodput (débit utile) reste élevé en cas de surcharge.
- La limitation de débit répartit la capacité par client : le token bucket autorise des rafales jusqu'à sa capacité, avec un rythme de recharge régulier, le leaky bucket lisse la sortie, et les fenêtres fixes permettent des doubles rafales aux frontières.
- Répondez aux dépassements de limite par un 429 (ou un 503 quand c'est l'ensemble du serveur qui est surchargé) accompagné de Retry-After, et envoyez les messages qui échouent de façon répétée dans une file de lettres mortes surveillée et dotée d'un responsable.

Les files d'attente transforment les pics en arriérés

La charge n'arrive jamais de façon régulière. Les commandes du déjeuner se concentrent à 12:30, les acheteurs de billets appuient tous sur le bouton à 10:00:00, et un traitement par lots déverse un million d'événements à minuit. Un service synchrone doit soit disposer de la capacité nécessaire pour le pic le plus élevé, soit défaillir pendant ce pic. Placer une file d'attente entre le producteur du travail et le service qui l'effectue change le calcul : pendant un pic, le travail excédentaire attend dans la file, et les consommateurs le traitent à leur propre rythme régulier, en rattrapant leur retard une fois le pic passé. Le catalogue de patrons cloud de Microsoft appelle cela le nivellement de charge par file d'attente (queue-based load levelling). La section 2.4 a montré pourquoi des files se forment chaque fois que les arrivées se concentrent ; cette section porte sur leur utilisation délibérée.

L'avantage est réel : les consommateurs peuvent être dimensionnés pour la moyenne plutôt que pour le pic, un consommateur lent ou en cours de redémarrage ne perd pas de travail, et le producteur n'est plus couplé temporellement au consommateur (section 6.5). Le prix à payer est que le travail est désormais effectué plus tard. Une file transforme un pic en *arriéré*, et un arriéré n'est rien d'autre qu'un délai que subira chaque élément placé derrière lui. C'est acceptable pour envoyer des e-mails de confirmation ou mettre à jour un index de recherche, discutable pour affecter un livreur à une commande, et inacceptable pour un paiement dont le client attend la confirmation. La première question de conception pour toute file est donc : combien de temps un élément peut-il attendre, et qu'advient-il des éléments qui attendraient plus longtemps ?

Une file d'attente qui absorbe un pic



Pendant le pic, la file absorbe les arrivées qui dépassent la capacité des consommateurs ; ensuite, les consommateurs résorbent l'arriéré à leur rythme régulier.

Files de travail et consommateurs concurrents

L'usage le plus courant est une file de travail avec des consommateurs concurrents (competing consumers) : plusieurs workers identiques lisent une même file, chaque message ne va qu'à l'un d'eux, et le débit augmente quand on ajoute des workers. C'est l'axe X du cube de scalabilité appliqué au travail d'arrière-plan, et il passe naturellement à l'échelle sur le bon signal : l'arriéré par worker, ou l'âge du plus ancien message.

Les mécanismes comptent pour la justesse. Un worker prend un message, le traite, puis en accuse réception ; si le worker plante avant, le courtier rend le message de nouveau visible après un délai (SQS l'appelle le délai de visibilité, ou visibility timeout) et un autre worker le traite. La livraison se fait donc au moins une fois, et les consommateurs doivent être idempotents (section 6.4). Un délai de visibilité plus court que le temps de traitement réel fait traiter le même message deux fois en parallèle. L'ordre est perdu dès qu'il y a plus d'un consommateur, sauf si les messages sont partitionnés par clé, auquel cas le nombre de partitions plafonne le nombre de consommateurs utiles. Enfin, le préchargement (prefetch : un worker qui prend plusieurs messages à la fois pour économiser des allers-retours) doit rester faible pour les tâches lentes, sinon un worker accapare des messages que des workers inactifs auraient pu traiter.

Une file d’attente ne crée pas de capacité

Une file absorbe un excès temporaire ; elle ne peut pas absorber un excès permanent. Si le travail arrive plus vite qu’il n’est accompli pendant assez longtemps, l’arriéré croît sans limite, et la loi de Little ($L = \lambda \times W$) indique que le temps d’attente croît avec lui. Une file en mémoire non bornée finit par épuiser la mémoire et faire planter le processus, en perdant tout ce qu’elle contient ; une file durable non bornée remplit son disque ou conserve du travail pendant des heures alors que plus personne ne s’en soucie. Pire, la file masque le problème : les producteurs reçoivent des accusés de réception rapides, les taux d’erreur restent à zéro, et le seul symptôme est que les résultats arrivent de plus en plus tard. Des ingénieurs d’Amazon décrivent de tels arriérés comme potentiellement insurmontables, car un service qui prend assez de retard peut consacrer toute sa capacité à traiter du travail que ses clients ont abandonné depuis longtemps.

Deux pratiques en découlent. D’abord, mesurez l’âge du plus ancien message en plus de la longueur de la file : l’âge est ce que vivent les utilisateurs, et il réagit immédiatement quand les consommateurs se bloquent. Ensuite, donnez à chaque file une borne déduite du temps d’attente que vous jugez acceptable, à l’aide de la loi de Little, et décidez à l’avance de ce qui se passe quand la borne est atteinte. Cette décision fait l’objet des trois parties suivantes de cette section.

EXEMPLE COMMENTÉ

Exemple commenté : dimensionner la file des commandes pour une ouverture de billetterie

Les billets d'un concert dans un stade sont mis en vente à 10:00. Les commandes soumises sont placées dans une file ; des workers autorisent le paiement et attribuent les places, en 250 ms chacun. La prévision est de 600 commandes soumises par seconde pendant les deux premières minutes. La promesse faite aux clients est une réponse en moins de deux minutes. Il y a 25 000 billets et la commande moyenne porte sur 2,5 billets.

Capacité des workers avec 40 workers	40 ÷ 0,25 s = 160 commandes par seconde
Arriéré après le pic de deux minutes	(600 − 160) × 120 s = 52 800 commandes en attente
Attente de la dernière commande de la file (loi de Little, W = L ÷ λ)	52 800 ÷ 160 = 330 s, soit cinq minutes et demie : la promesse n'est pas tenue
File la plus longue qui permet de tenir la promesse	L = λ × W = 160 × 120 = 19 200 commandes
Plus de workers ?	Le contrat du prestataire de paiement autorise 300 requêtes par seconde, donc 75 workers au plus (300 commandes par seconde). L'arriéré devient (600 − 300) × 120 = 36 000 et l'attente 36 000 ÷ 300 = 120 s : la promesse est tenue de justesse, sans aucune marge
Ce que permet le stock	25 000 ÷ 2,5 = 10 000 commandes peuvent aboutir. À 600 par seconde, ce nombre de commandes arrive en moins de 17 secondes ; mettre en file les quelque 62 000 autres commandes ne fait que retarder l'inévitable « épuisé »
Décision	75 workers ; une salle d'attente placée devant la page de commande admet 300 acheteurs par seconde ; la file des commandes est bornée à 12 000 (le stock plus 20 % pour les paiements échoués) ; dès que les réservations couvrent le stock, les nouvelles soumissions reçoivent immédiatement la réponse « épuisé, inscrivez-vous sur la liste d'attente » au lieu de rejoindre la file

La loi de Little transforme une promesse sur le temps d'attente en une longueur maximale de file. Toute file autorisée à croître au-delà est une promesse non tenue qui n'a pas encore été annoncée.

C'est la limite de débit en aval, et non le nombre de workers que la plateforme pouvait démarrer, qui a fixé la capacité réelle : l'ajout de consommateurs se heurtait donc à un plafond strict.

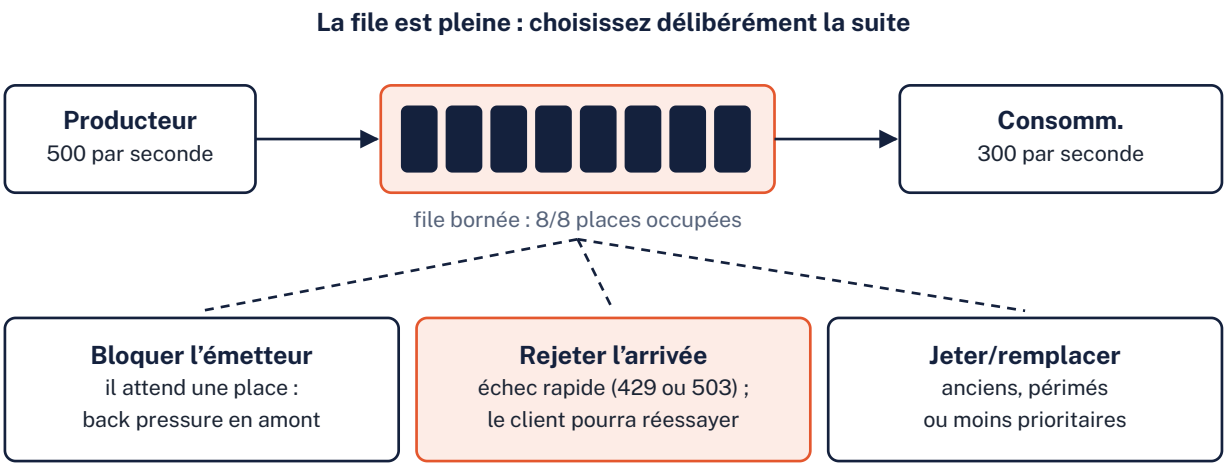
La meilleure borne pour une file est souvent une limite métier : il ne sert à rien de conserver du travail qui ne peut en aucun cas aboutir.

Back pressure : faire ralentir les producteurs

Le back pressure (contre-pression) désigne tout mécanisme par lequel une étape aval plus lente fait ralentir une étape amont plus rapide, au lieu de laisser le travail s'accumuler entre elles. Le terme vient des fluides dans les canalisations, et l'idée a déjà été rencontrée avec les tubes et filtres (section 5.4). Il est intégré à de nombreuses couches de la pile que vous utilisez déjà. Le contrôle de flux de TCP arrête un émetteur quand le tampon du récepteur est plein. Une file bloquante bornée fait attendre le `put` du producteur jusqu'à ce qu'il y ait de la place. Les consommateurs en mode pull, comme dans Kafka, ne prennent du travail que lorsqu'ils sont prêts, si bien que c'est le journal du courtier, et non la mémoire du consommateur, qui contient l'arriéré, et un producteur Kafka se bloque ou échoue quand son propre tampon d'envoi est plein. La spécification Reactive Streams permet à un abonné de demander un nombre précis d'éléments, de sorte qu'un flux ne livre jamais plus que ce que l'abonné a demandé.

Le back pressure fonctionne bien à l'intérieur d'un processus ou le long d'un pipeline. À travers une frontière requête-réponse avec des utilisateurs humains, il se traduit généralement par un rejet : un service HTTP surchargé ne peut pas faire patienter poliment un téléphone, il répond donc `429 Too Many Requests` ou `503 Service Unavailable`, idéalement avec un en-tête `Retry-After`, et compte sur le client pour ralentir. Quand une file bornée est pleine, il n'y a que trois choix : bloquer le producteur, rejeter le nouveau travail, ou abandonner quelque chose de déjà accepté. Chacun est le bon quelque part ; l'erreur est de ne pas choisir.

Trois réponses à une file pleine



Quand une file bornée est pleine, le système doit bloquer, rejeter ou abandonner ; choisir l'une de ces options délibérément est ce qui maintient la surcharge sous contrôle.

Choisir ce qui se passe quand une file est pleine

Stratégie	Comportement	Convient à	Risque
Bloquer le producteur	Le producteur attend qu'il y ait de la place	Pipelines et traitements par lots où le producteur peut faire une pause	Les producteurs bloqués monopolisent threads et connexions ; entre services, le ralentissement se propage vers l'amont
Rejeter le nouveau travail	Échouer rapidement avec une erreur comme 429 ou 503	Requêtes interactives ; le client peut réessayer, attendre ou afficher un message	Les clients qui réessaient immédiatement transforment le rejet en charge supplémentaire (section 9.4)
Abandonner le plus ancien	Écarter l'élément qui attend depuis le plus longtemps	Télémétrie, mises à jour de position, prix en direct : les données les plus récentes sont les plus précieuses	Perte silencieuse ; elle doit être comptée et visible
Abandonner par priorité	Écarter d'abord les éléments les moins importants	Charges mixtes : garder les validations de commande, abandonner les événements analytiques	Les basses priorités risquent la famine ; les priorités doivent être attribuées honnêtement
Déverser vers un stockage durable	Débordement sur disque ou dans une file durable	Travail qui doit finir par être fait mais peut attendre des heures	Ce n'est qu'une file plus grande : il lui faut toujours une borne et une estimation du temps de résorption

Délestage de charge et priorisation

Un serveur soumis à plus de charge qu'il ne peut en traiter fait pire que servir sa capacité et refuser le reste. Il accepte tout, chaque requête attend dans ses files internes, les temps de réponse dépassent les délais d'expiration des clients, et il finit par consacrer toute sa capacité à des requêtes dont les clients ont déjà abandonné. Le débit reste élevé tandis que le *goodput* (débit utile), le rythme des requêtes traitées à temps pour être utiles, s'effondre vers zéro. Le délestage de charge (load shedding) l'empêche en rejetant délibérément certaines requêtes pour que les autres réussissent.

Un bon délestage a trois propriétés. Il est **précoce et peu coûteux** : rejeter une requête doit coûter bien moins cher que la servir, il a donc lieu à l'entrée, avant tout travail lourd d'authentification ou de base de données, à partir d'un signal peu coûteux comme le nombre de requêtes en cours ou l'âge de la file. Il est **priorisé** : le livre SRE de Google décrit l'étiquetage de

chaque requête avec une criticité (de CRITICAL_PLUS à SHEDDABLE), afin que le trafic de traitement par lots et de préchargement soit abandonné avant le trafic destiné aux utilisateurs. Dans la plateforme de livraison de repas, les validations de commande et les mises à jour de position des livreurs passent avant les recommandations, qui passent elles-mêmes avant les événements analytiques. Enfin, il est **conscient du temps** : une requête qui a déjà attendu plus longtemps que le délai d'expiration de son appelant doit être abandonnée plutôt que traitée. Les ingénieurs de Facebook ont décrit deux raffinements : une file à délai contrôlé qui n'autorise qu'une courte attente quand une file ne s'est pas vidée récemment, et le passage au mode dernier entré, premier sorti sous charge, afin que les requêtes les plus récentes, dont les utilisateurs attendent encore, soient servies en premier.

Une étape supplémentaire consiste à adapter la limite elle-même. Des limites de concurrence fixes sont soit trop basses un bon jour, soit trop hautes un mauvais jour ; les limiteurs de concurrence adaptatifs augmentent le nombre autorisé de requêtes en cours tant que la latence reste basse et le réduisent quand elle augmente, en empruntant l'idée au contrôle de congestion de TCP. Tout à l'entrée d'un système, le contrôle d'admission peut prendre la forme d'une salle d'attente virtuelle, comme pour l'ouverture de billetterie ci-dessus : les utilisateurs sont retenus dans une file à la périphérie et admis au rythme que le cœur du système peut servir, ce qui revient à du délestage de charge présenté comme de l'équité.

SI VOUS DÉBUTEZ

Pas à pas : occupé mais inutile

L'idée qui demande le plus d'attention ici est qu'un serveur surchargé qui accepte toutes les requêtes peut finir par ne faire presque aucun travail utile, bien qu'il ne cesse jamais de travailler. Refuser volontairement certaines requêtes, le délestage de charge, est ce qui garde les autres utiles.

- Supposons que l'étape de paiement de la plateforme de livraison de repas puisse traiter 100 requêtes par seconde, et que l'application mobile abandonne toute requête restée sans réponse au bout de 2 secondes.
- Lors d'une promotion, 150 requêtes arrivent chaque seconde. Le service les accepte toutes, si bien que sa file interne grossit de 50 requêtes chaque seconde.
- Au bout de 4 secondes, environ 200 requêtes attendent. À 100 par seconde, une nouvelle arrivée attend environ 2 secondes avant que son traitement commence : d'ici là, son application a déjà abandonné.
- La file continue de grossir, si bien qu'à partir de là presque toutes les requêtes se terminent trop tard. Le service traite toujours 100 requêtes par seconde, donc son débit paraît sain, mais son goodput, c'est-à-dire les requêtes traitées à temps pour être utiles, s'est effondré vers zéro.
- Ajoutez maintenant un délestage à l'entrée : dès que les requêtes en cours atteignent une limite choisie pour que le travail accepté se termine à temps, les nouvelles sont refusées immédiatement avec une erreur 503 (« service indisponible »), ce qui coûte bien moins cher que de les servir. Environ 50 par seconde sont refusées ; les 100 acceptées reçoivent une réponse à temps.
- Un bon délestage est aussi priorisé : un serveur qui traite plusieurs sortes de requêtes refuse les recommandations et les événements analytiques avant les validations de commande. Il abandonne aussi toute requête qui a déjà attendu plus longtemps que le délai d'expiration de son appelant.
- **La leçon :** en cas de surcharge, le vrai choix ne se fait pas entre servir tout le monde et servir une partie des demandes, mais entre en refuser volontairement une partie et ne servir presque personne à temps. Refusez tôt, à moindre coût et par priorité.

Algorithmes de limitation de débit

La limitation de débit (rate limiting) restreint la quantité de travail qu'un client, une clé d'API, un locataire (tenant) ou une adresse IP donnés peuvent soumettre sur une période. Son objectif diffère de celui du délestage de charge : le délestage protège le serveur contre une surcharge

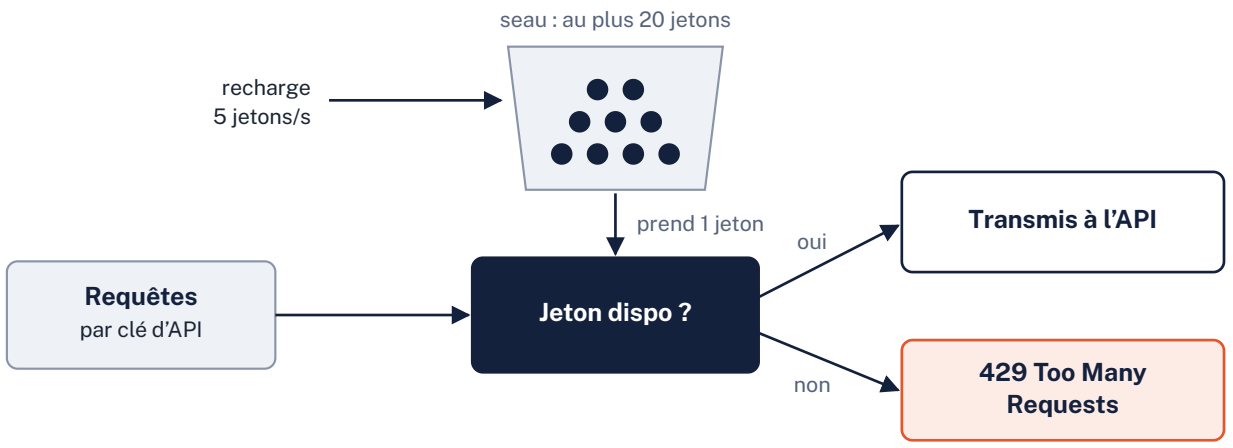
totale, quelle qu'en soit la cause, tandis que la limitation de débit partage équitablement la capacité et protège contre tout client isolé, qu'il soit bogué ou malveillant. Les deux répondent de la même façon, généralement par un HTTP 429. Quatre algorithmes dominent.

Comparaison des algorithmes de limitation de débit

Algorithme	Comment ça marche	Rafales	État par clé	Remarques
Token bucket (seau à jetons)	Des jetons sont ajoutés au rythme r jusqu'à la capacité b ; chaque requête en dépense un	Autorise des rafales jusqu'à b, puis r en moyenne	Deux nombres : les jetons et l'heure de la dernière recharge	Le plus utilisé ; la rafale et le débit soutenu se règlent séparément
Leaky bucket (seau percé)	Les requêtes entrent dans un seau qui se vide à un rythme constant ; quand il est plein, les nouvelles requêtes sont rejetées	La sortie est parfaitement lisse ; les rafales attendent ou sont refusées	Une file, ou un compteur et un horodatage	Utilisé pour la mise en forme du trafic (traffic shaping) ; la limitation des requêtes de NGINX repose sur lui
Compteur à fenêtre fixe	Compter les requêtes par fenêtre calendaire (par minute) ; remise à zéro à la frontière	Jusqu'à deux fois la limite de part et d'autre d'une frontière	Un compteur	Le plus simple ; 100 requêtes à 12:00:59 et 100 à 12:01:00 passent toutes
Journal à fenêtre glissante	Conserver l'horodatage de chaque requête de la dernière fenêtre ; les compter	Exact	Une entrée par requête	Précis mais gourmand en mémoire à haut débit
Compteur à fenêtre glissante	Pondérer le décompte de la fenêtre précédente par la part de celle-ci qui chevauche encore la fenêtre courante, et ajouter le décompte courant	Approximation fidèle	Deux compteurs	À 30 s d'une minute, avec 80 la minute précédente et 40 maintenant : 80 × 0,5 + 40 = 80

Le token bucket (seau à jetons) mérite qu'on s'y attarde, car c'est le choix par défaut habituel. Le seau contient au plus b jetons et en gagne r par seconde. Chaque requête prend un jeton ; s'il n'en reste aucun, elle est rejetée (ou, dans la variante de mise en forme, retardée). Un client resté silencieux peut donc envoyer d'un coup une rafale de b requêtes, ce qui convient aux clients réels, comme une application mobile qui ouvre un écran, alors que sur toute période plus longue il ne peut pas dépasser r en moyenne. Le leaky bucket (seau percé) en est l'image inversée : il lisse la sortie à un rythme constant, ce qui est utile pour protéger un système aval qui ne peut accepter aucune rafale.

Le token bucket



Chaque requête doit prendre un jeton ; le rythme de recharge fixe le débit soutenu et la capacité du seau fixe la plus grande rafale.

EXEMPLE COMMENTÉ

Exemple commenté : un token bucket pour une API partenaire

La clé d'API d'un restaurant partenaire est limitée par un token bucket de capacité b = 20 et de rythme de recharge r = 5 jetons par seconde (300 par minute en moyenne). Le seau est plein à t = 0.

t = 0 : une rafale de 30 requêtes	Les 20 premières prennent les 20 jetons et sont transmises ; 10 sont rejetées avec 429
t = 0 à 1 s	5 nouveaux jetons arrivent (5 par seconde)
t = 1 à 10 s : 8 requêtes par seconde en continu	Environ 5 par seconde sont acceptées et 3 rejetées ; le seau reste presque vide
t = 10 à 14 s : aucune requête	Le seau se recharge à 5 par seconde et est de nouveau plein (20) à t = 14
Borne supérieure sur toute période T	Au plus b + r × T requêtes : sur 60 secondes quelconques, 20 + 5 × 60 = 320, donc la moyenne à long terme ne peut jamais dépasser 5 par seconde
Retry-After pour une requête rejetée	Le jeton suivant arrive dans 1 ÷ r = 0,2 s, mais l'en-tête compte en secondes entières, donc la réponse indique Retry-After: 1

La capacité fixe la rafale qu'un client respectueux des règles peut envoyer après une période calme ; le rythme de recharge fixe le débit soutenu. Choisissez-les séparément.

Un client qui respecte Retry-After, avec un peu de jitter, se stabilise au rythme de recharge ; un client qui réessaie immédiatement ne fait que collectionner davantage de 429.

```
import math, time

class TokenBucket:
    def __init__(self, rate, capacity):
        self.rate = rate              # tokens added per second
        self.capacity = capacity      # largest burst allowed
        self.tokens = capacity        # start full
        self.updated = time.monotonic()

    def allow(self, cost=1):
        now = time.monotonic()
        elapsed = now - self.updated
        self.tokens = min(self.capacity, self.tokens + elapsed * self.rate)
        self.updated = now
        if self.tokens ⩾ cost:
            self.tokens -= cost
            return True, 0
        wait = (cost - self.tokens) / self.rate   # seconds until enough tokens
        return False, max(1, math.ceil(wait))      # for the Retry-After header

bucket = TokenBucket(rate=5, capacity=20)        # one bucket per API key
allowed, retry_after = bucket.allow()
```

Un token bucket à recharge paresseuse : les jetons sont complétés en fonction du temps écoulé depuis le dernier appel, si bien qu'aucun minuteur n'est nécessaire. En production, l'état réside dans un stockage partagé comme Redis, mis à jour de façon atomique.

HTTP 429, Retry-After et où appliquer les limites

HTTP dispose de signaux standard pour les deux situations. `429 Too Many Requests`, défini dans la RFC 6585 (2012), indique à un client qu'il a envoyé trop de requêtes en un temps donné. `503 Service Unavailable` indique que le serveur dans son ensemble est temporairement surchargé ou en maintenance. L'un comme l'autre peut porter un en-tête `Retry-After` (RFC 9110), qui donne un nombre de secondes ou une date après laquelle il est judicieux de réessayer. De nombreuses API renvoient aussi des en-têtes qui annoncent la limite, le quota restant et l'heure de remise à zéro, traditionnellement `X-RateLimit-*` ; un projet de l'IETF est en train de normaliser les champs `RateLimit` et `RateLimit-Policy`. Les clients bien conçus respectent `Retry-After`, ajoutent du jitter pour que les clients rejetés ne reviennent pas tous au même instant, et voient dans des 429 répétés une raison de ralentir plutôt que de réessayer plus fort (section 9.4).

Les limites peuvent être appliquées à plusieurs couches, et les systèmes matures en utilisent plus d'une : à la périphérie ou dans l'API gateway, par adresse IP et par clé d'API, pour arrêter les abus et les clients incontrôlés avant qu'ils ne coûtent quoi que ce soit ; par locataire, pour qu'un gros client ne puisse pas affamer les autres ; et à l'intérieur de chaque service, pour protéger sa propre capacité et ses dépendances. Une limite globale sur de nombreuses

instances de gateway nécessite un état partagé, généralement des compteurs atomiques dans Redis, ce qui ajoute un appel réseau par requête ; un compromis courant consiste à donner à chacune des N instances une limite locale d'environ 1 ÷ N du total et à accepter une certaine imprécision. Les limites doivent être fixées à partir de la capacité mesurée et publiées, pour que les partenaires puissent en tenir compte dans leur conception, et le nombre de rejets doit être surveillé, car une hausse soudaine signale soit une attaque, soit un bug chez un client, soit une limite qui ne correspond plus à la réalité.

Files de lettres mortes

Certains messages échouent à chaque fois : un contenu mal formé, une commande pour un restaurant qui n'existe plus, un bug déclenché par une entrée particulière. Réessayé indéfiniment, un tel message empoisonné bloque une partition ou gaspille un worker pour toujours. Une file de lettres mortes (dead-letter queue, présentée à la section 6.4) reçoit les messages qui ont échoué un nombre défini de fois (SQS appelle cela le nombre maximal de réceptions d'une politique de redrive), pour que le reste du travail puisse continuer.

Une file de lettres mortes est un outil pour les défauts, pas pour la surcharge : si des messages y atterrissent parce que les consommateurs sont trop lents, la solution est la capacité ou le back pressure, pas une seconde file. Chaque message dans une DLQ signifie que quelque chose que le métier attendait ne s'est pas produit ; la DLQ a donc besoin d'un responsable, d'une alerte sur sa profondeur ou son âge, d'outils pour inspecter les messages avec leur erreur et leur nombre de tentatives, et d'un moyen de les rejouer une fois la cause corrigée. Décidez à l'avance de ce que signifie un rejeu tardif pour l'ordre des messages et pour les messages dont le moment métier est passé : un remboursement peut être rejoué avec un jour de retard, une affectation de livreur ne le peut pas.

ATTENTION

Erreurs courantes

- Des files non bornées, en mémoire ou sur disque, qui transforment une surcharge temporaire en plantage ou en heures de travail périmé.
- Surveiller la longueur de la file mais pas l'âge du plus ancien message, si bien que des consommateurs bloqués passent inaperçus alors que la longueur semble normale.
- Délester la charge après avoir effectué la partie coûteuse du travail, si bien qu'un rejet coûte presque autant qu'un succès.
- Des limites de débit à fenêtre fixe qui laissent les clients envoyer deux fois la limite de part et d'autre d'une frontière de fenêtre.
- Des clients qui ignorent 429 et Retry-After et réessaient immédiatement, transformant une limite en déluge.
- Des délais de visibilité plus courts que le temps de traitement, si bien qu'un même message est traité par deux workers à la fois.
- Utiliser une file de lettres mortes comme réservoir de débordement au lieu de corriger la capacité, et la laisser sans responsable ni surveillance.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Des workers traitent 200 messages par seconde et les messages peuvent attendre au plus 30 secondes. Quelle est la plus grande longueur de file acceptable ?

D'après la loi de Little, 200 × 30 = 6000 messages ; une file plus longue signifie que certains messages attendront plus de 30 secondes.

▼ Pourquoi l'âge du plus ancien message est-il une meilleure alarme que la longueur de la file ?

L'âge mesure le délai que subissent les utilisateurs et augmente immédiatement si les consommateurs se bloquent, alors qu'une longueur qui paraît normale peut masquer une file bloquée ou une résorption lente.

▼ Un token bucket a une capacité de 50 et se recharge de 10 jetons par seconde. Combien de requêtes un client peut-il faire dans la première seconde après une longue période calme ?

Environ 60 : les 50 jetons déjà présents dans le seau plus les 10 ajoutés pendant cette seconde.

▼ **Quelle est la différence entre le délestage de charge et la limitation de débit ?**

Le délestage de charge protège le serveur contre une surcharge totale, quelle qu'en soit la cause ; la limitation de débit plafonne ce que chaque client peut envoyer, pour que la capacité soit partagée équitablement.

▼ **À vous : Une vente de billets amène 300 soumissions par seconde pendant deux minutes. Il y a 20 workers, qui prennent chacun 200 ms, et les fans se voient promettre une réponse en moins de deux minutes. Cela tient-il ? Proposez une limite de file d'attente ou un nombre de workers.**

La capacité est de 20 / 0,2 = 100 par seconde : l'arriéré croît donc de 200 par seconde et atteint 24 000 au bout de deux minutes. L'écouler prend encore 240 secondes, ce qui rompt la promesse. Une borne selon la loi de Little est 100 x 120 = 12 000, atteinte après 60 secondes ; au-delà, rejetez rapidement avec 429 et Retry-After. Sinon, 60 workers donnent 300 par seconde sans arriéré, si les limites du paiement et de la base de données le permettent.

SOURCES

- Microsoft Azure Architecture Center: Queue-Based Load Leveling pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling
- Google SRE book: Handling Overload: https://sre.google/sre-book/handling-overload/
- RFC 6585: Additional HTTP Status Codes (429 Too Many Requests): https://www.rfc-editor.org/rfc/rfc6585
- Ben Maurer: Fail at Scale (ACM Queue, 2015): https://queue.acm.org/detail.cfm?id=2839461

Délais d'expiration, nouvelles tentatives, disjoncteurs, cloisonnement et dégradation gracieuse

AVANT DE LIRE

Cette section s'appuie sur :

- 2.4 Performances et scalabilité
- 6.1 Pourquoi distribuer
- 6.5 Communication et API

Termes déjà vus qu'elle utilise : Défaillance partielle, Percentile (p95, p99), Latence de queue (tail latency), Loi de Little, Clé d'idempotence

L'IDÉE EN BREF

Imaginez que vous appelez un fournisseur qui ne décroche jamais. Si vous restez en ligne toute la journée, vous ne faites rien d'autre, et si chaque collègue fait de même, tout le bureau s'arrête. Un bureau bien organisé raccroche après quelques sonneries (un délai d'expiration), rappelle un peu plus tard mais pas indéfiniment (une nouvelle tentative avec une pause), cesse d'appeler un fournisseur manifestement fermé et vérifie de nouveau une heure plus tard (un disjoncteur), veille à ce qu'un seul fournisseur ne puisse pas occuper tous les téléphones (un cloisonnement), et dispose d'un plan B, comme la liste de prix de la veille (une solution de repli). Les services logiciels ont besoin exactement des mêmes habitudes.

CE QU'IL FAUT SAVOIR

- Les défaillances en cascade se produisent quand les appelants attendent une dépendance lente jusqu'à épuiser leurs propres threads et connexions ; chaque point d'intégration a besoin d'une protection délibérée (Nygard, Release It!).
- Fixez des délais d'expiration explicites de connexion et de requête à partir des percentiles de latence mesurés, de plus en plus courts en descendant la chaîne d'appels, et propagez l'échéance restante pour qu'aucun service ne travaille sur une requête abandonnée.
- Ne réessayez que les erreurs transitoires sur des opérations idempotentes, quelques fois, avec un backoff exponentiel plafonné et un jitter complet (Brooker, 2015).
- Les nouvelles tentatives se multiplient d'une couche à l'autre (trois tentatives à trois niveaux donnent 27 appels) ; réessayez à une seule couche et plafonnez les nouvelles tentatives avec un budget, par exemple 10 % des requêtes.

- Un disjoncteur échoue rapidement tant qu'une dépendance est en panne : il passe de l'état fermé à l'état ouvert en cas de défaillances, puis revient à l'état fermé en passant par des appels d'essai à l'état semi-ouvert.
- Le cloisonnement donne à chaque dépendance ou à chaque appelant son propre pool borné, et les solutions de repli maintiennent le parcours principal en état de marche, mais elles doivent être simples, indépendantes de la partie défaillante, testées, et échouer en mode fermé pour la sécurité.
- L'idempotence permet de réessayer, de couvrir et de rejouer sans risque ; les requêtes de couverture réduisent la latence de queue des lectures idempotentes, pour un faible surcroît de charge, budgété.

La défaillance est normale ; sa propagation est facultative

Dans un système distribué, il y a toujours une dépendance lente ou défaillante : un prestataire de paiement passe une mauvaise minute, une base de données bascule sur une autre, un lien réseau perd des paquets, un déploiement tourne mal. La section 6.1 a montré pourquoi ces défaillances partielles ne peuvent pas être évitées et pourquoi un appelant ne peut souvent même pas savoir ce qui s'est passé. La question architecturale n'est donc pas de savoir comment éviter la défaillance, mais comment empêcher une défaillance locale de devenir globale. Une défaillance en cascade est la façon typique dont les systèmes meurent : une dépendance lente ralentit ses appelants, leurs ressources se remplissent de requêtes en attente, ils deviennent à leur tour lents pour leurs propres appelants, et en quelques secondes un problème dans un composant mineur a fait tomber toute la plateforme.

Michael Nygard a catalogué les patrons qui interrompent de telles chaînes dans *Release It!* (2007 ; deuxième édition 2018), en s'appuyant sur des pannes qu'il avait étudiées. Son observation centrale est que les points d'intégration, tous les endroits où un système en appelle un autre, sont la principale source d'instabilité, et que chacun a besoin d'une protection délibérée. Les patrons de cette section (délais d'expiration, échéances, nouvelles tentatives avec backoff, disjoncteurs, cloisonnement, solutions de repli, idempotence et couverture) constituent la boîte à outils standard. Ce sont des tactiques au sens de la section 2.5 : chacune est une décision unique visant la disponibilité, et chacune a un coût.

Délais d'expiration : le premier et le plus important des patrons

Un délai d'expiration (timeout) borne la durée d'attente d'un appelant. Sans lui, un appel à une dépendance qui ne répond jamais monopolise un thread, une connexion et de la mémoire aussi longtemps que le système d'exploitation garde la connexion ouverte, ce qui peut durer de nombreuses minutes. Beaucoup de bibliothèques clientes n'ont pas de délai d'expiration par

défaut (la très répandue bibliothèque Python `requests` attend indéfiniment sauf indication contraire), si bien que l'absence de délai d'expiration est une décision de conception prise par omission. L'exemple commenté ci-dessous en montre la conséquence.

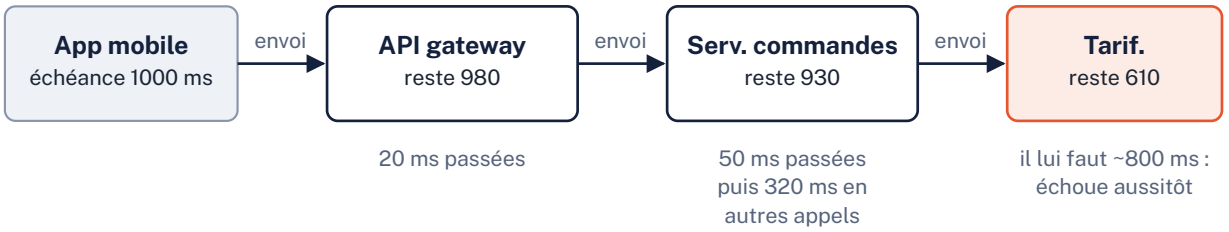
Deux délais d'expiration comptent généralement. Le **délai d'expiration de connexion** borne l'établissement d'une connexion et peut être court, quelques centaines de millisecondes à l'intérieur d'un centre de données, car un serveur sain accepte les connexions presque immédiatement. Le **délai d'expiration de requête** borne l'échange complet et doit venir de la mesure : légèrement au-dessus de la latence de la dépendance à un percentile élevé (p99 ou p99,9) sous charge normale, de sorte qu'il coupe les attentes pathologiques sans faire échouer les requêtes simplement lentes. Il doit aussi tenir dans le propre budget de l'appelant. Si l'API gateway abandonne au bout de 2 secondes mais que le service de commandes attend 10 secondes la tarification, le service de commandes continue de travailler pendant 8 secondes sur une requête que personne n'attend plus. Les délais d'expiration doivent donc raccourcir à mesure que l'on descend dans une chaîne d'appels, et c'est l'idée qui sous-tend les échéances.

Les échéances et leur propagation

Une échéance est un instant absolu auquel toute la requête doit être terminée. La propagation des échéances (deadline propagation) transmet le temps restant à chaque appel aval, de sorte que chaque service sait combien de budget il lui reste au lieu d'appliquer son propre délai d'expiration fixe. gRPC l'intègre nativement : un client fixe une échéance, le temps restant voyage avec l'appel, et un serveur peut vérifier si l'échéance est dépassée et arrêter le travail. Sur HTTP, les équipes propagent un en-tête contenant les millisecondes restantes ou l'échéance absolue, idéalement via le même middleware que celui qui propage le contexte de traçage (section 9.6).

La propagation apporte deux avantages. Un service peut échouer rapidement quand le temps restant est manifestement trop court pour le travail, plutôt que de lancer une requête de base de données qui ne pourra pas se terminer à temps. Et quand l'utilisateur abandonne, le travail en aval s'arrête aussi, au lieu de continuer à consommer de la capacité précisément pendant la surcharge qui a causé le retard. C'est du délestage de charge appliqué aux requêtes mortes, l'une des protections les plus efficaces dont puisse disposer un système sous contrainte.

Une échéance propagée le long d'une chaîne d'appels



20 ms passées

50 ms passées puis 320 ms en autres appels

il lui faut ~800 ms : échoue aussitôt

Chaque saut transmet le temps restant, pas un nouveau délai à lui

Chaque saut transmet le temps restant ; un service qui ne peut pas finir dans le temps restant échoue aussitôt plutôt que de faire un travail que personne n'attendra.

EXEMPLE COMMENTÉ

Exemple commenté : comment un seul délai d'expiration manquant devient une panne totale

Le service de commandes de la plateforme de livraison de repas tourne sur 6 instances de 200 threads de travail chacune : 1200 threads. À midi, il reçoit 400 requêtes par seconde : 100 validations de commande, qui appellent le prestataire de paiement (normalement 300 ms, si bien qu'une validation de commande occupe son thread environ 350 ms au total), et 300 autres requêtes (historique des commandes, panier, suivi) qui prennent 50 ms. Le client de paiement n'a aucun délai d'expiration configuré. À 12:31, le prestataire de paiement cesse de répondre mais garde les connexions ouvertes.

Threads occupés en temps normal (loi de Little)	Validation de commande 100 × 0,35 s = 35 ; autre trafic 300 × 0,05 s = 15 ; total 50 sur 1200 (environ 4 %)
Après 12:31	Chaque validation de commande monopolise désormais son thread indéfiniment : 100 threads de plus se retrouvent bloqués chaque seconde
Délai avant épuisement	(1200 − 15 − 35) ÷ 100 ≈ 11,5 secondes : en 12 secondes environ, tous les threads attendent le prestataire de paiement
Conséquence	L'historique des commandes, le panier et le suivi, qui n'appellent jamais les paiements, font maintenant la queue pour obtenir des threads et expirent au niveau de l'API gateway. Une défaillance touchant 25 % du trafic a mis hors service 100 % du trafic
Et en amont	Les connexions de l'API gateway vers le service de commandes s'accumulent de la même façon ; les contrôles de santé expirent, les instances sont marquées comme non saines et, si la sonde de vivacité est touchée, elles sont redémarrées, à froid, dans les mêmes conditions
Avec un délai d'expiration de 2 secondes	Threads utilisés par la validation de commande : 100 × 2 s = 200 sur l'ensemble du parc (loi de Little). Il reste environ 985 threads pour le reste du trafic : seule la validation de commande échoue, et elle échoue avec une erreur claire au bout de 2 s
Ajout d'un disjoncteur	Dès que, disons, la moitié des 20 derniers appels ont échoué, le disjoncteur s'ouvre : les validations de commande échouent en quelques millisecondes et n'utilisent presque aucun thread ; un appel d'essai toutes les 30 s détecte le rétablissement
Ajout d'un cloisonnement	Limiter les appels de paiement à 40 threads par instance (240 au total) garantit au moins 960 threads pour tout le reste, quels que soient les délais d'expiration définis

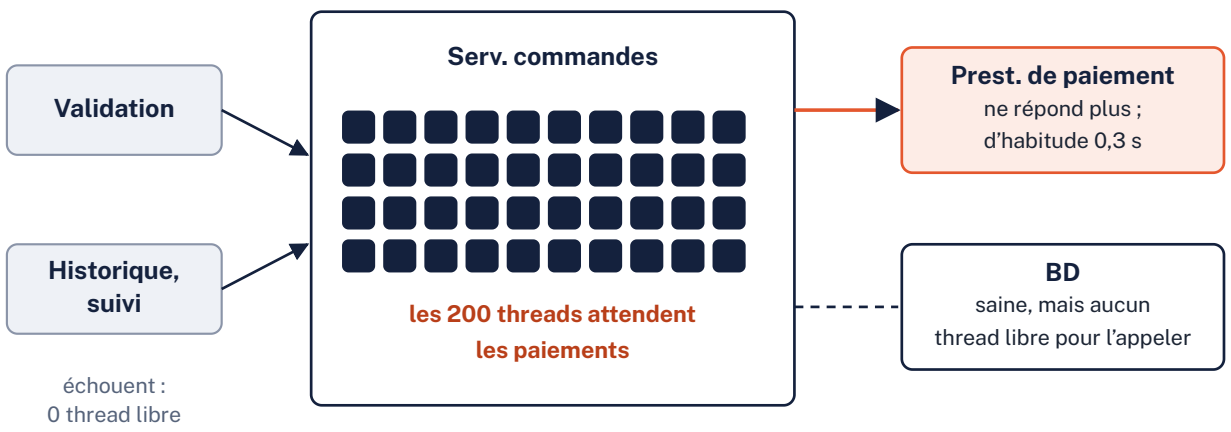
Ajout d'une solution de repli	La validation de commande propose au client un autre moyen de paiement, ou explique que les paiements par carte sont retardés, au lieu d'afficher une erreur générique

Le calcul repose encore sur la loi de Little : threads utilisés = débit de requêtes × durée pendant laquelle chaque requête occupe un thread. Sans délai d'expiration, la durée d'occupation n'est pas bornée, donc le nombre de threads nécessaires ne l'est pas non plus.

Les délais d'expiration limitent la durée ; le cloisonnement limite le nombre ; les disjoncteurs cessent complètement d'appeler. Chacun comble une lacune différente, c'est pourquoi on les utilise ensemble.

Release It! de Nygard décrit de vraies pannes qui ont suivi exactement ce schéma, souvent déclenchées par un point d'intégration que personne ne jugeait critique.

Épuisement du pool de threads



Sans délai d'expiration, un prestataire de paiement lent monopolise tous les threads du service de commandes, si bien que même les requêtes qui ne touchent jamais aux paiements échouent.

Nouvelles tentatives avec backoff exponentiel et jitter

Beaucoup de défaillances sont transitoires : un paquet perdu, une connexion réinitialisée pendant un basculement, un bref 503 renvoyé par une instance surchargée. Réessayer les transforme en succès que l'utilisateur ne voit jamais. Mais les nouvelles tentatives sont aussi de la charge, et des nouvelles tentatives mal conçues comptent parmi les causes de pannes les plus courantes. Quatre règles les rendent sûres. Ne réessayez que les erreurs **transitoires** (délais d'expiration, échecs de connexion, 502, 503, 504, et 429 après le délai indiqué), jamais des erreurs qui se reproduiront, comme 400 ou 404. Ne réessayez que les opérations

idempotentes ou celles qui portent une clé d'idempotence, car une requête qui a expiré a peut-être réussi. Réessayez un **petit nombre** de fois, en général deux ou trois. Et **attendez** entre les tentatives, avec un backoff exponentiel : un délai qui double à chaque tentative, jusqu'à un plafond.

Le backoff seul ne suffit pas. Si mille clients échouent au même moment, comme c'est le cas quand une dépendance a un raté, ils réessaient tous après 100 ms, puis tous après 200 ms, et arrivent en vagues synchronisées qui peuvent maintenir la dépendance à terre. L'article de Marc Brooker *Exponential Backoff And Jitter* (AWS Architecture Blog, 2015) a simulé ce phénomène et montré qu'ajouter de l'aléatoire, le jitter (gigue aléatoire), réduit fortement à la fois le travail total et le temps nécessaire pour terminer. Le «jitter complet» (full jitter) qu'il recommande attend un temps aléatoire compris entre zéro et la borne exponentielle : `sleep = random(0, min(cap, base × 2^attempt))`. Le jitter a sa place partout où de nombreux clients agissent en même temps : nouvelles tentatives, reconnexions, dates d'expiration du cache et tâches planifiées.

```
import random, time

class TransientError(Exception):
    """Timeouts, connection resets, 502, 503 and 504 responses."""

def call_with_retries(fn, attempts=3, base=0.1, cap=2.0):
    for attempt in range(attempts):
        try:
            return fn()
        except TransientError:
            if attempt == attempts - 1:
                raise                       # give up: let the caller decide
            ceiling = min(cap, base * 2 ** attempt)
            time.sleep(random.uniform(0, ceiling))   # full jitter
```

Réessayer une défaillance transitoire avec un backoff exponentiel plafonné et un jitter complet (Brooker, 2015). Seules les opérations idempotentes, ou celles qui portent une clé d'idempotence, doivent être réessayées.

SI VOUS DÉBUTEZ

Pas à pas : pourquoi les nouvelles tentatives ont besoin d'aléatoire

Le point qui fait le plus souvent trébucher ici est que le backoff exponentiel seul n'étale pas les nouvelles tentatives : quand de nombreux appelants échouent au même moment, ils attendent aussi les mêmes durées et reviennent ensemble. C'est l'ajout d'aléatoire, le jitter, qui disperse la foule.

- Supposons que 1000 appels de paiement, chacun avec une clé d'idempotence pour que les nouvelles tentatives soient sans risque, soient en cours quand le prestataire a un bref raté. Les 1000 échouent au même instant.
- Chaque appelant suit un backoff exponentiel simple : attendre 100 ms, réessayer ; en cas d'échec, attendre 200 ms, puis 400 ms. Comme ils ont tous échoué ensemble, les 1000 nouvelles tentatives arrivent ensemble, 100 ms plus tard.
- Un prestataire tout juste en train de se rétablir ne peut pas absorber 1000 appels en un instant : beaucoup échouent donc à nouveau, et la vague suivante arrive d'un bloc 200 ms plus tard. Entre les vagues, le prestataire est au calme ; pendant celles-ci, il est submergé, et les vagues peuvent le maintenir à terre.
- Avec le jitter complet, chaque appelant attend un temps aléatoire compris entre zéro et le délai de backoff. Les premières nouvelles tentatives sont réparties sur les 100 ms : environ 10 par milliseconde au lieu de 1000 d'un coup.
- Les appelants qui échouent à nouveau attendent un temps aléatoire allant jusqu'à 200 ms, puis jusqu'à 400 ms : la charge arrive donc étalée, ce qu'un prestataire en cours de rétablissement absorbe bien plus facilement. Les simulations de Brooker ont montré que cela réduit à la fois le travail total et le temps nécessaire pour terminer.
- Le même attroupement se produit avec tout ce que de nombreuses machines font en même temps, comme se reconnecter, voir des entrées de cache expirer ou lancer des tâches planifiées : le jitter y a donc aussi sa place.
- **La leçon :** le backoff fait attendre les appelants plus longtemps à chaque fois ; le jitter garantit qu'ils n'attendent pas tous le même temps. Attendre poliment ne suffit pas si tout le monde attend au même rythme.

Tempêtes de nouvelles tentatives et budgets de nouvelles tentatives

Les nouvelles tentatives se multiplient d'une couche à l'autre. Supposons qu'un toucher dans l'application mobile passe par l'API gateway, le service de commandes et le client de paiement, et que chacun des trois fasse jusqu'à trois tentatives. Si le prestataire de paiement est en panne, une seule action de l'utilisateur produit 3 × 3 × 3 = 27 appels vers lui, et une panne qui aurait dû se traduire par un taux d'erreur stable devient une tempête de nouvelles tentatives

(retry storm) de 27 fois la charge normale, qui maintient le prestataire à terre même après la correction de la cause initiale. Les utilisateurs qui touchent de nouveau l'écran ajoutent une quatrième couche.

Trois défenses fonctionnent ensemble. D'abord, **ne réessayez qu'à une seule couche**, généralement la plus proche de la dépendance défaillante, et laissez les autres couches faire remonter l'erreur ; le livre SRE de Google recommande qu'un serveur surchargé le signale explicitement, pour que les appelants situés au-dessus ne réessaient pas. Ensuite, utilisez un budget de nouvelles tentatives (retry budget) : le livre SRE décrit le fait de n'autoriser les nouvelles tentatives que tant qu'elles restent sous 10 % des requêtes d'un client, ce qui plafonne la charge supplémentaire due aux nouvelles tentatives à environ 1,1 fois la normale, au lieu de jusqu'à trois fois. Les SDK d'AWS implémentent un quota similaire de jetons de nouvelle tentative, et la politique de nouvelles tentatives de gRPC inclut une régulation (throttling) dans le même but. Enfin, combinez les nouvelles tentatives avec un disjoncteur, pour qu'une dépendance que l'on sait en panne ne soit pas réessayée du tout.

Les défaillances métastables : quand la panne survit à sa cause

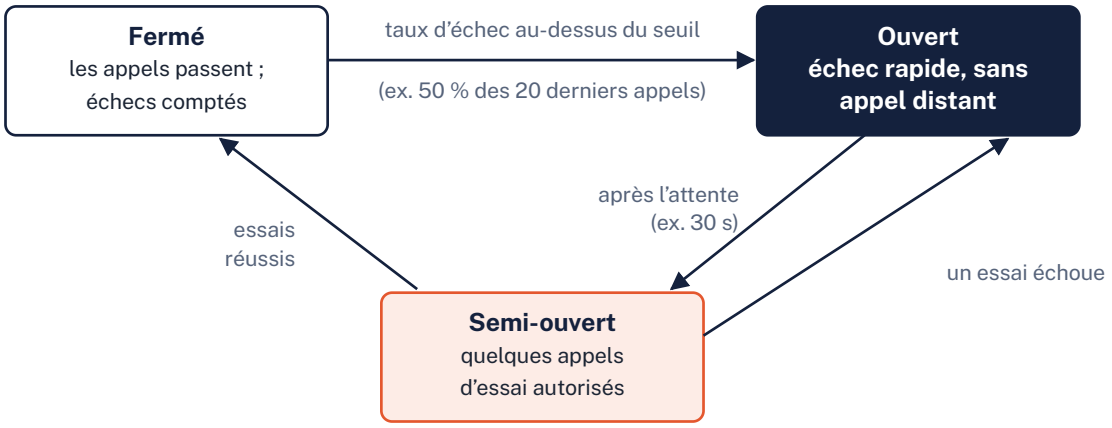
Une tempête de nouvelles tentatives est un exemple d'un schéma plus large. Dans une **défaillance métastable** (Bronson, Aghayev, Charapko et Zhu, HotOS 2021), un déclencheur, comme une brève perturbation de la base de données, fait basculer un système dans un mauvais état qui persiste après la disparition du déclencheur, parce qu'une boucle de rétroaction l'y maintient. Les commandes expirent, les clients réessaient, la base de données reste surchargée, et supprimer la perturbation ne répare rien. Les auteurs considèrent la boucle, et non le déclencheur, comme la cause profonde, et notent qu'elle est souvent construite à partir de fonctions qui améliorent l'efficacité ou la fiabilité, comme les nouvelles tentatives et les caches. Une étude ultérieure (Huang et ses collègues, OSDI 2022) a examiné 22 défaillances de ce type dans 11 organisations. S'en sortir exige en général une action forte : le délestage de charge (section 9.3) ou la modification de la politique de nouvelles tentatives. Testez donc le rétablissement après une surcharge, pas seulement l'arrivée à celle-ci.

Disjoncteurs

Un disjoncteur (circuit breaker), nommé ainsi par Nygard d'après l'appareil électrique, enveloppe les appels à une dépendance et observe leurs résultats. Tant que la dépendance est saine, le disjoncteur est **fermé** et les appels passent, tandis que les défaillances et les expirations de délai sont comptées. Quand les défaillances franchissent un seuil, par exemple la moitié des 20 derniers appels, le disjoncteur **s'ouvre** : pendant une période définie, chaque appel échoue immédiatement sans toucher au réseau, ce qui protège les threads de l'appelant et laisse à la dépendance le temps de se rétablir. Quand la période expire, le disjoncteur

devient **semi-ouvert** et laisse passer quelques appels d'essai. S'ils réussissent, il se ferme ; si l'un d'eux échoue, il s'ouvre de nouveau pour une autre période. La description qu'en a donnée Martin Fowler en 2014 a fait largement connaître le patron, et la bibliothèque Hystrix de Netflix (désormais en mode maintenance) l'a popularisé dans les microservices ; Resilience4j pour Java et Polly pour .NET sont aujourd'hui les bibliothèques courantes, et les service meshes offrent des protections apparentées comme les limites de concurrence et la détection des instances aberrantes (dans Envoy, celles-ci s'appellent circuit breaking, bien qu'elles fonctionnent différemment).

La machine à états du disjoncteur



Fermé compte les défaillances, ouvert échoue rapidement, et semi-ouvert teste le rétablissement avec quelques appels d'essai.

Les réglages méritent réflexion. Qu'est-ce qui compte comme une défaillance ? Les expirations de délai et les réponses 5xx, oui ; les réponses 4xx, généralement non, car elles reflètent la requête de l'appelant plutôt que la santé de la dépendance. Les succès lents peuvent aussi compter, pour qu'une dépendance qui répond en 5 secondes soit traitée comme défaillante. La fenêtre doit être assez grande pour éviter un déclenchement sur deux ou trois appels malchanceux (les bibliothèques exigent un nombre minimal d'appels), et la période d'ouverture assez longue pour permettre le rétablissement, mais assez courte pour ne pas prolonger une panne. Les disjoncteurs doivent être définis par dépendance, et parfois par endpoint, pour qu'un endpoint de reporting défaillant ne coupe pas la prise de commandes. Les nouvelles tentatives doivent traiter un disjoncteur ouvert comme une réponse définitive plutôt que comme quelque chose à réessayer. Enfin, chaque changement d'état doit être journalisé et représenté sur un graphique, car un disjoncteur ouvert est un symptôme que quelqu'un doit voir.

```
import time

class CircuitOpen(Exception):
    pass

class CircuitBreaker:
    def __init__(self, failure_threshold=5, reset_after=30.0):
        self.failure_threshold = failure_threshold  # failures in a row that trip it
        self.reset_after = reset_after              # seconds to stay open
        self.failures, self.state, self.opened_at = 0, "closed", 0.0

    def call(self, fn, *args, **kwargs):
        if self.state == "open":
            if time.monotonic() - self.opened_at < self.reset_after:
                raise CircuitOpen("failing fast")   # no remote call at all
            self.state = "half-open"                # let a trial call through
        try:
            result = fn(*args, **kwargs)
        except Exception:
            self.failures += 1
            if self.state == "half-open" or self.failures ⩾ self.failure_threshold:
                self.state, self.opened_at = "open", time.monotonic()
            raise
        self.failures, self.state = 0, "closed"    # success closes the circuit
        return result

payments = CircuitBreaker()
# payments.call(provider.authorise, order_id, amount, timeout=2.0)
```

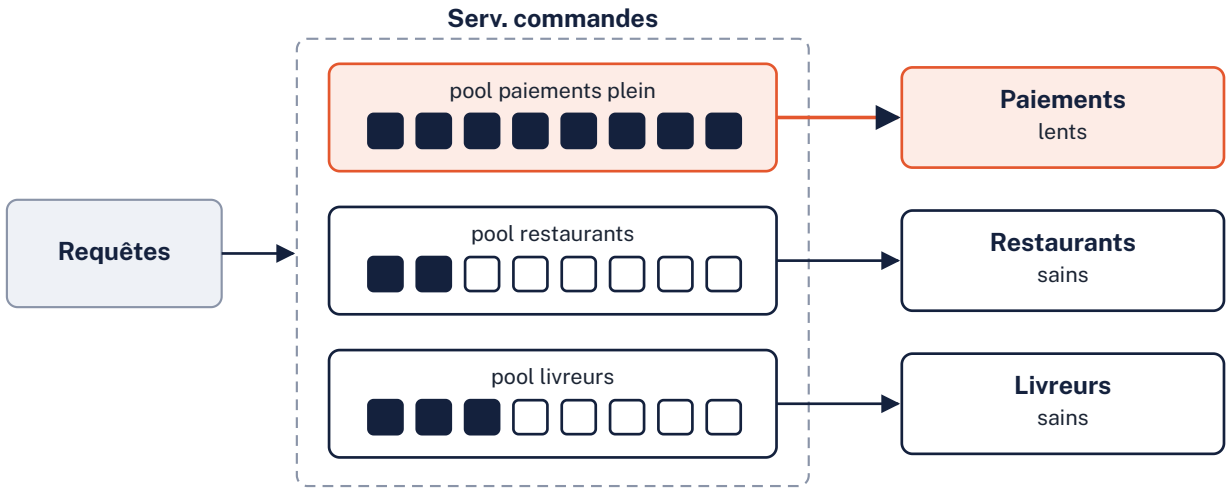
Un disjoncteur minimal. Les bibliothèques de production comme Resilience4j (Java) et Polly (.NET) ajoutent des fenêtres de taux d'échec, la détection des appels lents, des limites sur les appels d'essai simultanés, la sûreté des threads (thread safety) et des métriques.

Cloisonnement

La coque d'un navire est divisée par des cloisons étanches en compartiments, de sorte qu'une brèche inonde un seul compartiment au lieu de couler le navire. En logiciel, un cloisonnement (bulkhead) partitionne les ressources pour qu'une partie défaillante ou surchargée ne puisse pas tout consommer. La forme la plus courante donne à chaque dépendance son propre pool de threads, pool de connexions ou sémaphore borné : les paiements peuvent utiliser au plus 40 appels simultanés par instance, la consultation des restaurants 40, l'affectation des livreurs 40. Quand les paiements ralentissent, leur pool se remplit et les appels de paiement suivants sont rejetés immédiatement, mais les appels vers les restaurants et les livreurs continuent. Un délai d'expiration limite combien de temps chaque appel peut retenir des ressources ; un cloisonnement limite combien d'appels peuvent en retenir à la fois, si bien qu'il protège même quand les délais d'expiration sont trop longs.

Le cloisonnement fonctionne aussi à plus grande échelle. Des déploiements distincts du même service pour différents appelants (un pool d'instances du service de commandes pour l'application grand public, un autre pour l'API des restaurants partenaires) empêchent une intégration partenaire défaillante d'affamer les clients. Les architectures cellulaires, l'axe Z du cube de scalabilité, font de chaque cellule un cloisonnement pour une tranche de clients, et le shuffle sharding, décrit par des ingénieurs d'AWS, attribue à chaque client une petite combinaison aléatoire d'instances, de sorte que le trafic empoisonné d'un client n'affecte que très peu d'autres clients. Le coût de tout cloisonnement est le taux d'utilisation : la capacité réservée à un compartiment ne peut pas être prêtée à un autre, donc les pools doivent être dimensionnés à partir de la demande mesurée.

Le cloisonnement à l'intérieur d'un service



Avec un pool distinct par dépendance, un prestataire de paiement lent ne remplit que le pool des paiements ; les appels aux restaurants et aux livreurs se poursuivent normalement.

Solutions de repli et dégradation gracieuse

Quand un appel échoue ou qu'un disjoncteur est ouvert, l'appelant doit tout de même répondre. Une solution de repli (fallback) décide de ce qu'il renvoie. Les options, de la meilleure à la pire pour l'utilisateur, sont : une **valeur en cache ou périmée** (la dernière carte connue, avec des prix confirmés à l'étape de paiement) ; une **valeur par défaut raisonnable** (un délai de livraison estimé de 30 à 45 minutes au lieu d'une estimation en direct) ; une **fonctionnalité réduite** (des plats populaires au lieu de recommandations personnalisées) ; le **report du travail** (accepter un avis et le publier quand le service d'avis se rétablit) ; et une **erreur claire et rapide** qui indique à l'utilisateur quoi faire. Concevoir ces options fonctionnalité par fonctionnalité, avec le responsable produit, voilà ce que la dégradation gracieuse signifie en pratique : le parcours

principal (parcourir, commander, payer, suivre) continue de fonctionner tandis que les fonctionnalités non essentielles s'effacent. La même réflexion guide le délestage de charge (section 9.3) : sachez à l'avance quelles fonctionnalités désactiver en premier.

Les décisions de sécurité sont l'exception : si le service d'autorisation est injoignable, le repli sûr est de refuser (échouer en mode fermé, fail closed), pas d'autoriser. Les praticiens ne sont d'ailleurs pas d'accord sur les solutions de repli en général. Nygard et les ingénieurs de Netflix les préconisent largement. Des ingénieurs d'Amazon ont soutenu, dans l'article de la Builders' Library *Avoiding fallback in distributed systems*, que les chemins de repli sont rarement exercés, difficiles à tester, et peuvent eux-mêmes déclencher des pannes en faisant basculer un grand parc vers un mode non testé au pire moment ; ils préfèrent rendre le chemin principal plus fiable et échouer rapidement. Les deux positions s'accordent sur la règle pratique : une solution de repli doit être plus simple que ce qu'elle remplace, ne doit pas dépendre du composant défaillant, et doit être exercée régulièrement, par exemple lors de game days (section 9.8), faute de quoi elle ne devrait pas exister.

La stabilité statique au sein d'un service

La section 10.6 a appliqué la stabilité statique aux régions ; l'idée fonctionne à toutes les échelles. AWS sépare le *plan de contrôle*, qui apporte des changements à un système, du *plan de données*, qui effectue le travail quotidien, et dans une conception statiquement stable, le système continue de fonctionner lorsqu'une dépendance est altérée. Pour la plateforme de livraison de repas, cela signifie faire tourner dans chaque zone assez de capacité du service de commandes pour survivre à la perte d'une zone, plutôt que de compter sur l'autoscaling au moment de la panne. Cela signifie aussi conserver localement les dernières règles de routage et les derniers paramètres de restaurants connus comme valides, afin qu'une panne du service de configuration arrête les changements, pas les commandes. La capacité de réserve coûte de l'argent : réservez donc cela au parcours essentiel. Pour chaque solution de repli, demandez-vous ce qui doit changer, et ce qui doit fonctionner, au moment où elle est nécessaire.

Les cache stampedes comme problème de résilience

La section 7.6 présente les défenses contre les ruées : regroupement des requêtes, expiration avec jitter, stale-while-revalidate, rafraîchissement anticipé et préchauffage. La leçon de résilience est qu'un cache au taux de succès élevé est porteur. Bronson et ses collègues notent qu'avec un taux de succès de 90 %, perdre le cache multiplie à peu près par dix les requêtes à la base de données : la base située derrière le cache de la carte peut donc être incapable de survivre sans lui, et la défaillance peut devenir métastable. Trois habitudes en découlent. Gardez une limite de lectures concurrentes de la base, même pour les échecs de cache, à l'aide d'un cloisonnement, car AWS avertit que se rabattre sur une dépendance pendant une longue

panne du cache peut la surcharger. Utilisez une expiration souple et une expiration stricte, afin que les valeurs existantes soient encore servies si un rafraîchissement échoue. Et répétez un démarrage à froid, car les nouveaux serveurs commencent vides.

Les runtimes asynchrones et les pools de threads ne défaillent pas de la même façon

Les cloisonnements ci-dessus comptaient des threads, mais la défaillance dépend du modèle de concurrence. Avec un thread par requête, une dépendance lente épuise le pool, ce qui est pénible mais facile à voir. Les runtimes asynchrones comme Node.js et asyncio de Python servent de nombreux clients avec peu de threads : attendre coûte donc peu, mais le danger se déplace. Un seul callback bloquant ou gourmand en CPU retarde tout le monde sur cette boucle d'événements : la documentation de Python note qu'un calcul d'une seconde retarde toutes les tâches d'une seconde. Et comme rien ne s'épuise, le travail en cours peut croître jusqu'à ce que la mémoire ou un pool de connexions en aval cède. Les threads virtuels de Java conservent le style bloquant, et Oracle indique qu'ils apportent de l'échelle, pas de la vitesse ; ils ne doivent pas être mis en pool, et il faut donc plutôt limiter la concurrence avec des sémaphores. Quel que soit le runtime, fixez des limites explicites par dépendance, gardez le travail lourd hors de la boucle d'événements et mesurez le temps d'attente des tâches avant leur exécution.

Idempotence : ce qui rend sûres les nouvelles tentatives et la couverture

L'idempotence est la propriété selon laquelle faire quelque chose deux fois a le même effet que le faire une seule fois. Tous les patrons de cette section qui répètent du travail en dépendent : les nouvelles tentatives après un délai d'expiration, les requêtes de couverture, la nouvelle livraison de messages après le plantage d'un consommateur, et les rejeux depuis une file de lettres mortes. La section 6.5 a montré comment une clé d'idempotence fait de `POST /orders` une nouvelle tentative sans risque, et la section 6.4 comment les consommateurs idempotents dédupliquent les messages. La règle architecturale est simple : avant d'ajouter une nouvelle tentative, vérifiez que l'opération est idempotente et, si elle ne l'est pas, ajoutez d'abord une clé ou une détection des doublons. Une politique de nouvelles tentatives sur un appel de paiement non idempotent n'est pas un patron de résilience ; c'est un générateur de doubles débits.

Requêtes de couverture

Les nouvelles tentatives traitent les défaillances ; les requêtes de couverture (hedged requests) traitent la lenteur. La section 2.4 les a présentées à partir de l'article *The Tail at Scale* de Dean et Barroso (2013) : envoyez une requête à une réplique et, si elle n'a pas répondu dans un délai égal, par exemple, à la latence du 95e percentile, envoyez une copie à une seconde

réplique et utilisez la première réponse reçue, en annulant l'autre. Comme seuls les quelques pour cent de requêtes les plus lentes sont dupliqués, la charge supplémentaire est faible, tandis que la queue se réduit de façon spectaculaire, comme l'a montré le banc d'essai cité à la section 2.4.

La couverture a des préconditions. L'opération doit être idempotente, normalement une lecture. Il doit y avoir des répliques indépendantes, pour que la seconde copie ait peu de chances d'être lente pour la même raison. Les requêtes supplémentaires doivent être budgétées et désactivées en cas de surcharge, car quand toutes les répliques sont lentes, la couverture double la charge d'un système déjà en difficulté, exactement comme une nouvelle tentative non budgétée. Utilisée avec soin sur des chemins en fan-out à forte intensité de lecture, comme la recherche et la consultation des cartes, c'est l'un des outils les plus efficaces contre la latence de queue.

Les patrons en un coup d'œil

Patron	Protège contre	Principal coût ou risque
Délai d'expiration (timeout)	Attente indéfinie ; ressources retenues par des appels morts	Trop court, il fait échouer des appels sains mais lents ; trop long, il ne protège rien
Propagation des échéances (deadline propagation)	Travail qui se poursuit après l'abandon de l'appelant	Nécessite une prise en charge dans chaque service et chaque bibliothèque du chemin
Nouvelle tentative avec backoff et jitter	Défaillances transitoires	Charge supplémentaire ; doublons sauf idempotence
Budget de nouvelles tentatives (retry budget)	Tempêtes de nouvelles tentatives qui multiplient la charge	Certaines défaillances transitoires ne sont pas réessayées pendant les mauvaises périodes
Disjoncteur (circuit breaker)	Appels répétés à une dépendance en panne	Réglage ; peut couper une dépendance qui n'est que partiellement défaillante
Cloisonnement (bulkhead)	Une dépendance ou un appelant qui consomme toutes les ressources	La capacité réservée ne peut pas être partagée
Repli et dégradation	Défaillance totale d'une fonctionnalité	Chemins de code non testés ; résultats périmés ou approximatifs
Requête de couverture (hedged request)	Latence de queue due aux répliques lentes	Charge supplémentaire ; lectures uniquement

Liste de contrôle : chaque appel distant

Passez en revue chaque point d'intégration, y compris les bases de données, les caches, les courtiers et les API tierces, à l'aide de cette liste.

- [ ] **Les délais d'expiration de connexion et de requête sont définis explicitement**
  Choisis à partir des percentiles élevés mesurés de la dépendance et de la propre échéance de l'appelant, jamais laissés à la valeur par défaut d'une bibliothèque.
- [ ] **L'échéance restante est propagée**
  Les appels aval reçoivent le temps restant, et le travail s'arrête quand celui-ci est écoulé.
- [ ] **Les nouvelles tentatives sont limitées et sûres**
  Uniquement les erreurs transitoires, uniquement les opérations idempotentes ou celles qui ont une clé d'idempotence, à une seule couche, avec un backoff exponentiel plafonné et du jitter.
- [ ] **Un budget de nouvelles tentatives plafonne la charge supplémentaire**
  Les nouvelles tentatives restent une petite fraction du trafic (par exemple 10 %), si bien qu'une panne ne peut pas multiplier la charge.
- [ ] **Un disjoncteur protège la dépendance**
  Les seuils, le temps d'attente et ce qui compte comme une défaillance sont réglés, et son état est exporté sous forme de métrique.
- [ ] **La concurrence vers la dépendance est bornée**
  Un cloisonnement (pool dédié ou sémaphore) empêche cette dépendance de consommer tous les threads ou toutes les connexions.
- [ ] **Le comportement en cas de défaillance est décidé**
  Une solution de repli, une réponse dégradée ou une erreur claire, convenue avec le responsable produit, et un échec en mode fermé pour les décisions de sécurité.
- [ ] **Les délais d'expiration, nouvelles tentatives, déclenchements du disjoncteur et replis sont observables**
  Comptés, représentés sur des graphiques et assortis d'alertes, pour qu'une dégradation silencieuse soit remarquée (section 9.6).
- [ ] **Le chemin de défaillance est testé**
  L'injection de fautes ou un game day montre que les délais d'expiration, le disjoncteur et la solution de repli fonctionnent réellement (section 9.8).

ATTENTION

Erreurs courantes

- Laisser les délais d'expiration aux valeurs par défaut des bibliothèques, qui sont souvent infinies ou de plusieurs minutes.
- Des délais d'expiration en aval plus longs qu'en amont, si bien que le travail continue après le départ de l'appelant.
- Réessayer à chaque couche, réessayer des opérations non idempotentes, ou réessayer sans jitter.
- Compter les réponses 4xx comme des défaillances dans un disjoncteur, si bien qu'un seul client bogué le déclenche pour tout le monde.
- Un seul pool de threads ou de connexions partagé par toutes les dépendances, si bien que la plus lente décide pour toutes.
- Des solutions de repli jamais exercées, qui échouent ou surchargent autre chose la première fois qu'on en a besoin.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Un service reçoit 50 requêtes par seconde, chacune appelant une dépendance avec un délai d'expiration de 4 secondes. Si la dépendance se bloque, combien de threads ces appels peuvent-ils monopoliser ?

D'après la loi de Little, environ 50 × 4 = 200 threads ; c'est pourquoi le délai d'expiration et la limite du cloisonnement doivent être compatibles avec la taille du pool.

▼ Pourquoi le jitter complet est-il meilleur qu'un simple backoff exponentiel quand de nombreux clients échouent en même temps ?

Un backoff simple fait réessayer tous les clients aux mêmes instants, en vagues synchronisées ; des délais aléatoires étalent les nouvelles tentatives, si bien que la dépendance voit une charge régulière et plus faible.

▼ Dans quel état un disjoncteur ne fait-il aucun appel distant, et qu'est-ce qui met fin à cet état ?

Dans l'état ouvert ; quand la période d'attente expire, il devient semi-ouvert et laisse passer des appels d'essai.

▼ Pourquoi un contrôle d'autorisation ne doit-il pas se replier sur l'autorisation de l'accès quand le service d'autorisation est en panne ?

Les décisions de sécurité doivent échouer en mode fermé : autoriser l'accès pendant une panne laisserait entrer n'importe qui chaque fois que le service tomberait en panne ou serait attaqué.

▼ À vous : La passerelle et le service de commandes font chacun jusqu'à 4 tentatives pour l'appel de paiement, un POST sans clé d'idempotence ni délai d'expiration (timeout). Le prestataire ralentit. Quel est le pire cas, et quels trois changements feriez-vous ?

Chaque requête entrante peut provoquer 4 x 4 = 16 appels de paiement, ce qui ajoute de la charge à un prestataire en difficulté et pourrait débiter les clients deux fois. Les threads attendent aussi indéfiniment. Premièrement, fixez un délai d'expiration à partir de percentiles mesurés, et transmettez l'échéance restante. Deuxièmement, ne faites de nouvelles tentatives qu'à une seule couche, avec un backoff plafonné, du jitter et un budget de nouvelles tentatives, et uniquement avec une clé d'idempotence. Troisièmement, ajoutez un disjoncteur et un cloisonnement, afin que la défaillance du paiement ne puisse pas épuiser les threads des autres fonctionnalités.

SOURCES

- Marc Brooker: Exponential Backoff And Jitter (AWS Architecture Blog, 2015): https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/
- Martin Fowler: CircuitBreaker: https://martinfowler.com/bliki/CircuitBreaker.html
- Google SRE book: Addressing Cascading Failures: https://sre.google/sre-book/addressing-cascading-failures/
- Microsoft Azure Architecture Center: Bulkhead pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/bulkhead
- Dean and Barroso: The Tail at Scale (Google Research): https://research.google/pubs/pub40801/
- Amazon Builders' Library: Avoiding fallback in distributed systems: https://aws.amazon.com/builders-library/avoiding-fallback-in-distributed-systems/
- Metastable Failures in Distributed Systems (HotOS 2021): https://sigops.org/s/conferences/hotos/2021/papers/hotos21-s11-bronson.pdf
- Metastable Failures in the Wild (OSDI 2022): https://www.usenix.org/conference/osdi22/presentation/huang-lexiang
- Metastable Failures in Distributed Systems (Marc Brooker's commentary): https://brooker.co.za/blog/2021/05/24/metastable/

- Static stability using Availability Zones (Amazon Builders' Library): https://builder.aws.com/content/3K9SiqiJ369TGkkGzqPM95BuYWI/static-stability-using-availability-zones
- Caching challenges and strategies (Amazon Builders' Library): https://builder.aws.com/content/3K9TeXp6strMiJ4vhakpRD7BQWO/caching-challenges-and-strategies
- Developing with asyncio (Python documentation): https://docs.python.org/3/library/asyncio-dev.html
- Don't Block the Event Loop (or the Worker Pool) (Node.js): https://nodejs.org/en/learn/asynchronous-work/dont-block-the-event-loop
- Virtual Threads (Oracle Java 21 documentation): https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html

SLI, SLO, SLA et budgets d'erreur

AVANT DE LIRE

Cette section s'appuie sur :

- 2.4 Performances et scalabilité
- 2.5 Disponibilité et fiabilité

Termes déjà vus qu'elle utilise : Disponibilité, Fiabilité, Percentile (p95, p99), Latence

L'IDÉE EN BREF

Imaginez une compagnie de bus qui promet « presque tous les bus à l'heure ». Pour que cette promesse soit utile, elle doit préciser ce qui compte comme à l'heure (à deux minutes près), comment elle le mesure (à chaque arrêt) et ce que signifie « presque tous » (99 sur 100 sur un mois). Si elle n'y parvient pas, l'équipe chargée des horaires cesse d'essayer de nouvelles lignes et corrige d'abord la ponctualité. Comme la perfection est impossible et très coûteuse, le bus en retard sur cent est une marge qu'elle peut dépenser pour des travaux de voirie et de nouvelles lignes. La fiabilité des logiciels fonctionne exactement ainsi.

CE QU'IL FAUT SAVOIR

- Un SLI est une mesure centrée sur l'utilisateur, de préférence exprimée en bons événements divisés par les événements valides ; un SLO est un objectif pour cette mesure sur une fenêtre de temps ; un SLA est un contrat assorti de conséquences, fixé moins strictement que le SLO.
- Les SLI utiles couvrent la disponibilité, la latence et la qualité pour les services pilotés par les requêtes, la fraîcheur, l'exactitude et la couverture pour les pipelines, et la durabilité pour le stockage, mesurées aussi près de l'utilisateur que possible en pratique.
- Le budget d'erreur est égal à un moins le SLO : 99,9 % sur 30 jours autorisent 43,2 minutes d'indisponibilité totale ou 0,1 % des requêtes, et une politique de budget d'erreur écrite indique ce qui se passe quand il est épuisé.
- Le taux de consommation est le ratio d'erreurs observé divisé par le ratio autorisé ; le budget dépensé est égal au taux de consommation multiplié par la fenêtre d'alerte, divisé par la fenêtre du SLO : un taux de consommation de 14,4 pendant une heure dépense donc 2 % d'un budget sur 30 jours.

- Les alertes multifenêtres et multi-taux de consommation du SRE Workbook alertent l'astreinte à 2 % du budget en une heure ou à 5 % en six heures, et ouvrent un ticket à 10 % en trois jours, chacune étant confirmée par une fenêtre courte douze fois plus brève.
- Les erreurs courantes sont les objectifs de 100 %, un trop grand nombre de SLO, des SLI que les utilisateurs ne perçoivent pas, une mesure prise au mauvais endroit, et des SLO sans politique ni alertes derrière eux.

Trois termes, une seule idée

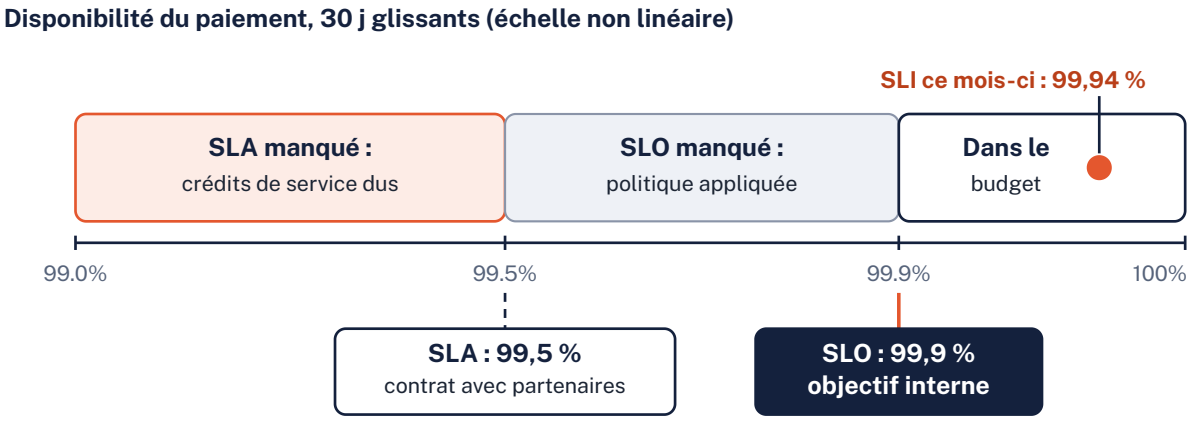
Le chapitre 2 a présenté les objectifs de disponibilité et les « neuf » (section 2.5) et a montré que 99,9 % sur 30 jours laissent 43,2 minutes. Cette section transforme ce calcul en discipline d'exploitation. Le vocabulaire vient du livre de Google *Site Reliability Engineering* (Beyer, Jones, Petoff et Murphy, O'Reilly, 2016) et de son ouvrage compagnon, *The Site Reliability Workbook* (2018), et il est aujourd'hui utilisé bien au-delà de Google.

Un indicateur de niveau de service (SLI) est une mesure quantitative, soigneusement définie, d'un aspect du service que reçoivent les utilisateurs. La forme la plus utile est un ratio : les bons événements divisés par les événements valides. «La proportion de requêtes de paiement qui renvoient une réponse réussie en moins de 800 ms, mesurée au niveau de l'équilibreur de charge (load balancer)» est un SLI. Il va de 0 % (rien ne fonctionne) à 100 % (rien n'est cassé), et la forme de ratio permet de lire et de comparer chaque SLI, et d'alerter sur lui, de la même manière.

Un objectif de niveau de service (SLO) est une cible pour un SLI sur une fenêtre donnée : «99,9 % des requêtes de paiement sont bonnes, mesuré sur 30 jours glissants.» C'est une promesse interne sur laquelle s'accordent l'équipe, le product owner et les personnes qui dépendent du service. Un accord de niveau de service (SLA) est un contrat avec les clients qui fixe des conséquences, généralement des crédits de service, si un niveau n'est pas atteint. Beaucoup de services ont des SLO mais pas de SLA ; les plateformes internes n'ont presque jamais de SLA.

L'idée qui relie les trois est le budget d'erreur. Si l'objectif est de 99,9 %, alors 0,1 % des événements peuvent être mauvais sans que personne ait failli. Ces 0,1 % ne sont pas du gaspillage ; c'est la marge dont dispose l'équipe pour prendre des risques : livrer des fonctionnalités, mener des migrations, expérimenter sur la configuration. Le budget transforme un débat sur des impressions («le service est-il assez fiable ?») en débat sur des chiffres («avons-nous dépensé notre budget ?»), et donne au produit et à l'ingénierie une règle commune pour savoir quand aller vite et quand ralentir.

SLI, SLO et SLA sur une même échelle



Le SLI est la mesure, le SLO l'objectif interne et le SLA un plancher contractuel moins strict ; l'écart entre le SLO et le SLA est la zone d'alerte.

Choisir de bons SLI

Un SLI n'est utile que dans la mesure où il est proche de ce que vivent les utilisateurs. Le Workbook propose une courte liste qui couvre la plupart des services. Les services pilotés par les requêtes (API, pages web) ont besoin de la **disponibilité** (la proportion de requêtes valides servies avec succès), de la **latence** (la proportion servie plus vite qu'un seuil) et parfois de la **qualité** (la proportion servie sans dégradation, par exemple une carte affichée sans recommandations personnalisées parce que cette dépendance était en panne). Les pipelines de données ont besoin de la **fraîcheur** (la proportion de données mises à jour plus récemment qu'un seuil), de l'**exactitude** (la proportion de résultats corrects) et de la **couverture** (la proportion des entrées traitées). Les systèmes de stockage ajoutent la **durabilité** (la proportion des données écrites qui peuvent encore être lues).

Les SLI de latence s'écrivent en proportions, pas en moyennes : «99 % des requêtes sur la carte servies en moins de 300 ms». Cela renvoie directement aux percentiles de la section 2.4 : dire que 99 % des requêtes doivent prendre moins de 300 ms revient à dire que le p99 doit être inférieur à 300 ms, mais la forme en proportion s'additionne proprement dans le temps et entre serveurs (on compte les bonnes requêtes et les requêtes valides), ce qui n'est pas le cas des percentiles. Il est courant de fixer deux seuils de latence, par exemple 90 % sous 200 ms et 99 % sous 800 ms, pour saisir à la fois l'expérience typique et celle de la queue.

L'endroit où vous mesurez compte autant que ce que vous mesurez. Le Workbook distingue la *spécification du SLI* (le résultat qui compte pour l'utilisateur, comme «commandes passées avec succès») de l'*implémentation du SLI* (la manière de le compter : journaux du serveur, métriques de l'équilibreur de charge, sondes synthétiques ou instrumentation dans l'app mobile). Une mesure prise à l'intérieur du serveur d'applications ne voit pas les requêtes qui ne l'ont jamais

atteint parce que le DNS, le réseau de diffusion de contenu ou l'équilibreur de charge a été défaillant. Une mesure prise dans le client voit tout, mais elle inclut les mauvaises connexions mobiles des utilisateurs et arrive en retard. La plupart des équipes commencent par les métriques de l'équilibreur de charge, peu coûteuses et proches de la périphérie, et ajoutent des sondes synthétiques pour les chemins à faible trafic.

Définissez précisément ce qui compte comme *valide* et ce qui compte comme *bon*, car ce sont les cas limites qui décident si le chiffre a un sens. Une 404 pour une URL mal saisie est-elle mauvaise ? En général non : c'est une réponse correcte. Une réponse 429 de limitation de débit est-elle mauvaise ? Tout dépend si le client respectait ou non son quota convenu. Les contrôles de santé (health checks) et les tests de charge sont-ils des événements valides ? Non ; excluez-les, sinon ils diluent le ratio. Consignez ces règles par écrit à côté du SLO.

Des SLI pour les exemples fil rouge

Parcours	Type de SLI	Spécification du SLI (bons / valides)	Exemple de SLO
Livraison de repas : parcourir les cartes	Latence	Requêtes sur la carte servies en moins de 300 ms / toutes les requêtes valides sur la carte	99 % sur 30 jours
Livraison de repas : passer commande	Disponibilité	Requêtes de commande servies avec succès ou avec un refus métier correct / toutes les requêtes de commande valides	99,9 % sur 30 jours
Livraison de repas : position du livreur	Fraîcheur	Affichages du plan de suivi où la position du livreur date de moins de 30 s / tous les affichages du plan de suivi	99 % sur 30 jours
Livraison de repas : versements aux restaurants	Exactitude	Lots de versements rapprochés sans écart / tous les lots de versements	99,99 % sur un trimestre
Billetterie : paiement à l'ouverture des ventes	Latence et disponibilité	Blocages de places confirmés en moins de 2 s / toutes les tentatives de blocage des fans admis	99,5 % pendant chaque ouverture des ventes
Billetterie : envoi des billets électroniques	Fraîcheur	Billets dans l'app du fan moins de 5 minutes après le paiement / toutes les commandes payées	99,9 % sur 30 jours

Fixer l'objectif

Un objectif est une décision métier, pas une décision technique. Le coût de chaque « neuf » supplémentaire augmente fortement, tandis qu'au-delà d'un certain point les utilisateurs ne remarquent plus les améliorations, parce que leur propre réseau mobile, leur Wi-Fi et leurs appareils tombent en panne plus souvent que cela. Le livre SRE de Google le dit sans détour : viser 100 % est une erreur pour presque tout, car cela ne laisse aucun budget pour le changement, coûte une fortune et ne peut de toute façon pas être atteint. Un objectif de 100 % transforme aussi chaque défaillance isolée en manquement à la politique, ce qui apprend aux gens à ignorer le SLO.

Les bons points de départ sont les performances passées (qu'a obtenu le service au dernier trimestre, et les utilisateurs étaient-ils satisfaits ?), les objectifs des services dont vous dépendez (vous ne pouvez pas promettre 99,99 % au-dessus d'une base de données à 99,9 %, comme la section 6.1 l'a montré pour les chaînes d'appels) et le coût métier du manque de fiabilité. Pour la plateforme de livraison de repas, les commandes échouées à l'heure du déjeuner coûtent du chiffre d'affaires et la confiance des clients : 99,9 % sur la prise de commande se défend donc ; les mises à jour de la position des livreurs sont moins critiques et 99 % suffisent. Commencez un peu en dessous des performances actuelles, pour que l'équipe ne soit pas immédiatement en infraction, puis resserrez l'objectif plus tard, preuves à l'appui.

Choisissez la fenêtre délibérément. Une fenêtre glissante comme les 28 derniers jours correspond à la façon dont les utilisateurs se souviennent des pannes (ils n'oublient pas le premier du mois), et 28 jours contiennent exactement quatre fois chaque jour de la semaine, ce qui garde les cycles hebdomadaires équitables ; cette section utilise 30 jours, car les chiffres publiés de taux de consommation reposent sur cette durée, et les calculs fonctionnent de la même façon pour 28. Les mois calendaires conviennent aux rapports et aux contrats. Les services organisés autour d'événements, comme une ouverture de billetterie, peuvent nécessiter un SLO par événement, car une heure de défaillance pendant l'ouverture des ventes compte bien plus qu'une semaine calme.

Les budgets d'erreur en chiffres

Le budget d'erreur peut se compter en temps ou en événements. Les budgets fondés sur le temps conviennent aux services qui sont soit en marche, soit à l'arrêt ; les budgets fondés sur les requêtes sont plus justes pour les services au trafic irrégulier, car une panne à 03:00 touche moins d'utilisateurs qu'une panne à midi.

EXEMPLE COMMENTÉ

Exemple commenté : le budget de la prise de commande

L'API de commande de la plateforme de livraison de repas a un SLO de 99,9 % de bonnes requêtes sur 30 jours. Elle traite environ 3 millions de requêtes de commande valides sur cette période, fortement concentrées au déjeuner et au dîner.

Budget en fraction	1 - 0,999 = 0,001, soit 0,1 %
Budget en temps	0,001 × 30 jours × 1440 minutes = 43,2 minutes d'indisponibilité totale
Budget en requêtes	0,001 × 3 000 000 = 3000 mauvaises requêtes
Incident 1 : mauvais déploiement à 15:00	12 minutes × 800 requêtes par minute = 9600 requêtes ; 40 % en échec = 3840 mauvaises requêtes
Résultat	3840, c'est plus que tout le budget de 3000 requêtes : un seul incident épuise le mois
Le même incident à 03:00	12 × 50 requêtes par minute × 0,4 = 240 mauvaises requêtes, soit 8 % du budget

Les budgets fondés sur les requêtes pondèrent les défaillances selon le nombre d'utilisateurs touchés : c'est pourquoi les mêmes 12 minutes peuvent coûter 8 % du budget ou plus de 100 %.

La vue fondée sur le temps (43,2 minutes) aurait compté les deux incidents de la même façon, à 12 × 0,4 = 4,8 minutes d'indisponibilité équivalente, soit environ 11 % du budget.

Les politiques de budget d'erreur

Un budget ne veut rien dire si rien ne se passe quand il est épuisé. Une politique de budget d'erreur est un court document, convenu à l'avance par l'ingénierie, le produit et la direction, qui dit ce que l'équipe fera. L'exemple de politique du Workbook mesure le budget sur les quatre dernières semaines ; si le service dépasse son budget, tous les changements et toutes les mises en production s'arrêtent, sauf les correctifs urgents (problèmes P0) et les correctifs de sécurité, jusqu'à ce que le service revienne dans son SLO. Elle exige aussi un post-mortem avec au moins une action de priorité maximale chaque fois qu'un seul incident consomme plus de 20 % du budget sur quatre semaines, et fait remonter les désaccords au directeur technique (CTO).

La valeur d'une politique tient au fait qu'elle est convenue avant de s'appliquer. En pleine course à la mise en production, personne ne veut geler les changements ; avec une politique signée, le gel est une règle convenue d'avance plutôt qu'un affrontement entre un SRE et un chef de produit. Les bonnes politiques précisent aussi ce qui n'est *pas* imputé à l'équipe : une panne réseau à l'échelle de l'entreprise, une dépendance elle-même gelée, ou du trafic hors du périmètre du SLO, comme un test de charge.

Les praticiens ne s'accordent pas sur le degré d'automatisme que doit avoir la politique. Certaines équipes l'appliquent strictement, car les exceptions érodent la confiance dans tout le système. D'autres traitent le budget comme une information qui alimente une discussion, en faisant valoir qu'un gel strict peut retarder les correctifs mêmes qui rétabliraient la fiabilité. Une voie intermédiaire est courante : geler automatiquement les mises en production de fonctionnalités risquées, continuer à livrer les correctifs de fiabilité, et permettre à un dirigeant désigné d'accorder des exceptions documentées. Le cas inverse compte aussi : une équipe qui ne dépense jamais son budget est probablement trop prudente et pourrait livrer plus vite, mener plus d'expériences ou réduire ses coûts.

Taux de consommation et alertes sur les SLO

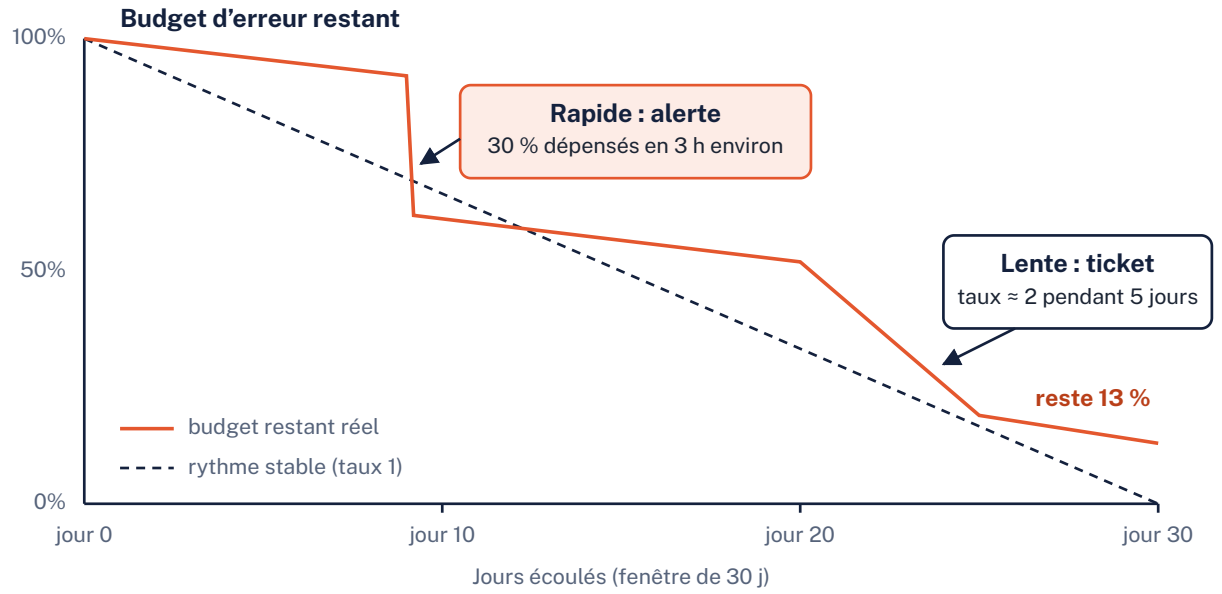
Alerter sur les taux d'erreur bruts («alerter l'astreinte si plus de 1 % des requêtes échouent pendant 5 minutes») génère du bruit : de brefs pics réveillent des gens la nuit pour des problèmes qui ne menaceraient jamais le SLO, tandis qu'un taux d'échec constant de 0,5 % peut épuiser discrètement le mois sans jamais franchir le seuil. La réponse du Workbook consiste à alerter sur le taux de consommation (burn rate) : la vitesse à laquelle le budget est dépensé, comparée au rythme qui l'épuiserait exactement à la fin de la fenêtre.

Taux de consommation = ratio d'erreurs observé / ratio d'erreurs autorisé. Avec un SLO de 99,9 %, le ratio autorisé est de 0,1 % : un ratio d'erreurs de 0,1 % correspond donc à un taux de consommation de 1 (le budget dure exactement la fenêtre), 1 % à un taux de 10, et 100 % (une panne totale) à un taux de 1000. La fraction du budget de la fenêtre dépensée pendant une fenêtre d'alerte vaut taux de consommation x fenêtre d'alerte / fenêtre du SLO. Pour une fenêtre de 30 jours (720 heures), un taux de consommation de 14,4 maintenu pendant une heure dépense 14,4 × 1 / 720 = 0,02, soit 2 % du budget du mois ; à ce rythme, le budget entier serait épuisé en 720 / 14,4 = 50 heures.

Le Workbook recommande une alerte multifenêtre et multi-taux de consommation comme réglage par défaut raisonnable pour un SLO de 99,9 % : alerter l'astreinte si 2 % du budget sont dépensés en une heure (taux de consommation 14,4) ou 5 % en six heures (taux 6), et ouvrir un ticket si 10 % sont dépensés en trois jours (taux 1). Chaque condition doit être vérifiée sur une fenêtre longue et sur une fenêtre courte, la fenêtre courte valant un douzième de la longue (respectivement 5 minutes, 30 minutes et 6 heures). La fenêtre longue garantit que le

problème est significatif ; la fenêtre courte garantit qu'il se produit encore, si bien que l'alerte cesse de se déclencher quelques minutes après un correctif au lieu de rester rouge pendant une heure.

Consommation rapide et consommation lente



Une consommation rapide dépense une grande part du budget en quelques heures et doit réveiller quelqu'un ; une consommation lente le dépense en plusieurs jours et relève d'un ticket.

```
SLO = 0.999                # 99.9% of requests good
PERIOD_H = 30 * 24         # 720-hour SLO window

def burn_rate(bad, total, slo=SLO):
    """Observed error ratio divided by the ratio the SLO allows."""
    if total == 0:
        return 0.0
    return (bad / total) / (1 - slo)

def budget_spent(rate, window_h, period_h=PERIOD_H):
    """Fraction of the whole window's budget used in window_h hours."""
    return rate * window_h / period_h

def page(w1h, w5m, w6h, w30m):
    """Each argument is (bad, total) for that window."""
    fast = burn_rate(*w1h) >= 14.4 and burn_rate(*w5m) >= 14.4
    slow = burn_rate(*w6h) >= 6 and burn_rate(*w30m) >= 6
    return fast or slow

print(round(burn_rate(bad=288, total=20_000), 1))   # 14.4
print(budget_spent(14.4, window_h=1))               # 0.02, i.e. 2%
```

Taux de consommation, budget dépensé et règle d'alerte d'astreinte à deux fenêtres. En production, la même logique s'écrit sous forme de règles d'alerte dans le système de supervision (par exemple les règles d'enregistrement et d'alerte de Prometheus).

Les seuils fixent aussi le délai de détection. Si l'API de commande tombe complètement en panne (ratio d'erreurs de 100 %, taux de consommation de 1000), le ratio d'erreurs sur une heure dépasse 1,44 % au bout de 0,0144 × 60 = environ 0,9 minute : l'alerte d'astreinte part donc en moins d'une minute. Si 2 % des requêtes échouent (taux de consommation de 20), le ratio sur une heure atteint 1,44 % au bout de 0,72 × 60 = environ 43 minutes. Les problèmes graves déclenchent l'alerte vite ; les problèmes légers ne la déclenchent que s'ils persistent, ce qui est exactement le comportement souhaité à trois heures du matin.

Les services à faible trafic mettent ce calcul en défaut : avec 10 requêtes par heure, un seul échec représente un ratio d'erreurs de 10 % et un taux de consommation de 100. Les remèdes du Workbook sont le trafic synthétique pour ajouter du signal, le regroupement de services apparentés à faible trafic sous un même SLO, la réduction de l'impact des échecs isolés grâce aux nouvelles tentatives et aux solutions de repli, ou l'acceptation d'une fenêtre plus longue. Pour le système de billetterie, presque inactif entre deux ouvertures des ventes puis extrêmement sollicité, des SLO par événement avec des alertes sur le taux de consommation pendant la fenêtre de l'événement fonctionnent mieux qu'un chiffre mensuel.

Les SLA et les crédits de service

Un SLA fait passer l'objectif de l'ingénierie au droit des contrats. Il définit le service couvert, la manière de mesurer la disponibilité, les exclusions (maintenance planifiée, mauvaise utilisation par le client, force majeure), la compensation et la manière dont un client la réclame. La compensation habituelle est un crédit de service : un pourcentage de la redevance mensuelle restitué, échelonné selon l'ampleur du manquement. Les crédits sont généralement faibles par rapport à la perte réelle du client : il vaut donc mieux lire un SLA comme un signal de la confiance du fournisseur que comme une assurance ; les clients dont les pertes seraient importantes ont besoin de leur propre résilience, pas d'un crédit plus élevé.

Prenons, à titre d'illustration, un SLA que la plateforme de livraison de repas pourrait proposer à ses restaurants partenaires pour son API de prise de commandes : 99,5 % de disponibilité mensuelle, avec un crédit de 10 % sur la commission de plateforme du mois en dessous de 99,5 % et de 25 % en dessous de 99,0 %. Le SLO interne est de 99,9 %. L'écart entre les deux est délibéré : l'équipe doit épuiser son budget de SLO, appliquer sa politique et se rétablir bien avant que le contrat soit menacé. Ne fixez jamais le SLO au même niveau que le SLA : vous n'apprendriez l'existence d'un problème qu'au moment où vous devez déjà de l'argent.

Deux conséquences architecturales en découlent. D'abord, un SLA ne peut être honnête que si vous connaissez les SLA de vos propres dépendances (fournisseur cloud, prestataire de paiement, fournisseur de cartographie) et la façon dont ils se combinent. Ensuite, les SLA exigent des preuves : la méthode de mesure inscrite au contrat doit pouvoir être appliquée à partir de votre propre télémétrie, ce qui constitue une exigence pour la conception de l'observabilité.

Erreurs courantes

Trop de SLO. Un service doté de trente SLO n'en a aucun, car personne ne peut raisonner sur eux ni agir en conséquence. Commencez par deux ou trois par parcours utilisateur critique : en général un SLO de disponibilité et un ou deux SLO de latence. **Mesurer la mauvaise chose.** Le taux d'utilisation du CPU, la profondeur des files d'attente et le retard de réplication sont des causes utiles à surveiller, mais ce ne sont pas des SLI, car les utilisateurs ne les vivent pas directement. **Mesurer au mauvais endroit.** Les taux de réussite côté serveur paraissent parfaits alors que le CDN renvoie des erreurs. **Faire des moyennes.** Un SLO sur la latence moyenne masque la queue lente dont se plaignent les utilisateurs.

Des objectifs sans conséquences. Un SLO sans politique de budget ni alertes sur le taux de consommation n'est qu'une décoration de tableau de bord. **Des objectifs fixés par les seuls ingénieurs.** Le product owner doit donner son accord, car le SLO arbitre entre fonctionnalités et fiabilité. **Ne jamais réviser.** Réexaminez les SLO tous les trimestres ou tous les deux

trimestres : si les utilisateurs se plaignent alors que le SLO est au vert, c'est le SLI ou l'objectif qui est faux ; si le budget n'est jamais entamé, l'objectif est peut-être trop lâche ou l'équipe trop prudente. **Un seul SLO pour des parcours très différents.** Regrouper la consultation de la carte et le paiement dans un même ratio permet aux nombreuses requêtes bon marché de masquer les échecs des requêtes rares et précieuses.

Les SLO se rattachent au reste de ce chapitre. Les alertes sur le taux de consommation sont les alertes fondées sur les symptômes que recommande la partie consacrée à l'observabilité ; les incidents qui consomment le budget alimentent les revues post-incident ; et la mesure de la réponse d'un SLO est une fonction d'aptitude naturelle pour l'architecture.

SI VOUS DÉBUTEZ

Si vous débutez

Voyez le budget d'erreur comme un argent de poche mensuel réservé aux erreurs. Le SLI est le relevé bancaire, le SLO est l'argent de poche convenu avec votre famille, et le SLA est le contrat de prêt avec la banque, qui vous facture des frais si vous êtes largement à découvert. Dépenser votre argent de poche n'a rien de grave : c'est fait pour ça. L'épuiser signifie que vous cessez d'acheter des petits plaisirs (de nouvelles fonctionnalités) jusqu'au mois suivant.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi un SLI s'écrit-il généralement sous la forme bons événements divisés par événements valides ?

Parce que le ratio est toujours compris entre 0 % et 100 %, s'additionne proprement entre serveurs et dans le temps par simple comptage, et permet de lire, de comparer et de transformer chaque SLI en budget d'erreur de la même manière.

▼ Un SLO de 99,9 % sur 30 jours subit un taux de consommation de 6 pendant six heures. Quelle fraction du budget est dépensée ?

6 × 6 / 720 = 0,05, soit 5 % du budget sur 30 jours.

▼ Pourquoi un SLA devrait-il être moins strict que le SLO interne ?

Pour que l'équipe détecte un budget d'erreur épuisé et y réagisse bien avant que le contrat soit enfreint et que des crédits de service deviennent exigibles.

▼ **Qu’apporte la fenêtre courte dans une alerte multifenêtre ?**

Elle confirme que le problème se produit toujours, si bien que l’alerte se réinitialise quelques minutes après un correctif au lieu de rester active pendant toute la fenêtre longue.

▼ **À vous : Un SLO est de 99,9 % sur 30 jours, pour 2 millions de requêtes. Sur 3 heures de trafic régulier, 1 % des requêtes échouent. Trouvez le budget en mauvaises requêtes et en minutes, le taux de consommation (burn rate) et la part du budget utilisée.**

Budget : 0,1 % de 2 000 000 = 2000 requêtes en échec, ou 43,2 minutes de panne totale. Taux de consommation : 1 % / 0,1 % = 10. Budget consommé : 10 x 3 heures / 720 heures = environ 4,2 %. Un taux de 10 n’atteint pas le seuil d’alerte rapide de 14,4, mais l’alerte « 5 % en six heures » se déclencherait s’il persistait, et cela mérite largement qu’on s’y intéresse. La politique de budget d’erreur décide ensuite de la suite.

SOURCES

- Google SRE book: Service Level Objectives (2016): https://sre.google/sre-book/service-level-objectives/
- The Site Reliability Workbook: Implementing SLOs (2018): https://sre.google/workbook/implementing-slos/
- The Site Reliability Workbook: Alerting on SLOs: https://sre.google/workbook/alerting-on-slos/
- The Site Reliability Workbook: Example error budget policy: https://sre.google/workbook/error-budget-policy/

Observabilité : journaux, métriques et traces

AVANT DE LIRE

Cette section s'appuie sur :

- 2.4 Performances et scalabilité
- 6.1 Pourquoi distribuer
- 9.5 SLO et budgets d'erreur

Termes déjà vus qu'elle utilise : Percentile (p95, p99), Objectif de niveau de service (SLO), Indicateur de niveau de service (SLI), Taux d'utilisation, Données à caractère personnel

L'IDÉE EN BREF

Le tableau de bord d'une voiture affiche quelques cadrans et voyants : vitesse, carburant, température du moteur. C'est la supervision : elle vous informe des problèmes que quelqu'un a prévus. Quand la voiture fait un bruit étrange qu'aucun voyant n'explique, un mécanicien branche un ordinateur de diagnostic et lit des enregistrements détaillés de ce que faisait chaque pièce, afin de trouver une panne que personne n'avait anticipée. L'observabilité consiste à construire le logiciel de façon que ce type d'investigation soit toujours possible : chaque partie tient des enregistrements clairs, et les enregistrements d'un même parcours peuvent être suivis d'une partie à l'autre.

CE QU'IL FAUT SAVOIR

- La supervision vérifie des conditions prédéfinies ; l'observabilité est la capacité d'expliquer un comportement nouveau et imprévu à partir de la télémétrie existante, et elle doit être intégrée dès la conception grâce à une instrumentation cohérente et à des identifiants partagés.
- Les métriques sont des agrégats peu coûteux pour les tendances et les alertes, les journaux structurés enregistrent des événements individuels, les traces montrent le chemin d'une requête à travers les services, et les profils montrent où le code consomme des ressources.
- Les compteurs, les jauges et les histogrammes sont les types de métriques de base ; les percentiles ne peuvent pas être moyennés entre serveurs ni dans le temps, mais les compartiments (buckets) des histogrammes peuvent être additionnés, et le percentile lu sur le résultat fusionné.

- RED (rate, errors, duration : débit, erreurs, durée) décrit les services, USE (utilisation, saturation, errors : taux d'utilisation, saturation, erreurs) décrit les ressources, et les quatre signaux d'or sont la latence, le trafic, les erreurs et la saturation.
- Le traçage distribué (Dapper, 2010) relie les spans grâce à la propagation du contexte, normalisée par l'en-tête W3C traceparent ; OpenTelemetry, projet de la CNCF parvenu au stade « graduated » en 2026, est le standard neutre vis-à-vis des fournisseurs pour produire traces, métriques et journaux.
- Des valeurs d'étiquettes non bornées font exploser la cardinalité des métriques et leur coût : le détail par utilisateur ou par commande a donc sa place dans les traces et les journaux, et l'échantillonnage en queue (tail-based sampling) conserve les traces qui comptent.
- Alertez l'astreinte sur les symptômes que ressentent les utilisateurs, comme les taux de consommation des SLO, et non sur des causes comme le CPU, et concevez chaque tableau de bord pour répondre à une seule question pour un seul public.

Supervision et observabilité

La supervision (monitoring) consiste à collecter des mesures et des vérifications prédéfinies et à déclencher une alarme quand elles franchissent un seuil : disque plein à 90 %, taux d'erreur supérieur à 1 %, contrôle de santé en échec. Elle répond à des questions que vous saviez devoir poser à l'avance. L'observabilité est une propriété du système : la mesure dans laquelle vous pouvez comprendre son état interne, y compris des états que personne n'avait prévus, à partir de la télémétrie qu'il émet déjà, sans déployer de nouveau code pour le découvrir. Le mot vient de la théorie de la commande, où Rudolf Kálmán l'a défini en 1960 comme la capacité de déduire l'état interne d'un système à partir de ses sorties.

La distinction compte, car les systèmes distribués tombent surtout en panne de façons nouvelles. Une commande de livraison de repas peut être lente parce que l'app tablette d'un restaurant multiplie les nouvelles tentatives de façon agressive, parce que l'endpoint d'un prestataire de paiement est dégradé dans une région, ou parce qu'une nouvelle fonctionnalité de la carte rend une requête de base de données lente, uniquement pour les restaurants qui ont plus de 500 articles. Personne n'a créé de tableau de bord pour aucun de ces cas. La supervision vous dit *que* le paiement est lent ; l'observabilité vous permet de demander *pour qui, où et pourquoi*, en découpant la télémétrie selon n'importe quel attribut jusqu'à ce que le motif apparaisse.

Les praticiens ne s'accordent pas sur la portée de cette distinction. Une vision, répandue chez les éditeurs d'outils, présente l'observabilité comme un parapluie couvrant trois piliers : journaux, métriques et traces. Charity Majors, Liz Fong-Jones et George Miranda (*Observability Engineering*, O'Reilly, 2022) soutiennent que ce cadrage en piliers passe à côté de l'essentiel, et

que l'observabilité exige des événements riches, structurés et à forte cardinalité, que l'on peut interroger selon des combinaisons arbitraires. Les deux camps s'accordent sur la conséquence architecturale : l'observabilité est un attribut de qualité qui doit être intégré dès la conception, grâce à une instrumentation cohérente, à des identifiants partagés et à la propagation du contexte, car on ne peut pas l'ajouter après coup pendant un incident.

Les types de télémétrie

Signal	Ce que c'est	Idéal pour	Principal coût ou limite
Métriques	Séries temporelles numériques, agrégées par intervalle et étiquetées selon quelques dimensions	Tendances à long terme peu coûteuses, tableaux de bord, alertes sur les SLI	Pré-agrégées : ne peuvent pas répondre à des questions sur des requêtes individuelles ; les étiquettes doivent être à faible cardinalité
Journaux	Enregistrements horodatés d'événements discrets, idéalement structurés	Le détail d'un événement : une erreur, une décision, une entrée d'audit	Le volume et le coût croissent avec le trafic ; le texte libre est difficile à interroger
Traces	Arbres de spans chronométrés montrant le chemin d'une requête à travers les services	Trouver où passe le temps et quel saut a échoué dans un appel distribué	Doivent être échantillonnées à grande échelle ; inutiles si un saut perd le contexte
Profils	Échantillons montrant où le code consomme du temps CPU ou alloue de la mémoire	Expliquer pourquoi un service est lent ou coûteux au niveau du code	Outillage plus récent ; le profilage continu ajoute un léger surcoût
Événements larges (wide events)	Un enregistrement richement doté d'attributs par requête et par service	Découpage ad hoc selon n'importe quel attribut (client, version, région)	Nécessitent un stockage conçu pour les requêtes à forte cardinalité

Journalisation structurée (structured logging)

Une ligne de journal comme `Payment failed for order 81F3 after 3001ms` est facile à écrire et difficile à exploiter : chaque recherche nécessite une expression régulière fragile, et chaque développeur formule les choses différemment. La journalisation structurée (structured logging) écrit chaque entrée sous forme de champs clé-valeur, généralement en JSON, avec des noms convenus : horodatage, niveau, service, nom de l'événement, identifiants métier (commande, restaurant, livreur), résultat, durée, ainsi que les identifiants de trace et de span

qui relient la ligne à sa trace. Les recherches deviennent précises («tous les événements card_authorisation_failed pour le prestataire cardco au cours de la dernière heure, regroupés par http_status»), et les journaux de différents services peuvent être joints.

Quelques règles gardent les journaux utiles et abordables. Journalisez des événements, pas de la prose ; utilisez un nom d'événement fixe pour chaque type d'événement. Utilisez les niveaux de façon cohérente, et ne journalisez pas au niveau erreur des situations qui ne demandent aucune action. Ne journalisez jamais de secrets, de jetons, de numéros de carte complets ni de données personnelles inutiles ; les pipelines de télémétrie sont copiés, envoyés à des tiers et conservés pendant des mois : ils sont donc une source courante de fuites de données, et les données personnelles présentes dans les journaux relèvent du RGPD comme toute autre copie (section 8.7). La journalisation et la détection de sécurité, un objectif voisin mais différent, avec ses propres besoins de conservation et d'intégrité, sont traitées à la section 8.8.

```
traceparent: 00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01
             version-trace id (16 bytes)-parent span id (8 bytes)-flags (01 = sampled)

{"ts": "2026-10-22T12:31:07.412Z", "level": "error",
 "service": "payment-service", "event": "card_authorisation_failed",
 "order_id": "ord_81F3", "provider": "cardco", "http_status": 504,
 "duration_ms": 3001, "attempt": 2,
 "trace_id": "4bf92f3577b34da6a3ce929d0e0e4736", "span_id": "00f067aa0ba902b7"}
```

Un en-tête W3C traceparent et une entrée de journal structurée du service de paiement. Le trace_id partagé permet à une recherche de passer de cette ligne de journal à la trace entière.

Les métriques, et pourquoi les percentiles ne se moyennent pas

La plupart des systèmes de métriques, comme Prometheus (projet « graduated » de la CNCF) et le modèle de métriques d'OpenTelemetry, proposent les mêmes quelques types. Un compteur (counter) ne fait qu'augmenter, comme le nombre total de requêtes servies ; vous le lisez sous forme de taux (requêtes par seconde sur les cinq dernières minutes), ce qui résiste aussi aux redémarrages. Une gauge (jauge) est une valeur courante qui monte et descend : longueur d'une file d'attente, connexions utilisées, mémoire. Un histogramme répartit les observations dans des compartiments, ou buckets (par exemple requêtes de moins de 50 ms, de moins de 100 ms, de moins de 250 ms, etc.), avec en plus un nombre total et une somme, à partir desquels on estime les percentiles. Prometheus a aussi un type summary, qui calcule des quantiles à l'intérieur de chaque processus.

La distinction entre histogrammes et summaries n'est pas un détail. Comme l'a noté la section 2.4, un percentile est une propriété d'une distribution entière, et le percentile d'une distribution combinée ne peut pas être calculé à partir des percentiles de ses parties. Faire la moyenne de valeurs de p99 entre serveurs, ou entre minutes, produit un nombre qui n'est le percentile de

rien. Les nombres par compartiment, en revanche, peuvent simplement être additionnés : additionnez chaque compartiment sur tous les serveurs, puis estimez le percentile à partir de l'histogramme fusionné. C'est pourquoi les SLI de latence à l'échelle d'un parc doivent reposer sur des histogrammes (ou sur des nombres de requêtes sous un seuil, comme dans la section 9.5), jamais sur des summaries par instance.

EXEMPLE COMMENTÉ

Exemple commenté : faire la moyenne de deux p99

Le service de la carte tourne sur deux instances, qui traitent chacune 10 000 requêtes sur un intervalle de cinq minutes. L'instance A est saine : toutes ses requêtes prennent entre 20 et 300 ms, et son p99 est de 180 ms. L'instance B a un voisin bruyant : 7000 requêtes prennent entre 20 et 300 ms, mais 3000 sont réparties uniformément entre 500 et 1100 ms.

p99 de B (ses 100 plus lentes sur 10 000)	1100 - 600 × (100 / 3000) = 1080 ms
Moyenne des deux p99	(180 + 1080) / 2 = 630 ms
Vrai p99 des 20 000 requêtes	Les 200 plus lentes sont toutes sur B : 1100 - 600 × (200 / 3000) = 1060 ms
Erreur due à la moyenne	630 ms affichées contre 1060 ms vécues : le tableau de bord sous-estime la queue de 430 ms
Avec des histogrammes	Additionnez les nombres par compartiment de A et de B, puis lisez le 99e percentile sur les compartiments fusionnés : environ 1060 ms, avec une précision limitée seulement par la largeur des compartiments

Les percentiles moyennés peuvent être faux dans les deux sens, et l'erreur augmente quand les instances reçoivent des trafics différents.

Il en va de même dans le temps : le p99 sur une heure n'est pas la moyenne de douze p99 sur cinq minutes.

Que mesurer : RED, USE et les signaux d'or

Trois courtes listes de contrôle couvrent l'essentiel de ce qu'un service doit exposer. La méthode RED, proposée par Tom Wilkie en 2015, s'applique à tout ce qui sert des requêtes : **Rate** (le débit, en requêtes par seconde), **Errors** (les requêtes en échec par seconde) et **Duration** (la distribution des temps de réponse). La méthode USE, de Brendan Gregg, s'applique aux ressources : pour chaque CPU, disque, lien réseau, pool de connexions et file

d'attente, vérifiez l'**Utilisation** (son degré d'occupation), la **Saturation** (la quantité de travail en attente) et les **Errors** (erreurs). Les quatre signaux d'or (four golden signals) du chapitre du livre SRE de Google consacré à la supervision des systèmes distribués sont la **latence**, le **trafic**, les **erreurs** et la **saturation** : en pratique, RED plus la saturation.

En pratique, RED décrit ce que vivent les utilisateurs et USE explique pourquoi. Si la durée du service de paiement augmente (RED), USE appliqué à ses ressources peut révéler un pool de connexions utilisé à 100 % avec 40 requêtes en attente (saturation), ce qui désigne à son tour le prestataire de cartes lent qui garde des connexions ouvertes. Mesurez la latence séparément pour les réussites et pour les échecs : une erreur rapide n'est pas une bonne latence, et un délai d'expiration (timeout) est à la fois une erreur et une réponse lente.

Trois listes de contrôle comparées

Liste de contrôle	S'applique à	Signaux	Question typique à laquelle elle répond
RED (Wilkie)	Services pilotés par les requêtes	Débit, erreurs, durée	Le service de commandes sert-il bien les utilisateurs en ce moment ?
USE (Gregg)	Ressources : CPU, mémoire, disques, pools, files d'attente	Taux d'utilisation, saturation, erreurs	Quelle ressource est le goulot d'étranglement ?
Quatre signaux d'or (livre SRE)	Systèmes en contact avec les utilisateurs	Latence, trafic, erreurs, saturation	Le système est-il en bonne santé, et à quelle distance est-il de ses limites ?

Traçage distribué (distributed tracing)

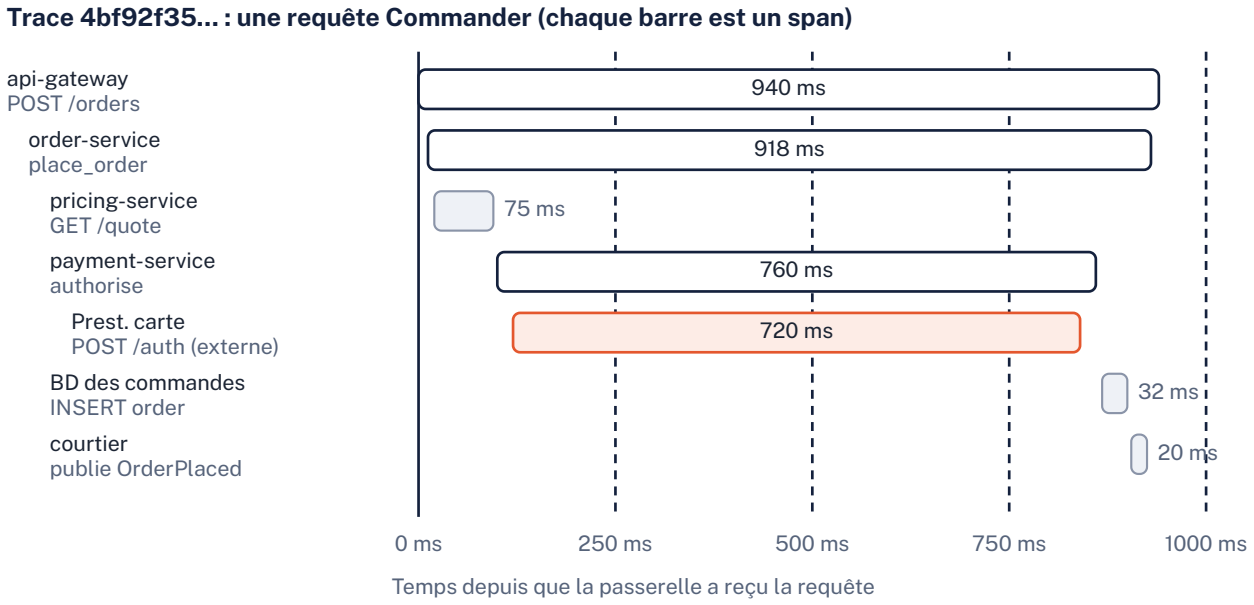
Dans un monolithe, une pile d'appels (stack trace) montre où une requête lente a passé son temps. Entre services, il n'y a pas de pile unique. Le traçage distribué (distributed tracing) en reconstruit une : chaque requête reçoit un identifiant de trace à la périphérie, chaque unité de travail devient un span (avec un nom, une heure de début, une durée, un statut, des attributs et l'identifiant de son span parent), et un backend de traçage assemble les spans en arbre. L'article de Google sur Dapper (Sigelman et ses collègues, 2010) a décrit l'approche à grande échelle : un faible surcoût, une instrumentation dans des bibliothèques partagées plutôt que dans le code applicatif, et un échantillonnage pour que seule une fraction des requêtes soit enregistrée.

Le traçage dépend entièrement de la propagation du contexte : chaque service doit transmettre le contexte de trace au suivant. La spécification W3C Trace Context (recommandation du W3C depuis 2020) la normalise pour HTTP avec un en-tête `traceparent` qui contient une version,

un ID de trace de 16 octets, l'ID de 8 octets du span appelant et des indicateurs, par exemple si la trace est échantillonnée, ainsi qu'un en-tête `tracestate` facultatif pour les données propres aux fournisseurs. Pour la messagerie, les mêmes valeurs circulent dans les en-têtes des messages, et le span du consommateur fait référence à celui du producteur. Un seul service qui perd l'en-tête (une vieille bibliothèque, un client HTTP écrit à la main, un traitement par lots) casse toutes les traces qui passent par lui : c'est pourquoi la propagation a sa place dans les bibliothèques partagées de la plateforme, et non dans le code de chaque équipe.

Enregistrer chaque span de chaque requête coûte cher à grande échelle, si bien que les traces sont échantillonnées. L'échantillonnage en tête (head-based sampling) décide au début de la requête (en garder 1 sur 100) et coûte peu, mais il écarte la plupart des requêtes lentes ou en échec, rares, qui vous intéressent. L'échantillonnage en queue (tail-based sampling) met les spans en mémoire tampon jusqu'à la fin de la requête, puis conserve toutes les erreurs, toutes les requêtes plus lentes qu'un seuil et une petite fraction du reste ; il demande plus de mémoire dans le collecteur, mais garde les traces intéressantes.

Une trace à travers les services de livraison de repas



La cascade montre d'emblée que 720 des 940 ms sont passées à attendre le prestataire de cartes externe, et non dans les propres services de la plateforme.

SI VOUS DÉBUTEZ

Pas à pas : suivre une commande à travers les services

L'idée la plus difficile de cette section est la façon dont une trace est assemblée à partir de morceaux enregistrés par différents services, dont aucun ne voit la requête entière. Le liant, ce sont deux identifiants transmis avec chaque appel.

- Un client passe une commande. L'API gateway reçoit POST /orders, attribue à la requête un nouvel ID de trace et démarre son propre span : un enregistrement avec un nom, une heure de début et, une fois terminé, une durée.
- Quand la passerelle appelle le service de commandes, elle ajoute un en-tête traceparent qui porte l'ID de trace et l'ID de son propre span. Le service de commandes démarre son span place_order et enregistre le span de la passerelle comme parent.
- Le service de commandes fait de même pour chaque appel qu'il émet, vers la tarification et vers le paiement, et place le contexte de trace dans les en-têtes de l'événement OrderPlaced. Chaque nouveau span enregistre l'ID de trace et son parent.
- Chaque service envoie séparément ses spans terminés au backend de traçage. L'ID de trace partagé permet au backend de les rassembler, et les liens vers les parents lui permettent de les disposer dans la cascade du diagramme.
- Supposons maintenant que le service de commandes appelle la tarification au moyen d'un client HTTP écrit à la main qui n'ajoute pas l'en-tête. La tarification démarrerait alors sa propre nouvelle trace, et ses spans manqueraient dans la cascade de la commande.
- C'est pourquoi la propagation du contexte a sa place dans des bibliothèques partagées qu'utilisent toutes les équipes : un seul saut qui l'oublie casse toutes les traces qui passent par lui.
- **La leçon :** chaque service n'enregistre que sa propre partie d'une requête. L'ID de trace et les liens vers les parents transmis à chaque appel font de ces parties une seule histoire : une trace n'est donc jamais plus complète que son saut le plus faible.

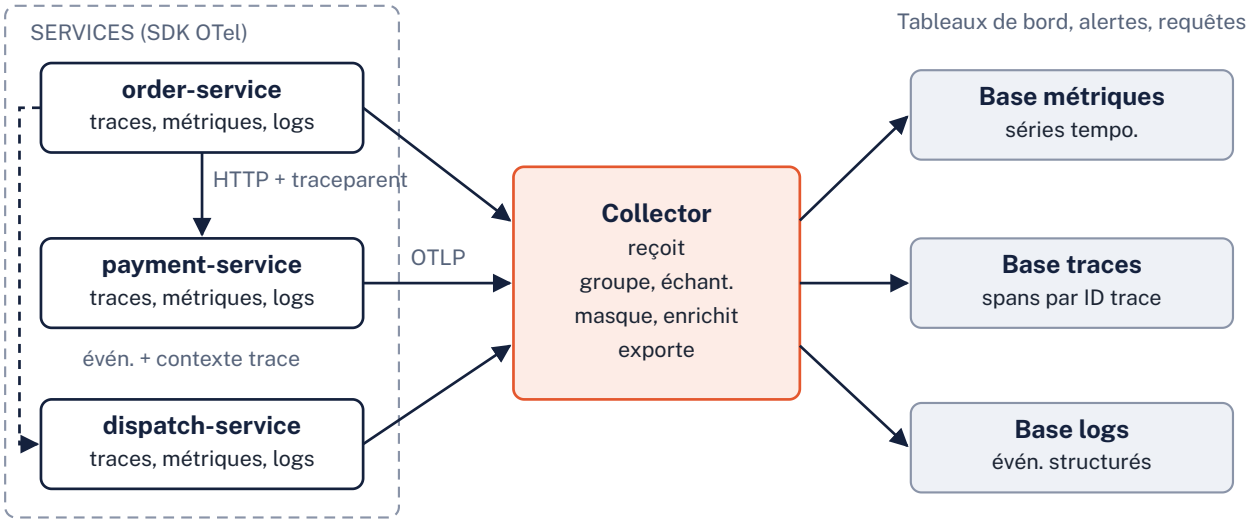
OpenTelemetry : un standard unique pour l'instrumentation

Jusque vers 2019, chaque fournisseur de traçage et de supervision avait ses propres agents et bibliothèques, si bien que changer de fournisseur obligeait à réinstrumenter chaque service. OpenTelemetry (OTel) est né en 2019 de la fusion de deux projets open source antérieurs, OpenTracing et OpenCensus, sous l'égide de la Cloud Native Computing Foundation (CNCF), et a atteint le stade de projet « graduated » de la CNCF en mai 2026. Il fournit des API et des SDK pour chaque langage, une instrumentation automatique pour les frameworks et clients courants, le protocole de transport OTLP, le Collector, et des conventions sémantiques qui fixent les noms d'attributs comme `http.request.method` et `http.response.status_code`,

afin que la télémétrie de différentes équipes et de différents langages concorde. Les traces, les métriques et les journaux sont stables ; les profils ont été ajoutés comme quatrième signal et sont entrés en alpha publique en 2026.

L'intérêt architectural tient à la séparation entre l'instrumentation et le backend. Les services émettent de l'OTLP vers un Collector, qui tourne généralement à côté de chaque nœud ou comme passerelle centrale ; le Collector regroupe les données par lots, les échantillonne, supprime les attributs sensibles, ajoute des métadonnées comme la version déployée, et les exporte vers les stockages qu'utilise l'organisation, quels qu'ils soient. Changer de fournisseur, ou envoyer les mêmes données à deux backends pendant une migration, devient une modification de configuration du Collector au lieu d'une modification de code dans deux cents services. Une équipe plateforme fournit généralement la configuration du SDK, la propagation et le Collector sous forme de chemin balisé (paved road), si bien que les équipes produit n'ont qu'à ajouter des attributs métier.

Le pipeline de télémétrie



L'instrumentation réside dans les services, le traitement dans le Collector et le stockage dans des backends remplaçables.

```
from opentelemetry import trace
import structlog

tracer = trace.get_tracer("order-service")
log = structlog.get_logger()

def place_order(order):
    with tracer.start_as_current_span("place_order") as span:
        span.set_attribute("restaurant.id", order.restaurant_id)
        span.set_attribute("order.item_count", len(order.items))
        quote = pricing.quote(order)       # instrumented HTTP client adds
        payment.authorise(order, quote)    # the traceparent header itself
        log.info("order_placed", order_id=order.id,
                 total_pence=quote.total_pence)
        return order.id
```

Instrumentation manuelle avec l'API Python d'OpenTelemetry et un logger structuré. Les clients HTTP instrumentés automatiquement créent des spans enfants et propagent le contexte sans code supplémentaire.

Cardinalité et coût

La télémétrie peut facilement coûter plus cher que le système qu'elle observe. Pour les métriques, le principal facteur de coût est la cardinalité : chaque combinaison distincte de valeurs d'étiquettes est une série temporelle distincte qu'il faut stocker et indexer. Les étiquettes qui ont peu de valeurs (méthode HTTP, classe de statut, région) ne posent pas de problème. Les étiquettes aux valeurs non bornées (ID d'utilisateur, ID de commande, URL complète contenant des identifiants, texte d'un message d'erreur) multiplient les séries sans limite et peuvent faire tomber un système de métriques.

EXEMPLE COMMENTÉ

Exemple commenté : compter les séries temporelles

Le service de commandes exporte un histogramme de durée des requêtes avec 12 compartiments. Prometheus stocke chaque compartiment comme une série, plus une somme et un compte, soit 14 séries par combinaison d'étiquettes.

Paires route et méthode	60
Classes de statut (1xx à 5xx)	5
Instances	30
Séries pour cette seule métrique	60 × 5 × 30 × 14 = 126,000
Ajout d'une étiquette de région (5 régions)	126 000 × 5 = 630 000 séries
Ajout de restaurant_id (40 000 restaurants)	126 000 × 40 000 = environ 5 milliards de séries potentielles : ingérable

Les questions par restaurant ou par client ont leur place dans les traces, les journaux ou les événements larges, où une forte cardinalité est attendue, et non dans les étiquettes des métriques.

Les exemplars, qui associent un ID de trace à titre d'échantillon à un compartiment d'histogramme, relient un pic de latence sur une métrique peu coûteuse à une trace détaillée et coûteuse.

Les autres leviers de coût sont l'échantillonnage des traces (l'échantillonnage en queue conserve les traces utiles), une conservation plus courte pour les données détaillées et plus longue pour les agrégats, la suppression par défaut des journaux de débogage en production, et la refacturation de la télémétrie aux équipes qui la produisent, pour que son coût soit visible. Traitez le volume de télémétrie comme un paramètre de conception doté d'un budget, comme n'importe quelle autre ressource.

Alerter sur les symptômes, pas sur les causes

Selon le livre SRE, les alertes d'astreinte doivent porter sur les symptômes que ressentent les utilisateurs, et non sur les nombreuses causes possibles. Un CPU élevé, un disque plein sur une réplique ou un conteneur redémarré sont des causes : souvent sans conséquence, parfois déjà

traitées par l'automatisation, et jamais une liste complète de ce qui peut mal tourner. Un SLO dont le budget fond, des paiements en échec ou un pipeline de positions des livreurs bloqué sont des symptômes : ils comptent quelle qu'en soit la cause. Les alertes fondées sur les symptômes sont moins nombreuses, rarement fausses, et couvrent des défaillances que personne n'avait prévues ; les signaux fondés sur les causes ont leur place sur des tableaux de bord qui aident à diagnostiquer un symptôme une fois l'alerte déclenchée.

Chaque alerte d'astreinte doit être urgente, actionnable et réelle. La lassitude face aux alertes (alert fatigue) est en soi un risque pour la fiabilité : les ingénieurs d'astreinte réveillés par des alertes sans intérêt apprennent à les ignorer et manquent celle qui compte. Une règle pratique consiste à alerter l'astreinte sur les taux de consommation des SLO (section 9.5), à envoyer les consommations plus lentes et les avertissements sur les ressources vers une file de tickets, et à relier chaque alerte d'astreinte à un runbook (procédure d'exploitation) et à un tableau de bord.

GUIDE DE DÉCISION

Guide de décision : cette alerte doit-elle mobiliser une personne d'astreinte ?

Appliquez ces questions à chaque règle d'alerte avant de l'autoriser à réveiller quelqu'un, puis à nouveau chaque fois qu'elle se déclenche sans que personne ait besoin d'agir.

Q1 **La condition signifie-t-elle que des utilisateurs sont lésés maintenant, ou le seront bientôt (un SLO dont le budget fond vite, un parcours en échec) ?**

- **Oui** Passez à la question suivante.
- **Non** N'alertez pas l'astreinte. Placez la condition sur un tableau de bord pour le diagnostic, ou ouvrez un ticket si quelqu'un doit s'en occuper pendant les heures ouvrées.

Q2 **Faut-il qu'un humain agisse, plutôt qu'une réponse automatique comme un redémarrage ou un ajout d'instances (scale-out) ?**

- **Oui** Passez à la question suivante.
- **Non** Automatisez la réponse et n'alertez que si l'automatisation échoue.

Q3 **L'action doit-elle avoir lieu avant le prochain jour ouvré ?**

- **Oui** Alertez l'astreinte, et reliez l'alerte à un runbook et au tableau de bord pertinent.
- **Non** Envoyez-la dans la file de tickets de l'équipe plutôt que de déclencher l'astreinte.

Q4 **Cette alerte a-t-elle déclenché l'astreinte au cours du dernier mois sans que personne ait eu besoin de faire quoi que ce soit ?**

- **Oui** Corrigez le seuil ou le signal, ou supprimez l'alerte : une alerte d'astreinte sur laquelle personne n'agit apprend aux gens à ignorer les alertes.
- **Non** Conservez-la, et réexaminez-la lors de la prochaine passation d'astreinte.

Des tableaux de bord qui répondent à des questions

Un mur de cinquante graphiques ne répond vite à aucune question. Les bons tableaux de bord se conçoivent comme n'importe quelle interface utilisateur : pour un public identifié et une question précise. Une vue d'ensemble du service répond à «ce service est-il en bonne santé ?» avec ses SLO, le budget d'erreur restant et les métriques RED en haut, et les métriques USE de ses ressources clés en dessous. Un tableau de bord de parcours pour l'ouverture des ventes répond à «les fans obtiennent-ils des billets ?» avec les admissions, les blocages de places, les paiements et les échecs par minute, de bout en bout. Les tableaux de bord de diagnostic s'atteignent en descendant dans le détail ; ils ne s'affichent pas en premier.

Gérez les tableaux de bord en tant que code sous contrôle de version, donnez à chacun un propriétaire, affichez les déploiements et les changements de feature flags sous forme d'annotations sur les graphiques (la plupart des incidents suivent un changement), et supprimez les tableaux de bord que personne n'a ouverts depuis des mois. Pendant un incident, la première vue utile est généralement le parcours vu par l'utilisateur, la deuxième la trace d'une requête en échec, et seulement ensuite les métriques des ressources.

ATTENTION

Erreurs courantes en matière d'observabilité

- N'ajouter le traçage qu'à quelques services : un seul saut sans propagation casse toutes les traces qui le traversent.
- Mettre des ID d'utilisateur, des ID de commande ou des URL brutes dans les étiquettes des métriques.
- Faire la moyenne de percentiles entre instances ou entre fenêtres de temps.
- Mesurer uniquement sur le serveur, si bien que les défaillances du DNS, du CDN ou de l'équilibreur de charge restent invisibles.
- Journaliser des données personnelles et des secrets, ce qui fait de la base de journaux la copie la moins protégée des données des clients.
- Alerter sur chaque cause, si bien que les vrais symptômes se perdent dans le bruit.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi le p99 de tout un parc doit-il être calculé à partir d'histogrammes fusionnés plutôt qu'à partir du p99 de chaque instance ?

Parce qu'un percentile de données combinées n'est pas une fonction des percentiles des parties ; les nombres par compartiment s'additionnent exactement, si bien que l'histogramme fusionné donne une estimation juste (limitée par la largeur des compartiments), alors que des p99 moyennés peuvent être très faux.

▼ Qu'est-ce qui casse si un service d'une chaîne d'appels ne transmet pas l'en-tête traceparent ?

Chaque trace qui passe par lui est coupée en deux : les spans situés après ce saut démarrent une nouvelle trace, si bien que la vue de bout en bout et le lien entre cause et symptôme sont perdus.

▼ **Pourquoi order_id est-il une mauvaise étiquette de métrique, mais un bon attribut de span ?**

Comme étiquette, il crée une nouvelle série temporelle par commande, ce qui fait exploser la cardinalité et le coût ; comme attribut de span ou de journal, il est stocké une fois par événement, là où une forte cardinalité est attendue et utile pour les requêtes.

▼ **Lequel de ces signaux devrait alerter quelqu'un : un CPU à 95 % sur une instance, ou le SLO du paiement qui se consomme à un taux de 20 ?**

La consommation du SLO, car c'est un symptôme que ressentent les utilisateurs ; la mesure du CPU est une cause possible, qui a sa place sur un tableau de bord de diagnostic.

▼ **À vous : Un histogramme de latence a 10 compartiments, plus sum et count, sur 40 routes, 5 classes de statut et 20 instances. Combien de séries temporelles ? Un développeur ajoute une étiquette customer_id pour 200 000 clients. Calculez l'effet et donnez votre conseil.**

12 x 40 x 5 x 20 = 48 000 séries. Ajouter customer_id pourrait multiplier ce chiffre par 200 000, jusqu'à 9,6 milliards, ce qui serait inabordable et pourrait casser le système de métriques. Les métriques conviennent aux étiquettes bornées. Placez plutôt l'ID client dans les traces et les journaux structurés, où la requête d'une personne peut être suivie. Reliez-les par l'ID de trace, de sorte qu'un graphique montrant un pic de latence mène à une trace d'un client touché.

SOURCES

- Google SRE book: Monitoring Distributed Systems: https://sre.google/sre-book/monitoring-distributed-systems/
- Sigelman et al.: Dapper, a Large-Scale Distributed Systems Tracing Infrastructure (Google, 2010): https://research.google/pubs/dapper-a-large-scale-distributed-systems-tracing-infrastructure/
- W3C Trace Context: https://www.w3.org/TR/trace-context/
- OpenTelemetry documentation: Signals: https://opentelemetry.io/docs/concepts/signals/
- Prometheus documentation: Histograms and summaries: https://prometheus.io/docs/practices/histograms/
- Charity Majors, Liz Fong-Jones and George Miranda: Observability Engineering (O'Reilly, 2022): https://www.oreilly.com/library/view/observability-engineering/9781492076438/

Planification de la capacité et tests de performances

AVANT DE LIRE

Cette section s'appuie sur :

- 2.4 Performances et scalabilité
- 9.1 Passage à l'échelle
- 9.3 Files d'attente et back pressure

Termes déjà vus qu'elle utilise : Marge de capacité (headroom), Loi de Little, Percentile (p95, p99), Autoscaling (mise à l'échelle automatique), Fonction d'aptitude (fitness function)

L'IDÉE EN BREF

Avant un grand mariage, un traiteur estime le nombre d'invités qui viendront, calcule combien d'assiettes chaque cuisinier peut servir en une heure, prévoit quelques employés de réserve au cas où quelqu'un tomberait malade et commande les fours supplémentaires des semaines à l'avance, parce qu'on ne peut pas les acheter le jour même. Puis il organise une répétition grandeur nature pour vérifier que la cuisine tient vraiment le coup. Les équipes logicielles font de même : elles prévoient la demande, mesurent ce que chaque serveur peut supporter, ajoutent de la capacité de réserve, commandent ce qui prend du temps à obtenir et répètent avec des clients simulés réalistes.

CE QU'IL FAUT SAVOIR

- La planification de la capacité prévoit la demande de pointe, mesure la capacité par unité dans des conditions réalistes, ajoute de la marge et de la redondance, et commande tout ce qui a un long délai d'approvisionnement, y compris les quotas et les limites des tiers.
- Les tests de charge, de stress, d'endurance, de pic et de point de rupture répondent à des questions différentes : tenir les objectifs au pic, comportement en cas de défaillance au-delà, dégradation dans le temps, réaction aux hausses soudaines et limite réelle.
- Les modèles de charge fermés ralentissent quand le système ralentit, tandis que dans les modèles ouverts les arrivées continuent à un rythme fixé ; le trafic public et les ouvertures de billetterie sont ouverts, donc les tests en modèle fermé peuvent être dangereusement optimistes (Schroeder, Wierman et Harchol-Balter, 2006).
- L'omission coordonnée (Gil Tene) masque les réponses lentes lorsqu'un testeur attend les réponses avant d'envoyer ; mesurez à partir de l'heure d'envoi prévue, avec des générateurs en modèle ouvert et des histogrammes complets.

- k6, Gatling, JMeter et Locust sont les principaux outils open source ; des scénarios réalistes, des volumes de données réalistes, les clés très sollicitées, la latence des dépendances et un générateur non saturé comptent davantage que l'outil.
- Les budgets de performance, comme les seuils des Core Web Vitals ou une répartition de la latence par saut, sont vérifiés automatiquement dans le pipeline de livraison en tant que fonctions d'aptitude.
- La ressource partagée la plus rare, souvent un tiers, fixe le taux d'admission sûr pour un événement comme l'ouverture d'une billetterie.

La capacité est un plan, pas un espoir

La planification de la capacité répond à l'avance à une seule question : aurons-nous assez de tout quand la demande atteindra son pic ? «Tout» va bien au-delà des serveurs. Cela comprend les connexions à la base de données et le débit d'écriture, la mémoire du cache, les partitions du courtier de messages (message broker), les quotas du compte cloud, les limites de débit des tiers, les licences et les personnes d'astreinte. L'élasticité du cloud permet d'ajouter rapidement une partie de la capacité, mais pas toute : on ne double pas en quelques secondes une base de données primaire, un prestataire de paiement peut exiger plusieurs semaines de préavis pour relever une limite, et l'autoscaling (mise à l'échelle automatique, traité à la section 9.1) réagit en quelques minutes à un changement brutal qui survient en quelques secondes.

La méthode comporte quatre étapes, répétées selon un cycle régulier (chaque trimestre pour une croissance régulière, à chaque événement pour les lancements et les ouvertures de billetterie) :

Le cycle de planification de la capacité

1. **Prévoir la demande**
   Combinez la croissance organique (commandes par jour en hausse de 4 % par mois), la saisonnalité (pics du déjeuner et du dîner, vendredis soir, Noël) et les événements connus (une campagne marketing, l'ouverture de la billetterie d'un stade). Exprimez la demande dans les unités selon lesquelles le système passe à l'échelle : requêtes par seconde par parcours, écritures par seconde, messages par seconde, au pic, et non en moyenne journalière.

2. **Mesurer la capacité par unité**
   Déterminez, par des tests, quelle charge une instance, une base de données ou une partition peut supporter tout en respectant son SLO de latence. Le coude de la courbe du temps de réponse, là où la mise en file d'attente commence à dominer (section 2.4), est la limite pratique, et non le point où tout s'effondre.

3 **Ajouter de la marge et de la redondance**
Dimensionnez pour le pic prévu, plus une marge pour les erreurs de prévision, les rafales et les nouvelles tentatives, et de sorte que la perte d'une instance ou d'une zone de disponibilité entière laisse encore assez de capacité (redondance N+1 ou au niveau des zones).

4 **Planifier les délais d'approvisionnement et faire le bilan**
Dressez la liste de ce qui demande du temps à obtenir (augmentations de quotas, capacité réservée, mises à niveau de bases de données, limites des tiers, matériel pour les systèmes sur site) et commandez-le suffisamment tôt. Ensuite, comparez la demande réelle à la prévision et améliorez le modèle.

Le calcul de dimensionnement avec la loi de Little a été détaillé à la section 9.1 ; cette section se concentre sur la manière d'obtenir des chiffres fiables à y injecter. L'erreur la plus courante en planification de la capacité n'est pas un mauvais calcul, mais une capacité par instance mesurée dans des conditions irréalistes ; la deuxième est d'oublier une contrainte partagée, comme une limite imposée par un tiers, qu'aucune marge de capacité (headroom) sur vos propres serveurs ne peut compenser.

Les types de tests de performances

Les tests de performances sont le seul moyen de mesurer la capacité, de trouver les goulots d'étranglement avant les clients et de vérifier qu'une version n'a pas aggravé la situation. Les tests diffèrent par la forme de charge qu'ils appliquent et par la question à laquelle ils répondent. Un test de charge vérifie le pic attendu ; un test de stress va au-delà pour voir comment le système défaille ; un test d'endurance (soak test) dure des heures pour révéler les fuites lentes ; un test de pic (spike test) vérifie les réactions à des sauts soudains ; un test de point de rupture augmente la charge jusqu'à ce que quelque chose cède, pour trouver la vraie limite.

Types de tests de performances

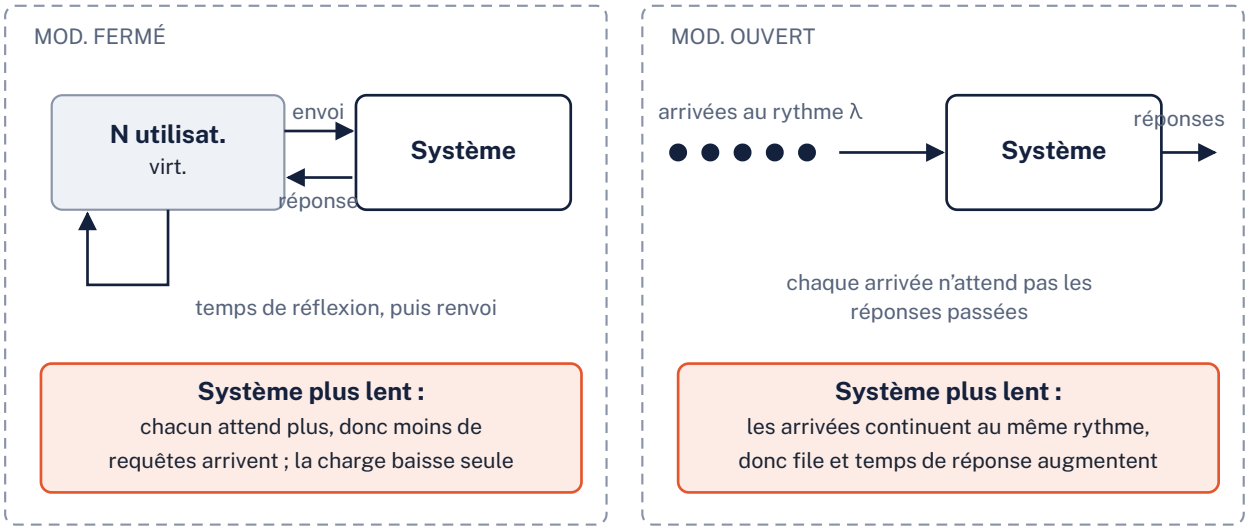
Test	Question à laquelle il répond	Forme de la charge	Durée typique	Points à surveiller
Fumée (smoke)	Le script de test et l'environnement fonctionnent-ils seulement ?	Quelques utilisateurs	Quelques minutes	Erreurs de script, mauvais environnement
Charge	Atteignons-nous nos objectifs de latence et d'erreurs au pic attendu ?	Montée jusqu'au pic prévu, puis maintien	30 à 60 minutes	Latence p99, taux d'erreur, saturation des ressources
Stress	Comment le système défaille-t-il au-delà du pic, et se rétablit-il ?	Montée jusqu'à 1,5 à 3 fois le pic, puis redescente	30 à 60 minutes	Dégradation gracieuse, délestage de charge, temps de rétablissement
Endurance (soak)	Quelque chose se dégrade-t-il avec le temps ?	Charge normale ou de pic, maintenue	Des heures à des jours	Fuites de mémoire, files qui s'allongent, fuites de connexions, disque qui se remplit
Spike (investigation limitée dans le temps)	Pouvons-nous absorber un afflux soudain ?	Saut d'une charge faible à très élevée en quelques secondes	Quelques minutes	Retard de l'autoscaling, débordement des files, comportement du limiteur de débit
Point de rupture (capacité)	Où est la limite, et quel est le premier goulot d'étranglement ?	Montée régulière jusqu'à la défaillance	Jusqu'à la rupture	Le coude de la latence, la première ressource saturée

Modèles de charge ouverts et fermés

Les générateurs de charge simulent les utilisateurs de l'une de deux manières, et choisir la mauvaise produit des résultats non seulement inexacts, mais dangereusement optimistes. Dans un modèle de charge fermé, l'outil fait tourner un nombre fixe d'utilisateurs virtuels ; chacun envoie une requête, attend la réponse, marque une pause (le temps de réflexion) et recommence. Si le système ralentit, chaque utilisateur virtuel envoie moins de requêtes, et la charge baisse d'elle-même. Dans un modèle de charge ouvert, les requêtes arrivent à un rythme fixé, indépendamment de la vitesse à laquelle les précédentes reçoivent une réponse, comme le font les vrais visiteurs d'un site web public : des milliers de fans qui appuient sur Acheter à 10:00 ne s'attendent pas les uns les autres.

Bianca Schroeder, Adam Wierman et Mor Harchol-Balter ont montré («Open Versus Closed: A Cautionary Tale», NSDI 2006) que les deux modèles peuvent donner des temps de réponse très différents pour le même système et la même charge moyenne, et que des décisions d'ordonnancement ou de capacité réglées avec l'un peuvent être fausses avec l'autre. Les systèmes internes dont la population de clients est fixe, comme une application de centre d'appels utilisée par 300 agents, sont raisonnablement fermés. Le trafic internet public, les consommateurs pilotés par les événements alimentés par une file et les ouvertures de billetterie sont ouverts, ou presque. Pour de tels systèmes, c'est un test en modèle ouvert qui est réaliste ; les exécuteurs `constant-arrival-rate` et `ramping-arrival-rate` de k6, par exemple, le mettent en œuvre, tandis que son exécuteur `constant-vus` est fermé.

Fermé ou ouvert



Un test en modèle fermé lève poliment le pied quand le système peine ; le vrai trafic public, non.

SI VOUS DÉBUTEZ

Pas à pas : un test qui lève poliment le pied

L'idée qui demande le plus d'attention ici est que la manière dont un générateur de charge envoie les requêtes peut masquer une surcharge. Dans un modèle fermé, les utilisateurs simulés attendent chaque réponse, si bien qu'un système en difficulté reçoit automatiquement moins de charge ; les vrais fans lors d'une ouverture de billetterie ne s'attendent pas les uns les autres.

- Une équipe teste le plan de salle de l'ouverture de billetterie avec 300 utilisateurs virtuels. Chacun récupère le plan, attend la réponse, fait une pause de 2,5 secondes et recommence. Avec des réponses en 0,5 seconde, chaque cycle dure 3 secondes : 300 ÷ 3 = 100 requêtes par seconde.
- À mi-parcours, une requête lente sur la base de données fait passer chaque réponse à 5 secondes. Chaque cycle dure désormais 7,5 secondes, donc les mêmes 300 utilisateurs n'envoient plus que 300 ÷ 7,5 = 40 requêtes par seconde.
- La charge a plus que diminué de moitié précisément quand le système est en difficulté, un répit que le vrai trafic n'accorde jamais : le rapport paraît donc bien meilleur que ne le sera l'ouverture de billetterie.
- Les vrais fans suivent un modèle ouvert : 100 par seconde continuent d'arriver, quel que soit le sort des requêtes précédentes. À 5 secondes chacune, la loi de Little donne 100 × 5 = 500 requêtes en cours, dix fois les 50 habituelles, et si le système ne peut pas en terminer 100 par seconde, la file d'attente ne cesse de grandir.
- Les exécuteurs à taux d'arrivée de k6 reproduisent ce comportement : ils lancent de nouvelles requêtes à un rythme fixé, quelle que soit la lenteur des réponses précédentes, à condition que le test autorise suffisamment d'utilisateurs virtuels.
- Les tests fermés conviennent à une population réellement fixe, comme une application de centre d'appels utilisée par 300 agents qui n'envoient rien pendant qu'ils attendent.
- **La leçon :** adaptez le modèle à ceux qui envoient le trafic. Pour les sites publics, les consommateurs de files et les ouvertures de billetterie, testez à un taux d'arrivée fixé, sinon le test est le plus optimiste précisément là où cela compte le plus.

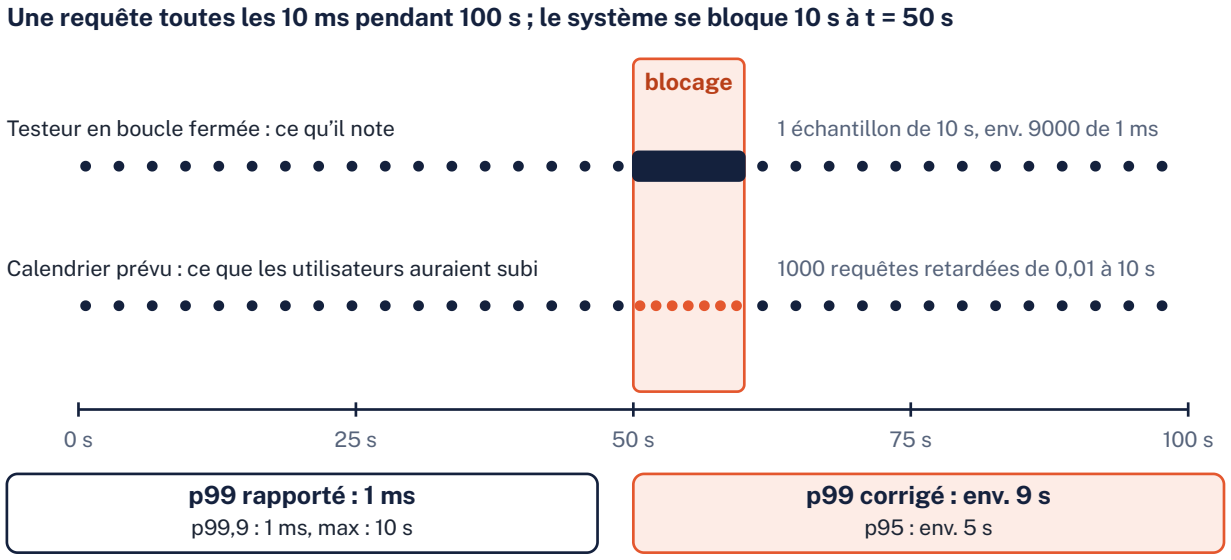
Omission coordonnée (coordinated omission)

Gil Tene, d'Azul Systems, auteur de la bibliothèque HdrHistogram, a nommé une erreur de mesure qui touche de nombreux tests de latence et outils de supervision : l'omission coordonnée (coordinated omission). Un générateur de charge en boucle fermée attend chaque réponse avant d'envoyer la requête suivante. Quand le système se bloque, le générateur se

bloque avec lui, si bien que les requêtes que de vrais utilisateurs auraient envoyées pendant le blocage, et qui auraient toutes été lentes, ne sont jamais envoyées ni mesurées. Le testeur se «coordonne» avec le système qu'il teste et omet précisément les mauvais échantillons.

Prenons un test qui prévoit d'envoyer une requête toutes les 10 ms pendant 100 secondes à un service qui répond normalement en 1 ms. À 50 secondes, le service se fige pendant 10 secondes (une longue pause du ramasse-miettes, par exemple). Le testeur en boucle fermée enregistre environ 9000 échantillons rapides et un échantillon de 10 secondes : il indique donc un p99 de 1 ms, et même un p99,9 de 1 ms. Selon le calendrier prévu, pourtant, environ 1000 requêtes seraient arrivées pendant le blocage, et auraient attendu entre 10 secondes et 10 ms, soit environ 5 secondes en moyenne. Dix pour cent de toutes les requêtes ont été fortement retardées : le vrai p95 est d'environ 5 secondes, et le vrai p99 d'environ 9 secondes.

Ce que le testeur ne voit pas



L'unique échantillon lent du testeur masque mille expériences utilisateur lentes.

Les remèdes consistent à utiliser des générateurs en modèle ouvert (à taux d'arrivée) pour les systèmes ouverts, à mesurer la latence à partir du moment où une requête *devait* être envoyée plutôt que du moment où elle l'a réellement été, à enregistrer des histogrammes complets plutôt que des moyennes, et à se méfier de tout rapport de latence dont le maximum est très au-dessus du p99,9 avec peu d'échantillons entre les deux. L'outil de benchmark HTTP modifié par Tene, wrk2, et les options de correction de HdrHistogram existent précisément pour cela. La même erreur apparaît dans la supervision en production lorsqu'une bibliothèque cliente ne mesure que les requêtes qu'elle a réussi à envoyer.

Les outils

Quatre outils open source couvrent la plupart des besoins. **Apache JMeter** est le vétéran : basé sur Java, avec un éditeur graphique de plans de test, de nombreux plug-ins de protocoles et des groupes de threads fermés par défaut. **Gatling** écrit les tests sous forme de code (Scala, Java et Kotlin, avec JavaScript et TypeScript ajoutés plus récemment), est efficace à forte charge et propose des profils d'injection ouverts et fermés. **k6**, maintenu par Grafana Labs, exécute des scripts de test en JavaScript sur un moteur écrit en Go, possède des exécuteurs explicites en modèle ouvert et fermé, et des seuils de réussite ou d'échec adaptés à l'intégration continue. **Locust** décrit le comportement des utilisateurs en Python et répartit la charge entre des processus de travail ; ses utilisateurs fonctionnent en boucle fermée. Des services commerciaux et hébergés dans le cloud ajoutent la génération de charge distribuée à grande échelle et les rapports.

L'outil compte moins que le scénario. Un bon script reproduit le mélange réel des parcours (d'après les statistiques de production : par exemple 70 % de navigation, 20 % de modifications du panier, 10 % de paiement), des temps de réflexion réalistes, des données réalistes (des utilisateurs, des restaurants et des places différents, et non le même 10 000 fois) et le bon modèle de charge ; il vérifie aussi que les réponses sont correctes, car une page d'erreur rapide n'est pas une requête réussie.

```
import http from 'k6/http';
import { check } from 'k6';

export const options = {
  scenarios: {
    onsale: {
      executor: 'ramping-arrival-rate',   // open model: arrivals per second
      startRate: 100, timeUnit: '1s',
      preAllocatedVUs: 2000, maxVUs: 20000,
      stages: [
        { target: 4900, duration: '2m' },  // ramp to 1.5x design load
        { target: 4900, duration: '20m' }, // hold
      ],
    },
  },
  thresholds: {                           // fail the run if missed
    http_req_failed: ['rate<0.005'],
    http_req_duration: ['p(99)<500'],
  },
};

export default function () {
  const res = http.get('https://perf.tickets.example/api/events/42/seats');
  check(res, { 'status is 200': (r) => r.status === 200 });
}
```

Un test k6 en modèle ouvert pour le plan de salle de l'ouverture de billetterie, qui monte jusqu'à 4900 requêtes par seconde et fait échouer l'exécution si le p99 dépasse 500 ms ou si plus de 0,5 % des requêtes échouent.

Des environnements et des données réalistes

Un test de performances ne vaut que ce que vaut son environnement. Une base de données de test de 10 000 lignes dont les index tiennent en mémoire ne dit rien de la production avec 200 millions de lignes dont les index n'y tiennent pas. La liste de contrôle ci-dessous repère les différences habituelles.

Taille et forme des données : des volumes à l'échelle de la production et des distributions réalistes, y compris les clés très sollicitées, comme les restaurants les plus populaires ou les meilleures places, qui créent une contention qu'un test uniforme ne voit jamais. **Mêmes briques :** les mêmes types d'instances, limites de conteneurs, configuration de base de données, tailles de pools de connexions et topologie réseau (y compris les sauts entre zones). **État du cache :** testez à la fois avec un cache chaud et un cache froid, car un déploiement ou un failover les vide (section 7.6). **Dépendances :** des services tiers remplacés par des bouchons qui reproduisent leur vraie distribution de latence et leurs limites de débit, pas des réponses instantanées. **Isolement :** aucun test d'une autre équipe partageant l'environnement, sinon les

résultats ne sont que du bruit. **Le générateur lui-même :** une capacité de génération de charge suffisante, aux bons emplacements ; un générateur qui tourne à 100 % de CPU se mesure lui-même.

Certaines organisations testent en production, avec prudence, parce qu'aucune copie ne lui ressemble tout à fait : utilisateurs synthétiques et événements de test exclus des rapports métier, duplication du trafic (traffic shadowing : copie de vraies requêtes vers une nouvelle version dont les réponses sont jetées) et montées progressives avec conditions d'arrêt automatique. Cela exige les mêmes garde-fous que les expériences de chaos : un faible rayon d'impact (blast radius), un interrupteur d'arrêt d'urgence et une surveillance étroite des SLO.

Les budgets de performance

Un budget de performance est une limite qu'une modification ne doit pas dépasser, convenue à l'avance et vérifiée automatiquement, afin que les performances ne s'érodent pas une petite régression après l'autre. Pour les frontends web, les budgets portent sur le poids des pages, la taille des scripts et les Core Web Vitals, pour lesquels les seuils de Google pour une bonne expérience, mesurés au 75e percentile des chargements de pages, sont un Largest Contentful Paint de 2,5 secondes au maximum, un Interaction to Next Paint de 200 ms au maximum et un Cumulative Layout Shift de 0,1 au maximum.

Pour les backends, un budget de latence répartit un SLO de bout en bout entre les sauts d'un parcours. L'étape de paiement de la plateforme de livraison de repas a un objectif de p99 de 800 ms ; l'équipe attribue 20 ms à la passerelle, 60 ms au service de commandes, 80 ms à la tarification, 450 ms au paiement (dont l'essentiel pour le prestataire de cartes externe), 40 ms à la base de données, 100 ms au réseau et à la sérialisation, et garde 50 ms en réserve. Chaque équipe connaît désormais son propre objectif, et une modification qui pousse la tarification à 150 ms apparaît comme un dépassement de budget avant de devenir un non-respect du SLO. En toute rigueur, les percentiles des parties ne s'additionnent pas pour donner le percentile de l'ensemble : traitez donc cette répartition comme une allocation d'ingénierie, et vérifiez séparément le chiffre de bout en bout.

On fait respecter les budgets en exécutant un test de performances court et stable dans le pipeline de livraison et en faisant échouer le build lorsqu'un seuil est franchi, comme le font les seuils k6 ci-dessus. C'est une fonction d'aptitude pour les performances. Les tests du pipeline doivent être reproductibles plus que réalistes : même environnement, mêmes données, même charge, pour qu'une variation de 10 % signale une vraie régression et non du bruit.

EXEMPLE COMMENTÉ

Exemple commenté : un plan de capacité pour l'ouverture d'une billetterie

Il s'agit d'un événement différent, et plus important, que l'ouverture de billetterie de la section 9.3, avec ses propres chiffres. Une tournée dans des stades compte trois soirs de 60 000 places, soit 180 000 billets au total, mis en vente un vendredi à 10:00. Les fans entrent dans une salle d'attente virtuelle et sont admis dans le parcours d'achat à un rythme contrôlé. Les tests de charge ont mesuré le coude de chaque instance d'API à 520 requêtes par seconde ; l'équipe planifie sur la base de 400 pour garder le p99 sous 500 ms.

Prévision de la demande	520 000 inscriptions en prévente, dont environ 75 % devraient se présenter : prévoir 400 000 fans dans la salle d'attente. À 2,5 billets par commande en moyenne, 72 000 commandes suffisent à vendre toute la tournée.
Contrainte déterminante	La limite convenue avec le prestataire de paiement est de 120 paiements par seconde. Avec 80 % des fans admis qui achètent et un tiers de la limite réservé aux nouvelles tentatives et aux erreurs, admettre 100 fans par seconde (6000 par minute) : 80 paiements par seconde.
Fans actifs (loi de Little)	100 fans/s × 240 s dans le parcours d'achat = 24 000 fans actifs simultanément
Charge de l'API	100 fans/s × 25 requêtes par parcours = 2500 requêtes/s ; × 1,3 pour la concentration des appels au plan de salle = 3250 requêtes/s de charge de conception
Instances	3250 / 400 = 8,1, donc 9 instances ; pour survivre à la perte d'une zone sur trois, les 9 doivent tenir dans deux zones : 5 par zone × 3 = 15 instances, démarrées avant 09:30
Base de données des places bloquées	1,5 tentative de blocage par fan = 150 écritures/s, pour une capacité mesurée de 600 écritures/s avec un p99 sous 50 ms : 25 % d'utilisation, mais tester la contention sur les places les plus demandées
Salle d'attente	400 000 fans qui interrogent le serveur toutes les 20 s = 20 000 requêtes/s, servies en périphérie du CDN, sans jamais atteindre la couche API
Résultat	72 000 commandes / 80 par seconde = 900 s : tout est vendu en 15 minutes environ, après l'admission d'environ 90 000 fans ; la salle d'attente doit prévenir rapidement tous les autres
Délais d'approvisionnement	Limite de paiement confirmée 3 semaines avant ; quota cloud relevé 1 semaine avant ; test de charge à 1,5x (environ 4900 requêtes/s), test de pic et test d'endurance de 2 heures 2 semaines avant, dans un environnement de taille production

Le taux d'admission est fixé par la ressource partagée la plus rare, ici le prestataire de paiement, et non par le nombre de serveurs que l'équipe peut s'offrir ; doubler les serveurs ne vendrait pas les billets plus vite.

L'autoscaling est ici un filet de sécurité, pas le plan : un saut de zéro à 3250 requêtes par seconde à 10:00:00 arrive plus vite que les instances ne peuvent démarrer.

ATTENTION

Les pièges qui font mentir les tests de performances

- Des générateurs en modèle fermé pour un trafic ouvert, et une latence mesurée à partir de l'heure d'envoi réelle plutôt que prévue (omission coordonnée).
- Rapporter des moyennes au lieu de percentiles et d'histogrammes.
- Des données de test minuscules et uniformes, qui tiennent en mémoire et n'ont aucune clé très sollicitée.
- Des bouchons qui répondent instantanément alors que la vraie dépendance met 400 ms et impose une limite de débit.
- Compter des réponses d'erreur rapides comme des requêtes réussies.
- Un générateur de charge qui est lui-même le goulot d'étranglement.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi un test de charge en modèle fermé tend-il à sous-estimer les temps de réponse d'un site web public ?

Parce que ses utilisateurs virtuels attendent chaque réponse avant d'en envoyer d'autres : quand le système ralentit, la charge offerte baisse, alors que les vrais visiteurs continuent d'arriver et que les files s'allongent.

▼ Un testeur affiche un p99 de 2 ms et un maximum de 8 s, presque sans échantillons entre les deux. Que devez-vous soupçonner ?

Une omission coordonnée : le testeur s'est probablement bloqué avec le système et n'a pas enregistré les nombreuses requêtes qui auraient été retardées pendant la pause.

▼ Dans l'exemple de l'ouverture de billetterie, pourquoi l'ajout d'instances d'API ne permet-il pas de tout vendre plus vite ?

Parce que c'est la limite de 120 paiements par seconde du prestataire de paiement, et non la couche API, qui plafonne le nombre de fans pouvant acheter chaque seconde.

▼ **Qu’est-ce qu’un test d’endurance est conçu pour trouver ?**

Les problèmes qui n’apparaissent qu’avec le temps sous une charge constante, comme les fuites de mémoire ou de connexions, les files qui s’allongent et les disques qui se remplissent.

▼ **À vous : Une équipe teste l’ouverture de sa billetterie avec 1000 utilisateurs virtuels, chacun attendant une réponse avant d’envoyer à nouveau, et annonce un p99 de 300 ms. Les vrais fans arriveront à raison de 2000 par seconde. Pourquoi ne pas s’y fier ? Que lanceriez-vous à la place ?**

C’est une charge de travail fermée : quand le système ralentit, les testeurs ralentissent aussi, et la pression retombe, ce que de vrais fans ne feraient pas. Elle masque aussi le retard par omission coordonnée, puisque la latence est mesurée depuis l’envoi réel et non depuis l’envoi prévu. Exécutez un test en modèle ouvert à un rythme d’arrivée fixe, en mesurant depuis l’heure d’envoi prévue avec des histogrammes complets. Ajoutez un test de pic, avec des données réalistes, des places très demandées et les limites du prestataire de paiement.

SOURCES

- Schroeder, Wierman and Harchol-Balter: Open Versus Closed: A Cautionary Tale (NSDI 2006): https://www.usenix.org/conference/nsdi-06/open-versus-closed-cautionary-tale
- Grafana k6 documentation: Open and closed models: https://grafana.com/docs/k6/latest/using-k6/scenarios/concepts/open-vs-closed/
- web.dev: Web Vitals: https://web.dev/articles/vitals
- Google SRE book: Handling Overload: https://sre.google/sre-book/handling-overload/

SECTION 9.8 SUR 9.8 · INTERMÉDIAIRE · 25 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Ingénierie du chaos, incidents et apprentissage sans reproche

AVANT DE LIRE

Cette section s'appuie sur :

- 2.5 Disponibilité et fiabilité
- 9.4 Patrons de résilience
- 9.5 SLO et budgets d'erreur

Termes déjà vus qu'elle utilise : Temps moyen de réparation (MTTR), Budget d'erreur (error budget), Objectif de niveau de service (SLO), Runbook (procédure d'exploitation), Failover (basculement)

L'IDÉE EN BREF

Les pompiers n'attendent pas un vrai incendie pour savoir si les lances fonctionnent. Ils font des exercices, allument de petits feux maîtrisés pour s'entraîner et, après chaque vrai incendie, se réunissent pour comprendre ce qui s'est passé, sans chercher de coupable, afin que le prochain incendie se passe mieux. Les équipes logicielles font de même. Elles cassent délibérément de petites parties de leurs systèmes pour vérifier que les filets de sécurité fonctionnent, suivent un plan clair avec des rôles nommés quand quelque chose tourne vraiment mal, puis étudient honnêtement chaque incident pour que le système, et pas seulement les personnes, s'améliore.

CE QU'IL FAUT SAVOIR

- L'ingénierie du chaos, née du Chaos Monkey de Netflix (décrit en décembre 2010), mène des expériences sur un système pour gagner la confiance qu'il résiste à des conditions turbulentes ; c'est un test contrôlé d'une hypothèse, pas une casse aléatoire.
- Les Principles of Chaos Engineering sont : une hypothèse sur le comportement en état stable, des événements du monde réel, des expériences en production, une automatisation continue et un rayon d'impact (blast radius) minimal avec des conditions d'arrêt.
- Les game days répètent des défaillances importantes avec toute l'équipe et testent les personnes, les accès, les runbooks, les sauvegardes et la communication autant que le logiciel.
- La gestion des incidents sépare les rôles : un coordinateur d'incident qui coordonne au lieu de déboguer, des intervenants techniques, un responsable de la communication chargé de la page de statut et un scribe ; les niveaux de gravité sont définis par l'impact sur les

utilisateurs, et l'atténuation passe avant le diagnostic.

- Les revues post-incident sans reproche (livre SRE de Google ; John Allspaw, 2012) cherchent les seconds récits et plusieurs facteurs contributifs plutôt qu'une cause racine unique ou une erreur humaine.
- Le MTTR calculé sur de nombreux incidents est une métrique trompeuse, car les durées sont très asymétriques ; l'impact sur les SLO, les thèmes récurrents et les modifications du système menées à bien sont de meilleurs repères.
- Les actions fortes suppriment les dangers ou ajoutent des garde-fous, ont des responsables et des échéances, et sont priorisées au moyen de la politique de budget d'erreur ; les rappels à la prudence ne changent rien.

S'exercer à la défaillance, volontairement

Tout ce que contient ce chapitre jusqu'ici est une promesse sur la manière dont le système se comportera quand quelque chose tournera mal : les délais d'expiration se déclencheront, les disjoncteurs s'ouvriront, la charge sera délestée, les répliques prendront le relais, les alertes parviendront à quelqu'un qui sait quoi faire. Le chapitre 2 avertissait qu'un failover jamais exercé échoue souvent le jour où l'on en a enfin besoin (section 2.5). Cette section explique comment transformer ces promesses en preuves, et comment les organisations réagissent aux défaillances qui surviennent malgré tout et en tirent des leçons.

Deux pratiques vont de pair. Avant les défaillances, l'*ingénierie du chaos* et les *game days* provoquent délibérément de petites défaillances maîtrisées pour trouver des faiblesses et former les personnes. Après les défaillances, la *gestion des incidents* limite les dégâts et les *revues post-incident sans reproche* transforment chaque incident en amélioration durable. Les deux reposent sur la même conviction : les systèmes complexes sont toujours en partie défaillants, et la résilience vient de la capacité à découvrir comment, tôt et sans risque.

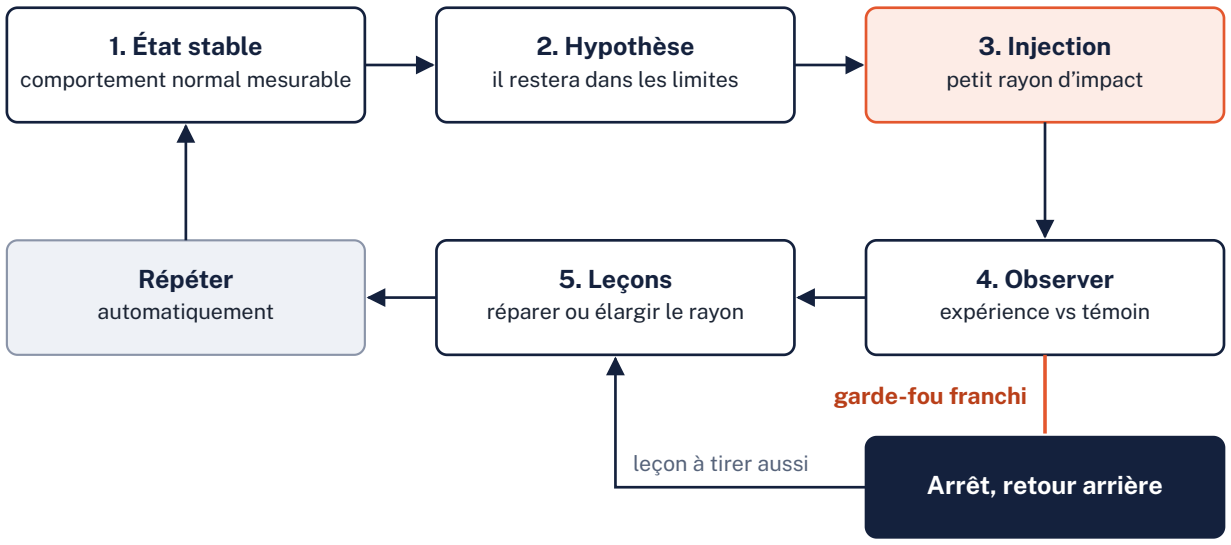
Ingénierie du chaos

L'ingénierie du chaos est née de la migration de Netflix de ses propres centres de données vers Amazon Web Services, entamée après une grave corruption de base de données en 2008. Des instances cloud individuelles pouvaient disparaître à tout moment ; Netflix a donc créé Chaos Monkey, un outil qui arrêtait aléatoirement des instances de production pendant les heures de bureau, quand les ingénieurs étaient à leur poste pour remarquer et corriger tout problème qui en résultait. Netflix l'a d'abord décrit publiquement dans un billet de blog de décembre 2010, puis a décrit une famille d'outils apparentés appelée Simian Army en juillet 2011. Des outils ultérieurs sont allés plus loin : Chaos Kong simulait la perte d'une région AWS entière en évacuant le trafic de celle-ci.

Des ingénieurs de Netflix et d'autres ont ensuite formalisé la discipline dans les *Principles of Chaos Engineering* (principlesofchaos.org, dernière révision en 2019), qui la définissent comme le fait d'expérimenter sur un système pour gagner confiance dans sa capacité à résister à des conditions turbulentes en production. Le mot qui compte est *expérimenter* : l'ingénierie du chaos ne consiste pas à casser des choses au hasard pour voir ce qui se passe, mais à appliquer la méthode scientifique à la résilience. Les principes énumèrent cinq pratiques :

Formulez une hypothèse autour du comportement en état stable. Choisissez une sortie mesurable qui indique un fonctionnement normal, comme le nombre de commandes passées par minute ou le SLI de l'étape de paiement, plutôt que des métriques internes comme le CPU. L'hypothèse d'état stable prédit que cette sortie restera dans des limites définies pendant l'injection de la défaillance. **Variez les événements du monde réel.** Injectez les défaillances qui se produisent réellement : des instances qui meurent, une dépendance qui ralentit, une perte de paquets, un disque plein, un certificat expiré, une dérive d'horloge, un pic de trafic. **Menez les expériences en production.** Seule la production a le vrai trafic, les vraies données et la vraie configuration. **Automatisez les expériences pour qu'elles s'exécutent en continu.** Une expérience ponctuelle prouve la résilience une fois ; les systèmes changent tous les jours. **Réduisez au minimum le rayon d'impact.** Commencez par la plus petite expérience capable d'apprendre quelque chose (une instance, un faible pourcentage du trafic) et gardez un bouton d'arrêt à portée de main.

La boucle d'expérience de chaos



Une expérience de chaos est un test contrôlé d'une hypothèse, avec des garde-fous qui l'arrêtent avant que les utilisateurs n'en pâtissent sérieusement.

Mener des expériences en production est le principe le plus débattu. Ses défenseurs font valoir que les environnements de staging diffèrent par le trafic, les données, l'échelle et la configuration, que seules les expériences en production révèlent les vraies faiblesses, et qu'une petite expérience maîtrisée à 15:00 est bien plus clémente pour les utilisateurs que la même défaillance survenant sans contrôle à 03:00. Les critiques soulignent que de nombreuses organisations n'ont pas l'observabilité, l'automatisation et la confiance interne nécessaires pour le faire sans risque. Le consensus pratique est une progression : commencer dans un environnement de staging, puis passer en production avec un rayon d'impact minime et des conditions d'arrêt automatique liées aux SLO, et n'élargir qu'à mesure que la confiance grandit. Ne commencez jamais par un système que vous savez déjà fragile ; corrigez d'abord les faiblesses connues, car une expérience dont vous pouvez prédire le résultat n'apprend rien.

Les outils sont désormais largement disponibles : les fournisseurs cloud proposent l'injection de fautes managée (AWS Fault Injection Service, Azure Chaos Studio), Chaos Mesh et LitmusChaos sont des projets open source de la CNCF pour Kubernetes, et des plateformes commerciales comme Gremlin fournissent des bibliothèques d'attaques avec des contrôles de sécurité. Les service meshes et les proxys peuvent injecter de la latence et des erreurs dans un pourcentage des appels sans toucher au code de l'application. Les outils sont la partie facile ; le plus dur est de choisir des hypothèses pertinentes et d'agir sur les résultats.

EXEMPLE COMMENTÉ

Exemple commenté : une expérience de latence sur le service de commandes

Le service de commandes de la plateforme de livraison de repas appelle le service de la carte pour confirmer le prix des articles. La conception (conforme à la section 9.4) prévoit pour cet appel un délai d'expiration (timeout) de 300 ms et un repli sur les prix en cache. Chaque instance du service de commandes dispose de 50 threads de travail et reçoit environ 40 commandes par seconde, chacune occupant normalement un thread pendant 0,2 s.

État stable	Taux de réussite des commandes d'au moins 99,9 % ; p99 de l'étape de paiement sous 800 ms ; commandes par minute à moins de 5 % de la même minute la semaine précédente
Hypothèse	Si les appels au service de la carte depuis une instance du service de commandes prennent 2000 ms de plus, les métriques d'état stable de cette instance restent dans leurs limites, car le délai d'expiration et le repli absorbent le retard
Rayon d'impact (blast radius)	1 instance sur 20 (5 % du trafic des commandes) dans une région, à 15:00 un jour de semaine, pendant 15 minutes, les 19 autres instances servant de groupe témoin
Conditions d'arrêt	Taux de réussite de l'instance testée inférieur à 99 % pendant 2 minutes, ou taux de consommation du SLO régional de l'étape de paiement supérieur à 14,4 : arrêt automatique
Utilisation prévue des threads	Avec le délai d'expiration de 300 ms : 40/s × (0,2 + 0,3) s = 20 threads occupés sur 50
Ce qui s'est passé	Le taux de réussite est tombé à 97 % en 3 minutes et l'arrêt s'est déclenché. Les traces montraient des appels qui attendaient 2,0 s : le délai d'expiration était en réalité de 3000 ms, une valeur par défaut d'une bibliothèque qui écrasait le réglage voulu
Pourquoi cela a échoué	40/s × (0,2 + 2,0) s = 88 threads nécessaires, mais il n'en existe que 50 : les requêtes se sont mises en file d'attente et ont expiré au niveau de la passerelle
Actions	Définir le délai d'expiration dans la configuration partagée du client, ajouter un test automatisé qui fait échouer le build si un appel sortant n'a pas de délai d'expiration explicite, relancer l'expérience, puis l'élargir à une zone entière

L'expérience a révélé, sur 5 % du trafic d'une région et en 3 minutes, une faiblesse qu'un vrai ralentissement du service de la carte à l'heure du déjeuner aurait transformée en panne régionale.

La conception était juste ; la configuration ne l'était pas. Les expériences testent ce qui est déployé, pas ce qui est documenté.

Les game days

Un game day (journée d'exercice) est un exercice planifié au cours duquel une équipe provoque ou simule délibérément une défaillance importante et s'entraîne à y répondre. Jesse Robbins a popularisé cette pratique chez Amazon, où son intitulé de poste aurait été Master of Disaster. Google organise de grands exercices de tests de reprise après sinistre (Disaster Recovery Testing, DiRT), et son livre SRE décrit la Wheel of Misfortune, un jeu de rôle dans lequel des ingénieurs d'astreinte revivent un incident passé tandis qu'un maître du jeu révèle ce que la supervision aurait montré.

Les game days testent plus que le logiciel. Ils vérifient si les alertes parviennent aux bonnes personnes, si celles-ci disposent des accès nécessaires à 03:00, si les runbooks sont à jour, si la procédure de failover fonctionne réellement, si les sauvegardes se restaurent et si le plan de communication atteint les clients. Un game day typique de la plateforme de livraison de repas pourrait basculer la base de données des commandes vers son instance de secours pendant un après-midi calme, sous les yeux de toute l'équipe d'astreinte, avec un animateur qui injecte des surprises («la personne qui connaît le runbook est en congé») et un observateur qui note chaque retard et chaque confusion pour la revue qui suivra. Avant une grosse ouverture de billetterie, la même équipe pourrait répéter la perte du prestataire de paiement et le passage au prestataire de secours en pleine vente.

Une astreinte que les personnes peuvent tenir dans la durée

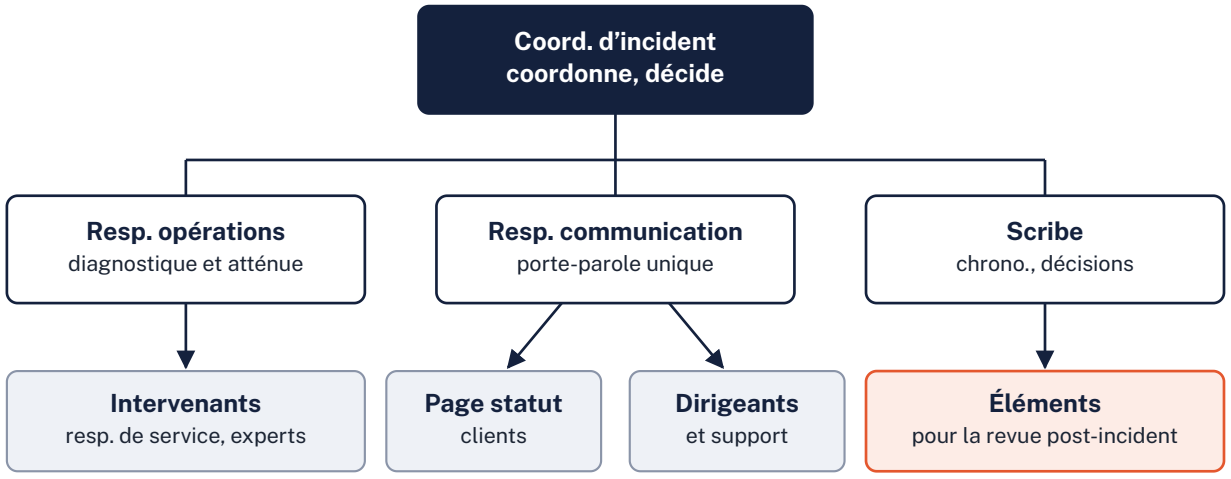
L'astreinte est un système à capacité limitée, et l'épuisement des ingénieurs est l'un de ses modes de défaillance. Les livres de Google sur le SRE donnent des chiffres de référence utiles : au plus 25 % du temps d'un ingénieur en astreinte, un objectif d'au plus deux incidents par période d'astreinte de 12 heures, et un minimum de huit personnes pour un planning 24 heures sur 24 sur un seul site, ou six par site sur deux sites. Ce sont les chiffres d'une seule entreprise, pas des règles, mais ils montrent pourquoi une équipe de quatre personnes ne peut pas couvrir durablement à la fois le pic du soir de la plateforme et la nuit. Les architectes ont un effet direct sur ce point, car chaque composant auto-hébergé alourdit le planning de quelqu'un. Faites en sorte que chaque alerte appelle une action (section 9.6), adaptez les délais de réponse à la gravité, traitez les alertes répétées comme des défauts à corriger, soignez la passation entre les périodes et rémunérez le travail hors horaires.

La gestion des incidents

Un incident est un événement imprévu qui perturbe un service, ou menace de le perturber, et qui exige une réponse coordonnée. La gestion des incidents est la pratique qui consiste à conduire cette réponse de façon rapide, calme et bien communiquée. La plupart des pratiques modernes s'inspirent de l'Incident Command System, conçu pour la lutte contre les feux de forêt en Californie dans les années 1970, et adapté au logiciel dans le chapitre du livre SRE de Google consacré à la gestion des incidents.

L'idée centrale est la séparation des rôles. Le coordinateur d'incident (incident commander) est responsable de l'incident : il fixe les priorités, attribue les rôles, prend les décisions et garde la vue d'ensemble, et il ne débogue délibérément *pas*, car un coordinateur plongé dans un fichier journal ne coordonne plus rien. Le responsable des opérations et les intervenants diagnostiquent et atténuent. Le responsable de la communication est l'unique porte-parole auprès de tous ceux qui sont extérieurs à la réponse : équipes de support, dirigeants et clients, via la page de statut, avec des mises à jour au rythme promis même quand il n'y a rien de nouveau. Un scribe consigne la chronologie, les décisions et les hypothèses. Dans une petite entreprise, une même personne peut tenir plusieurs rôles, mais ces rôles doivent tout de même être nommés. Les rôles sont transmis explicitement lors des changements d'équipe.

Les rôles en cas d'incident



Un seul canal d'incident ; le coordinateur ne débogue pas

Séparer la coordination, le travail technique et la communication permet à chaque rôle de rester concentré.

Plusieurs habitudes font la différence. Déclarez les incidents tôt et sans formalités : mieux vaut lever une fausse alerte que découvrir au bout d'une heure que trois équipes déboguent le même problème chacune de leur côté. Classez chaque incident selon un niveau de gravité défini par l'impact sur les utilisateurs, et non par la cause, pour que chacun sache qui appeler

et avec quelle ampleur communiquer. Atténuez avant de diagnostiquer : annulez le dernier déploiement, basculez, désactivez le feature flag ou délestez la charge d'abord, et cherchez le pourquoi ensuite. Utilisez un seul canal d'incident. Et n'oubliez pas les obligations externes : une violation de données à caractère personnel peut devoir être notifiée à l'autorité de contrôle dans les 72 heures en vertu du RGPD, c'est pourquoi les incidents de sécurité suivent le processus correspondant décrit à la section 8.8.

Niveaux de gravité indicatifs pour la plateforme de livraison de repas

Niveau	Exemple d'impact	Réponse	Communication
SEV1 (critique)	Commandes ou paiement indisponibles pour la plupart des utilisateurs d'une région ; suspicion de violation de données	Alerte immédiate de l'astreinte ; coordinateur d'incident et responsable de la communication désignés sur-le-champ	Page de statut sous 15 minutes ; mises à jour toutes les 30 minutes ; dirigeants informés
SEV2 (majeur)	Un parcours clé fortement dégradé pour de nombreux utilisateurs, par exemple un p99 de l'étape de paiement supérieur à 4 s dans une ville ; consommation rapide du budget du SLO	Alerter l'ingénieur d'astreinte ; coordinateur désigné	Page de statut si les clients peuvent s'en apercevoir ; mises à jour internes toutes les heures
SEV3 (mineur)	Dégradation partielle avec solution de contournement ; un outil interne en panne	Traité pendant les heures ouvrées par l'équipe responsable	Canal de l'équipe ; ticket
SEV4 (faible)	Problème cosmétique ou quasi-incident sans impact sur les utilisateurs	Élément du backlog	Revue facultative si elle apporte un enseignement

Les revues post-incident sans reproche

Après l'incident vient la partie la plus précieuse : l'apprentissage. Une revue post-incident (souvent appelée postmortem) reconstitue ce qui s'est passé, pourquoi cela paraissait logique aux personnes impliquées sur le moment et ce qui va changer. Le chapitre du livre SRE de Google consacré à la culture du postmortem fixe les attentes : un compte rendu écrit pour chaque incident important, couvrant l'impact, la chronologie, les causes contributives et les actions de suivi, relu et partagé, et rédigé sans reproche.

Le billet de John Allspaw de 2012, «Blameless PostMortems and a Just Culture», écrit lorsqu'il dirigeait les opérations techniques chez Etsy, explique pourquoi l'absence de reproche est pragmatique et pas seulement bienveillante. Les ingénieurs qui s'attendent à être sanctionnés

cessent de rendre pleinement compte de ce qu'ils ont vu et fait, si bien que l'organisation perd précisément l'information dont elle a besoin pour prévenir le prochain incident. Allspaw s'appuie sur les sciences de la sécurité, notamment les travaux de Sidney Dekker sur la culture juste, et sur l'idée de *seconds récits* : le premier récit dit «un ingénieur a lancé la mauvaise commande» ; le second demande pourquoi la commande était si facile à lancer, pourquoi l'outil n'a pas averti, pourquoi la procédure semblait raisonnable sous la pression du temps, et met au jour des conditions qui piégeront aussi la personne suivante. Sans reproche ne veut pas dire sans responsabilité : chacun doit toujours rendre pleinement compte, et l'organisation est responsable de corriger les conditions.

Les bonnes revues évitent le biais rétrospectif (le dénouement n'était pas évident sur le moment), les raisonnements contrefactuels («aurait dû», «n'a pas su») et la recherche d'une cause racine unique. Le court article de Richard Cook, «How Complex Systems Fail» (1998), soutient que les défaillances graves des systèmes complexes exigent l'alignement de plusieurs failles : il n'y a donc pas de cause racine unique, seulement des facteurs contributifs. Une chaîne des cinq pourquoi qui aboutit à «l'erreur humaine» s'est arrêtée trop tôt. Notez aussi ce qui a bien fonctionné : l'expertise et l'improvisation qui ont limité les dégâts font partie de ce qui maintient réellement le système en marche, et elles restent invisibles si personne ne les consigne.

SI VOUS DÉBUTEZ

Pas à pas : du « qui » au « comment »

Le point qui fait le plus souvent trébucher ici est qu'une bonne revue post-incident regarde au-delà de la personne qui a commis la dernière erreur, vers les conditions qui l'ont rendue facile, car ces conditions piégeront aussi la personne suivante. Imaginez que le délai d'expiration de l'exemple commenté n'ait pas été repéré par une expérience.

- Un midi, le service de la carte ralentit. Les appels que lui adresse le service de commandes attendent jusqu'à 3 secondes au lieu des 300 ms prévues, ses threads s'épuisent et les paiements échouent dans toute la région.
- Le premier récit s'écrit vite : un développeur a livré le client de la carte avec un délai d'expiration de 3 secondes. L'action qui en découle s'impose d'elle-même : faire plus attention à la configuration.
- Le second récit demande pourquoi l'erreur était facile à commettre et difficile à remarquer. Le développeur avait écrit 300 ms, mais une valeur par défaut de 3000 ms dans une bibliothèque l'avait discrètement écrasée ; rien dans le build ne vérifiait la présence de délais d'expiration explicites ; et la conception indiquait 300 ms, si bien que les relecteurs n'ont rien vu d'anormal.
- Ce sont des facteurs contributifs : plusieurs conditions qui se sont alignées. N'importe quel développeur pourrait tomber dans le même piège ; blâmer celui-ci ne change donc rien.
- Parce que la revue est sans reproche, le développeur peut dire ouvertement ce qu'il a configuré et pourquoi, ce qui conduit l'équipe jusqu'à la valeur par défaut de la bibliothèque.
- Les actions, comme dans l'exemple commenté, changent le système : définir le délai d'expiration dans la configuration partagée du client et ajouter une vérification au build qui échoue si un appel sortant n'a pas de délai d'expiration explicite, chacune avec un responsable et une échéance.
- **La leçon :** « l'erreur humaine » est le point de départ d'une revue, pas son point d'arrivée. Demandez-vous pourquoi chaque action paraissait logique sur le moment et corrigez les conditions ; un rappel à la prudence laisse le piège en place.

Apprendre des incidents, et les limites du MTTR

Les organisations veulent naturellement un chiffre qui montre si la réponse aux incidents s'améliore, et le temps moyen de rétablissement (MTTR) est le candidat habituel. Le chapitre 2 a utilisé le MTTR dans A = MTBF / (MTBF + MTTR), où il a du sens pour un composant unique dont les défaillances se répètent. Comme métrique organisationnelle couvrant de nombreux

incidents différents, il est bien plus faible. Les durées des incidents sont très asymétriques : la plupart sont courts et quelques-uns très longs, si bien que la moyenne est dominée par une poignée de valeurs extrêmes. Les analyses de milliers de rapports d'incidents publics menées par Courtney Nash dans la Verica Open Incident Database (rapports VOID publiés à partir de 2021) ont constaté exactement cette asymétrie, et un rapport de Google de 2021 signé Štěpán Davidovič, *Incident Metrics in SRE*, a montré par simulation qu'avec un nombre réaliste d'incidents, les variations du MTTR sont surtout du bruit et ne permettent pas de savoir de façon fiable si une amélioration a fonctionné.

De meilleurs signaux sont l'impact sur les SLO et les budgets d'erreur (qui pondèrent les incidents selon ce qu'ont ressenti les utilisateurs), les thèmes récurrents d'une revue à l'autre (le même type de modification de configuration qui pose problème trois fois), les quasi-incidents, la rapidité de détection des incidents, la fréquence à laquelle les revues aboutissent à des modifications du système menées à bien, et des éléments qualitatifs, par exemple le fait que les ingénieurs trouvent les revues utiles. De nombreuses organisations publient les revues en interne dans une bibliothèque consultable et tiennent régulièrement des réunions de revue des incidents, afin que la leçon d'une équipe profite aux autres.

Des actions qui changent réellement le système

Une revue ne vaut que par les changements qu'elle produit. Les actions diffèrent fortement par leur efficacité. Les plus fortes sont celles qui suppriment le danger (supprimer la commande dangereuse, rendre l'état non sûr impossible), puis les garde-fous qui empêchent l'erreur de se propager (validation de la configuration, déploiements progressifs et canaris, limites de débit sur les modifications, retour arrière automatique en cas de consommation du SLO), puis une détection et une atténuation plus rapides (une nouvelle alerte de SLO, un runbook testé, un failover automatisé). Les plus faibles sont la formation, la documentation et les rappels. «Faire plus attention» n'est pas une action corrective ; cela ne change rien aux conditions qui attendent la personne suivante.

Chaque action a besoin d'un responsable, d'une priorité et d'une échéance, et doit figurer dans le backlog habituel de l'équipe plutôt que dans un document oublié. La politique de budget d'erreur de la section 9.5 fournit le levier : un incident qui a consommé une large part du budget fait passer ses correctifs avant les fonctionnalités. Gardez la liste courte ; cinq actions terminées valent mieux que vingt actions ouvertes. Lorsqu'une action modifie l'architecture, par exemple en ajoutant un cloisonnement (bulkhead) ou en supprimant une dépendance synchrone, consignez-la dans un enregistrement de décision d'architecture qui cite l'incident, et liez les runbooks (procédures d'exploitation) pertinents aux alertes qui se déclencheront la prochaine fois.

Liste de contrôle : mener une revue post-incident

Utilisez-la pour chaque incident au-delà d'un niveau de gravité convenu, pour chaque incident qui consomme plus d'une part définie d'un budget d'erreur, et pour les quasi-incidents instructifs.

- [ ] **La programmer dans les quelques jours ouvrés**
  Les souvenirs s'estompent vite ; les journaux et l'historique des discussions peuvent expirer. Réservez la présence des personnes impliquées, d'un animateur qui n'était pas impliqué et des responsables des services touchés.
- [ ] **Construire d'abord la chronologie à partir des preuves**
  Rassemblez les alertes, les déploiements, les messages de discussion, les tableaux de bord et les traces dans une chronologie avant la réunion, pour que la discussion parte des faits plutôt que des souvenirs.
- [ ] **Décrire l'impact du point de vue des utilisateurs**
  Qui a été touché, combien de personnes, pendant combien de temps, quels SLO et quelle part du budget d'erreur ; incluez l'exposition financière, l'exposition des données et l'exposition réglementaire le cas échéant.
- [ ] **Poser explicitement le cadre sans reproche**
  Commencez en disant que le but est de comprendre comment le système et le travail fonctionnent réellement, et que personne ne sera sanctionné pour avoir décrit ce qu'il a fait et pourquoi.
- [ ] **Demander pourquoi cela paraissait logique sur le moment**
  Pour chaque décision clé, demandez ce que la personne savait, ce qu'elle essayait de faire et ce que les outils lui montraient. Évitez les raisonnements contrefactuels comme « aurait dû » et « n'a pas su ».
- [ ] **Chercher plusieurs facteurs contributifs**
  Des conditions techniques, organisationnelles et procédurales réunies ; résistez à la tentation de vous arrêter à la première « cause racine » unique ou à l'erreur humaine.
- [ ] **Noter ce qui a bien fonctionné**
  Consignez les adaptations, l'expertise et la chance qui ont limité les dégâts ; elles font partie de ce qui maintient réellement le système en marche.
- [ ] **Convenir de quelques actions qui changent le système**
  Préférez la suppression des dangers et l'ajout de garde-fous à la formation et aux rappels ; attribuez à chaque action un responsable, une priorité et une échéance, et suivez-la dans le backlog habituel.

- [ ] **Publier et partager**
  Rédigez la revue selon un modèle standard, diffusez-la largement, ajoutez-la à une bibliothèque consultable et réexaminez les actions ouvertes lors d'une réunion de revue régulière.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi une expérience de chaos a-t-elle besoin d'une hypothèse d'état stable plutôt que d'un simple « voyons ce qui casse » ?

Parce qu'une prédiction mesurable sur le comportement normal visible des utilisateurs indique si le système a tenu, fournit des conditions d'arrêt et transforme l'exercice en preuve plutôt qu'en anecdote.

▼ Pourquoi le coordinateur d'incident doit-il éviter de déboguer ?

Parce que son rôle est la coordination : priorités, rôles, décisions et vue d'ensemble. Un coordinateur absorbé par les détails techniques ne laisse personne à la barre de la réponse.

▼ Qu'est-ce qu'un « second récit » dans une revue post-incident ?

Le récit qui va au-delà de la personne qui a commis une erreur, vers les conditions, les outils, les pressions et la conception qui ont fait paraître l'action raisonnable et qui piégeront aussi la personne suivante.

▼ Pourquoi le MTTR moyen sur une année d'incidents est-il une mesure faible de l'amélioration ?

Les durées des incidents sont très asymétriques, si bien que quelques longs incidents dominent la moyenne, et qu'avec un nombre réaliste d'incidents ses variations sont surtout du bruit.

▼ **À vous : Chaque instance de commandes a 30 threads et reçoit 20 commandes par seconde, chacune occupant normalement un thread pendant 0,2 s. L'appel à la carte devrait expirer après 300 ms et utiliser les prix en cache. Vous injectez 5 s de latence. Prédisez les résultats.**

Normalement, 20 x 0,2 = 4 threads sont occupés. Avec le délai d'expiration, chaque commande occupe un thread pendant environ 0,5 s : 10 sont donc occupés, et les commandes survivent grâce aux prix en cache. Si le délai d'expiration manque, chacune occupe un thread pendant environ 5,2 s, ce qui exige 104 threads contre 30 : l'instance les épuise donc. Commencez par une seule instance, fixez des conditions d'arrêt sur le taux de réussite des commandes et corrigez toute faiblesse trouvée.

SOURCES

- Principles of Chaos Engineering: https://principlesofchaos.org/
- Netflix Technology Blog: The Netflix Simian Army (2011): https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116
- John Allspaw: Blameless PostMortems and a Just Culture (Etsy, 2012): https://www.etsy.com/codeascraft/blameless-postmortems
- Google SRE book: Postmortem Culture: Learning from Failure: https://sre.google/sre-book/postmortem-culture/
- Google SRE: Incident Metrics in SRE (Štěpán Davidovič): https://sre.google/resources/practices-and-processes/incident-metrics-in-sre/
- Being On-Call (Site Reliability Engineering, chapter 11): https://sre.google/sre-book/being-on-call/
- On-Call (The Site Reliability Workbook, chapter 8): https://sre.google/workbook/on-call/

Lectures complémentaires

- *Release It! Design and Deploy Production-Ready Software (2nd edition)*, Michael T. Nygard (2018), Pragmatic Bookshelf. Le livre qui a nommé les patrons disjoncteur et cloisonnement ; ses patrons et anti-patrons de stabilité, illustrés par de vraies pannes, sont la base de la section 9.4.
- *The Art of Scalability (2nd edition)*, Martin L. Abbott and Michael T. Fisher (2015), Addison-Wesley. Présente le cube de scalabilité et relie les choix techniques de passage à l'échelle à l'organisation et aux processus.
- *Site Reliability Engineering: How Google Runs Production Systems*, Betsy Beyer, Chris Jones, Jennifer Petoff and Niall Richard Murphy (editors) (2016), O'Reilly Media. Gratuit en ligne ; la source des SLO et des budgets d'erreur tels qu'on les pratique aujourd'hui, avec d'excellents chapitres sur la surcharge, les défaillances en cascade et les post-mortems.
  https://sre.google/sre-book/table-of-contents/
- *The Site Reliability Workbook: Practical Ways to Implement SRE*, Betsy Beyer, Niall Richard Murphy, David K. Rensin, Kent Kawahara and Stephen Thorne (editors) (2018), O'Reilly Media. Le compagnon pratique : mise en œuvre des SLO, alertes multifenêtres sur le taux de consommation et astreinte, avec des exemples commentés.
  https://sre.google/workbook/table-of-contents/
- *Exponential Backoff And Jitter*, Marc Brooker (2015), AWS Architecture Blog. Un court article fondé sur des simulations, qui montre pourquoi les nouvelles tentatives ont besoin de délais aléatoires ; l'origine de la formule « full jitter », largement utilisée.
  https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/
- *Systems Performance: Enterprise and the Cloud (2nd edition)*, Brendan Gregg (2020), Addison-Wesley. La référence sur l'analyse des performances, y compris la méthode USE, les outils d'observabilité et les pièges des benchmarks.
- *Chaos Engineering: System Resiliency in Practice*, Casey Rosenthal and Nora Jones (2020), O'Reilly Media. Écrit par des praticiens de Netflix et d'ailleurs ; explique les principes de l'ingénierie du chaos et la manière dont les organisations l'adoptent.
- *Fail at Scale: Reliability in the face of rapid change*, Ben Maurer (2015), ACM Queue, vol. 13, no. 8. Comment Facebook gère la surcharge et les dépendances lentes avec des files à délai contrôlé, un LIFO adaptatif et des limites de concurrence côté client.
  https://queue.acm.org/detail.cfm?id=2839461

Glossaire

Alerte multifenêtre et multi-taux de consommation
Alerte de SLO qui ne se déclenche que lorsque le taux de consommation dépasse un seuil à la fois sur une fenêtre longue et sur une fenêtre courte, ce qui combine une détection rapide, une réinitialisation rapide et peu de fausses alertes.

Anycast
Annonce de la même adresse IP depuis de nombreux emplacements, afin que le routage d'internet achemine les paquets de chaque utilisateur vers l'emplacement le plus proche qui l'annonce.

Autoscaling (mise à l'échelle automatique)
Modification automatique du nombre d'instances en fonction de la charge, de façon réactive à partir d'une métrique, selon un calendrier, ou de façon prédictive à partir d'une prévision.

Backoff exponentiel
Allongement de l'attente entre les nouvelles tentatives successives, généralement en doublant le délai à chaque fois jusqu'à un plafond, afin qu'un service en difficulté ait le temps de se rétablir.

Budget d'erreur (error budget)
Niveau de non-fiabilité qu'un SLO autorise : un moins l'objectif, appliqué à la fenêtre. Un SLO de 99,9 % sur 1 000 000 requêtes autorise 1000 requêtes en échec.

Budget de nouvelles tentatives (retry budget)
Limite des nouvelles tentatives exprimée en proportion du trafic normal (par exemple 10 %), afin qu'elles aident face aux défaillances occasionnelles sans pouvoir multiplier la charge pendant une panne.

Budget de performance
Limite convenue sur une mesure de performances, comme le poids d'une page ou la latence p99 par appel de service, qu'un changement ne doit pas dépasser, idéalement vérifiée automatiquement.

Cardinalité
Nombre de valeurs distinctes qu'un champ peut prendre ; pour les métriques, nombre de combinaisons distinctes d'étiquettes, chacune devenant une série temporelle distincte à stocker.

Cloisonnement (bulkhead)
Partitionnement de ressources comme les pools de threads, les pools de connexions ou les instances, afin qu'une défaillance ou une surcharge dans une partie ne puisse pas consommer les ressources des autres.

Compteur (counter)
Métrique dont la valeur ne fait qu'augmenter (ou revient à zéro au redémarrage), comme le nombre total de requêtes servies ; on examine généralement son taux d'augmentation.

Consommateurs concurrents (competing consumers)
Patron dans lequel plusieurs workers lisent la même file et chaque message n'est traité que par l'un d'eux, de sorte que le débit augmente en ajoutant des workers.

Contrôle de santé (health check)
Test périodique, comme une requête HTTP vers un endpoint d'état, qu'un équilibreur de charge ou un orchestrateur utilise pour décider si une instance doit recevoir du trafic ou être redémarrée.

Cooldown (période de stabilisation)
Période qui suit une action de mise à l'échelle, pendant laquelle l'autoscaler attend avant d'agir à nouveau, afin de pouvoir mesurer l'effet de la dernière modification.

Coordinateur d'incident (incident commander)
Personne qui coordonne la réponse à un incident : elle fixe les priorités, attribue les rôles et prend les décisions, plutôt que de réparer elle-même.

Crédit de service
Remboursement ou remise dû à un client au titre d'un SLA lorsque le fournisseur n'atteint pas un niveau de service promis.

Culture sans reproche (blameless)
Approche dans laquelle les personnes impliquées dans un incident peuvent décrire ce qu'elles ont fait et pourquoi sans crainte de sanction, afin que l'organisation apprenne comment le système fonctionne réellement.

Délai d'expiration (timeout)
Le temps maximal qu'un appelant attend une opération avant d'abandonner et de la considérer comme échouée ; chaque appel distant en a besoin.

Délestage de charge (load shedding)
Rejeter délibérément certaines requêtes, à moindre coût et tôt, lorsqu'un système est surchargé, afin que les requêtes qu'il accepte aboutissent encore dans les temps.

Disjoncteur (circuit breaker)
Enveloppe autour des appels à une dépendance qui, après un nombre suffisant de défaillances, cesse de l'appeler pendant un temps et échoue rapidement, puis laisse passer des appels d'essai pour tester le rétablissement (Michael Nygard, 2007).

Drainage des connexions (connection draining)
Arrêt de l'envoi de nouvelles requêtes à une instance en cours de retrait, tout en laissant se terminer les requêtes déjà en cours, afin que la réduction du nombre d'instances et les déploiements ne perdent aucun travail.

Échantillonnage des traces
Ne conserver qu'une partie des traces pour maîtriser les coûts : l'échantillonnage en tête (head-based) décide au début d'une requête, l'échantillonnage en queue (tail-based) décide après sa fin, ce qui lui permet de garder les requêtes lentes et en échec.

Équilibrage de charge global
Répartition des utilisateurs entre centres de données ou régions, généralement par des réponses DNS qui varient selon l'emplacement et l'état de santé, ou par routage anycast.

Équilibreur de charge de couche 4
Équilibreur de charge qui opère au niveau transport (TCP ou UDP) et transmet les connexions ou les paquets sans lire la requête HTTP qu'ils contiennent.

Équilibreur de charge de couche 7
Équilibreur de charge qui comprend le protocole applicatif, généralement HTTP, et peut donc router chaque requête selon le chemin, un en-tête ou un cookie, terminer TLS et effectuer de nouvelles tentatives.

Facteur contributif
L'une des différentes conditions qui, ensemble, ont rendu un incident possible ou l'ont aggravé ; les revues recherchent de nombreux facteurs contributifs plutôt qu'une cause racine unique.

Fenêtre glissante
Période de mesure qui avance en continu, comme les 28 derniers jours, par opposition à un mois civil qui repart à zéro le premier du mois.

Flapping (oscillation)
Situation où un autoscaler ajoute et retire de la capacité à répétition parce qu'il réagit à des changements de courte durée ou aux effets de ses propres actions.

Game day (journée d'exercice)
Exercice planifié au cours duquel une équipe provoque ou simule délibérément une défaillance et s'entraîne à la détecter, à y répondre et à s'en rétablir.

Gauge (jauge)
Métrique qui enregistre une valeur courante pouvant monter ou descendre, comme la longueur d'une file, la mémoire utilisée ou la température.

Histogramme
Métrique qui répartit les observations, comme la durée des requêtes, dans des plages de valeurs appelées buckets, afin de pouvoir estimer des percentiles et additionner les histogrammes de nombreux serveurs.

Hypothèse d'état stable
Dans une expérience de chaos, la prédiction qu'un signe mesurable de comportement normal, comme le nombre de commandes par minute, restera dans certaines limites pendant l'injection d'une défaillance.

Ingénierie du chaos
Expérimentation sur un système, idéalement en production et de manière maîtrisée, en injectant des défaillances réalistes afin d'acquérir la confiance qu'il résiste à des conditions turbulentes.

Jitter (gigue aléatoire)
Part d'aléatoire ajoutée aux délais entre nouvelles tentatives ou aux plannings, afin que de nombreux clients n'agissent pas au même moment et ne créent pas de pics synchronisés.

Journalisation structurée (structured logging)
Écrire les entrées de journal sous forme d'enregistrements clé-valeur lisibles par machine (souvent en JSON), avec des noms de champs cohérents, au lieu de phrases en texte libre.

Leaky bucket (seau percé)
Algorithme de limitation de débit ou de mise en forme du trafic dans lequel les requêtes entrent dans un seau et en sortent à un rythme constant ; lorsque le seau est plein, les nouvelles requêtes sont rejetées.

Limitation de débit (rate limiting)
Restreindre le nombre de requêtes qu'un client, une clé ou un locataire (tenant) peut faire sur une période, afin de protéger la capacité et de la partager équitablement ; les requêtes en excès reçoivent généralement une réponse HTTP 429.

Marge de capacité (headroom)
Capacité de réserve maintenue délibérément au-dessus du pic de charge prévu, pour absorber les rafales, les défaillances et le temps nécessaire pour ajouter de la capacité.

Méthode RED
La liste de contrôle de Tom Wilkie pour superviser les services pilotés par les requêtes : Rate (débit), Errors (erreurs) et Duration (durée) des requêtes.

Méthode USE
La liste de contrôle de Brendan Gregg pour analyser les ressources : pour chaque ressource, vérifier l'utilisation (Utilisation), la saturation (Saturation) et les erreurs (Errors).

Modèle de charge fermé
Modèle de charge avec un nombre fixe d'utilisateurs, chacun n'envoyant une nouvelle requête qu'après avoir reçu la réponse à la précédente (plus un temps de réflexion), de sorte que la charge baisse quand le système ralentit.

Modèle de charge ouvert
Modèle de charge dans lequel les nouvelles requêtes arrivent à un rythme fixé, quelle que soit la rapidité avec laquelle les précédentes reçoivent une réponse, comme des clients qui arrivent sur un site web.

Moins de connexions (least connections)
Algorithme d'équilibrage de charge qui envoie chaque nouvelle requête à l'instance ayant le moins de requêtes ou de connexions en cours.

Niveau de gravité
Classification prédéfinie de la gravité d'un incident, par exemple de SEV1 à SEV4, qui détermine qui est appelé, avec quelle rapidité et avec quelle diffusion il est communiqué.

Omission coordonnée (coordinated omission)
Erreur de mesure, nommée par Gil Tene, dans laquelle un générateur de charge attend les réponses lentes avant d'envoyer d'autres requêtes et n'enregistre donc jamais les délais que de vrais utilisateurs auraient subis.

OpenTelemetry
Projet open source de la CNCF, indépendant des fournisseurs, qui fournit des API, des SDK, un protocole (OTLP) et un collecteur pour produire et exporter des traces, des métriques et des journaux.

Page de statut
Page publique ou destinée aux clients qui signale les incidents en cours et l'état de santé du service, mise à jour par la personne chargée de la communication pendant l'incident.

Planification de la capacité
Prévision de la demande future, mesure de ce que chaque unité d'infrastructure peut traiter et mise en place d'une capacité suffisante, avec une marge, avant qu'elle ne soit nécessaire.

Politique de budget d'erreur
Énoncé écrit et convenu de ce qu'une équipe fera lorsque son budget d'erreur est dépensé trop vite ou épuisé, comme le gel des mises en production risquées.

Propagation des échéances (deadline propagation)
Transmission du temps restant pour l'ensemble d'une requête à chaque appel en aval, afin qu'aucun service ne continue à travailler sur une requête que son appelant a déjà abandonnée.

Propagation du contexte
Transmission des identifiants de trace (et d'autres éléments du contexte de la requête) d'un service au suivant avec chaque appel ou message, par exemple dans l'en-tête W3C traceparent.

Puissance de deux choix aléatoires (power of two choices)
Méthode d'équilibrage de charge qui choisit deux instances au hasard et envoie la requête à la moins chargée ; Mitzenmacher a montré qu'elle équilibre presque aussi bien que la vérification de chaque instance.

Quatre signaux d'or (four golden signals)
Latence, trafic, erreurs et saturation : les quatre mesures que le livre SRE de Google recommande de superviser pour tout système en contact avec les utilisateurs.

Rayon d'impact (blast radius)
Part du système et nombre d'utilisateurs qu'une défaillance, une modification ou une expérience peut affecter ; les expériences de chaos le maintiennent délibérément réduit.

Requête de couverture (hedged request)
Envoi d'une seconde copie d'une requête lente à une autre réplique après un court délai, en utilisant la première réponse qui arrive, afin de réduire la latence de queue.

Revue post-incident
Réunion structurée et rapport écrit, après un incident, qui reconstituent ce qui s'est passé, pourquoi cela paraissait logique aux personnes impliquées et ce qui va changer ; on parle aussi de postmortem.

Service sans état
Instance de service qui ne conserve aucune donnée entre deux requêtes dans sa propre mémoire ni sur son disque, de sorte que n'importe quelle instance peut traiter n'importe quelle requête et que l'on peut ajouter ou retirer des instances librement.

Session persistante (sticky session, affinité de session)
Réglage d'un équilibreur de charge qui envoie toutes les requêtes d'un même client vers la même instance, généralement pour que les données de session conservées dans la mémoire de cette instance puissent être retrouvées.

Sonde de disponibilité (readiness probe)
Contrôle de santé qui vérifie si une instance peut servir du trafic à cet instant ; un échec retire l'instance de l'équilibreur de charge sans la redémarrer.

Sonde de vivacité (liveness probe)
Contrôle de santé qui vérifie si un processus est bloqué sans possibilité de rétablissement ; un échec amène l'orchestrateur à redémarrer l'instance.

Span
Une opération chronométrée au sein d'une trace, comme un appel HTTP ou une requête de base de données, avec un nom, une heure de début, une durée, des attributs et un lien vers son span parent.

Supervision (monitoring)
Collecter et surveiller des mesures et des contrôles définis à l'avance afin que les types de défaillance connus soient détectés et signalés.

Taux de consommation (burn rate)
Vitesse à laquelle un budget d'erreur est dépensé par rapport au rythme qui l'épuiserait exactement à la fin de la fenêtre ; un taux de consommation de 1 le dépense entièrement en exactement une fenêtre.

Télémétrie
Les données qu'un système en fonctionnement émet sur son propre comportement : journaux, métriques, traces, profils et événements.

Tempête de nouvelles tentatives (retry storm)
Afflux de charge créé lorsque de nombreux clients, souvent à plusieurs niveaux, refont en même temps des requêtes qui échouent, ce qui maintient en surcharge un service déjà surchargé.

Test d'endurance (soak test)
Long test de performances sous charge normale, exécuté pendant des heures ou des jours, pour révéler des problèmes lents comme les fuites de mémoire ou les files qui grossissent ; on parle aussi de test d'endurance.

Test de charge
Test de performances qui applique la charge attendue (ou le pic prévu) pendant une période donnée afin de vérifier que les objectifs de latence et d'erreurs sont atteints.

Test de pic (spike test)
Test de performances qui augmente brusquement la charge, par exemple d'un facteur dix en quelques secondes, pour vérifier l'autoscaling, les files et les mécanismes de protection.

Test de point de rupture
Test de performances qui augmente la charge régulièrement jusqu'à ce que le système cède, afin de trouver sa capacité maximale et son premier goulot d'étranglement.

Test de stress
Test de performances qui pousse la charge au-delà du pic attendu pour voir comment le système défaille et s'il se rétablit.

Token bucket (seau à jetons)
Algorithme de limitation de débit dans lequel des jetons sont ajoutés à un seau à un rythme fixe jusqu'à une capacité maximale, et où chaque requête consomme un jeton ; il autorise des rafales jusqu'à cette capacité et un débit moyen régulier.

Traçage distribué (distributed tracing)
Enregistrement du chemin et de la durée d'une requête au fil de son passage par de nombreux services, afin de pouvoir trouver l'étape lente ou défaillante.

Utilisateur virtuel
Utilisateur simulé dans un outil de test de charge, qui exécute un script de requêtes, généralement en boucle.