

BE SOFTWARE ARCHITECTURE WISE · CHAPITRE 10

Architecture cloud native et d’infrastructure

L’endroit où un logiciel s’exécute, et la manière dont il s’exécute, sont désormais une décision d’architecture. Ce chapitre traite des modèles de services et de déploiement cloud et du modèle de responsabilité partagée, des conteneurs et de Kubernetes, de l’application douze facteurs, de l’infrastructure en tant que code et de GitOps, des pipelines CI/CD et des stratégies de déploiement, de la conception multirégion, de la reprise après sinistre et de l’edge computing, de l’ingénierie de plateforme, ainsi que du FinOps et de la dépendance au fournisseur. Il s’adresse aux développeurs et aux architectes qui connaissent les qualités opérationnelles du chapitre 9 et veulent choisir, justifier et exploiter les plateformes dont dépendent leurs systèmes, sans se laisser vendre une pile toute faite.

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

EN UN COUP D'ŒIL

Vous apprendrez à

- Comparer IaaS, CaaS, PaaS, FaaS et SaaS ainsi que les modèles de déploiement, et répartir les obligations relatives à une charge de travail selon le modèle de responsabilité partagée.
- Expliquer comment Kubernetes maintient un état souhaité déclaré grâce à son plan de contrôle, à ses nœuds et à ses boucles de réconciliation, configurer les requêtes, les limites et les sondes, et juger quand Kubernetes est surdimensionné pour le besoin.
- Auditer une application au regard des douze facteurs et des principes qui vont au-delà, et prioriser les corrections selon le risque.
- Concevoir du code d'infrastructure avec un état distant, du verrouillage, des modules, des environnements, des contrôles de politiques, des tests et une réconciliation GitOps.
- Choisir une stratégie de déploiement (progressif, bleu-vert, canari ou fantôme) et séparer le déploiement de la mise en production grâce aux feature flags.
- Choisir une stratégie de reprise après sinistre en fonction du RTO, du RPO et du coût, et expliquer les conséquences de la réplication des données entre régions et du déplacement de traitements vers la périphérie (edge).
- Expliquer comment une plateforme interne de développement dotée de golden paths réduit la charge cognitive, et comment mesurer si elle réussit.
- Calculer des coûts unitaires comme le coût par commande, identifier les facteurs architecturaux du coût du cloud et évaluer la dépendance au fournisseur comme un compromis chiffré.

Avant de commencer

Chapitre 0, section 0.7 (les serveurs et le cloud) et section 0.8 (comment se construit un logiciel) ; chapitre 9, en particulier le passage à l'échelle, l'absence d'état, les SLO et l'observabilité. Le chapitre 6 sur les systèmes distribués est utile.

Durée

Environ 3 heures et demie de lecture ; 45 minutes pour le test.

Termes clés

cloud native · modèle de responsabilité partagée · zone de disponibilité · image de conteneur · Kubernetes · boucle de réconciliation · application douze facteurs · infrastructure en tant que code · GitOps · canary release (déploiement canari) · RTO et RPO · golden path (chemin balisé) · FinOps · dépendance au fournisseur (lock-in)

Pourquoi l'infrastructure est devenue de l'architecture

Pendant la plus grande partie de l'histoire du logiciel, l'infrastructure venait après l'architecture : une équipe d'exploitation recevait une application terminée et trouvait des serveurs pour la faire tourner. Cette séparation a disparu. Dans un système moderne, le choix entre machines virtuelles, conteneurs, plateformes et fonctions détermine une grande partie

du coût ; le nombre de zones de disponibilité et de régions détermine la disponibilité ; le pipeline de déploiement détermine à quelle vitesse et avec quelle sécurité une équipe peut changer quoi que ce soit ; et l'infrastructure elle-même est écrite sous forme de code, revue dans des pull requests et versionnée avec l'application. Les décisions sur l'endroit et la manière dont un logiciel s'exécute comptent désormais parmi les plus significatives que prend un architecte, exactement au sens du chapitre 1 : elles coûtent cher à défaire, elles concernent toutes les équipes et elles fixent les attributs de qualité que le chapitre 2 vous a demandé de spécifier.

Ce chapitre s'appuie sur deux chapitres précédents. Le chapitre 0 a présenté les serveurs, les machines virtuelles, les conteneurs, les modèles de services cloud, les régions et les zones de disponibilité (les serveurs et le cloud). Le chapitre 9 a expliqué comment faire passer les systèmes à l'échelle, les faire se dégrader gracieusement et les garder observables (à partir des stratégies de passage à l'échelle). Ici, ces idées rencontrent les plateformes sur lesquelles tournent aujourd'hui la plupart des systèmes. L'exemple récurrent est la plateforme de livraison de repas utilisée tout au long du cours : applications pour les clients et les restaurants, service des commandes, cartes, paiements, affectation des livreurs et un coup de feu du soir très marqué.

Comment les sections s'articulent

Les quatre premières sections portent sur les fondations. Modèles de services et de déploiement cloud établit qui gère quoi en IaaS, en conteneurs en tant que service, en PaaS, en FaaS et en SaaS, présente le modèle de responsabilité partagée, les vraies raisons du cloud privé, hybride et du multicloud, ce que la CNCF entend par cloud native, et les cas où louer ne revient pas moins cher que posséder. Conteneurs et orchestration explique les images et les normes OCI, le plan de contrôle et les nœuds de Kubernetes, ses objets de base et la boucle de réconciliation qui en est le cœur, avec du vrai YAML, et dit franchement quand Kubernetes est surdimensionné. L'application douze facteurs passe en revue les douze facteurs avec leurs limites actuelles, les principes qui vont au-delà, et l'audit d'un service existant. Infrastructure en tant que code traite des outils déclaratifs (Terraform, OpenTofu, CloudFormation, Bicep, Pulumi), de l'état, de la dérive, des modules, de l'infrastructure immuable, de GitOps, des politiques en tant que code et des tests.

Les quatre sections suivantes portent sur le changement, l'échelle, les personnes et l'argent. Pipelines CI/CD et stratégies de déploiement suit un changement du commit jusqu'à la production, compare les déploiements progressifs, bleu-vert, canari et fantômes, et sépare le déploiement de la mise en production grâce aux feature flags. Multirégion, reprise après sinistre et edge computing compare les stratégies de reprise après sinistre selon le RTO, le RPO et le coût, explique les conséquences de la réplication des données entre régions sur la cohérence, et traite de l'architecture cellulaire et des fonctions en périphérie. Ingénierie de

plateforme traite la plateforme interne de développement comme un produit, avec des golden paths et des garde-fous, et montre comment mesurer si elle aide. FinOps, architecture des coûts et dépendance au fournisseur fait du coût une qualité architecturale, mesurée par commande, et traite la dépendance au fournisseur comme un compromis à chiffrer plutôt que comme un péché à éviter.

Chaque section peut se lire seule, mais l'ordre est voulu : on ne peut pas raisonner sur les stratégies de déploiement sans savoir ce que fait Kubernetes pendant un déploiement, ni sur l'ingénierie de plateforme sans code d'infrastructure ni pipelines à partir desquels construire la plateforme.

Une idée traverse toutes les sections

L'idée la plus importante de ce chapitre est celle d'un **état souhaité déclaré, réconcilié en continu par l'automatisation**. Kubernetes maintient le nombre de copies en cours d'exécution égal au nombre que vous avez déclaré. Le code d'infrastructure décrit les réseaux et les bases de données, et un outil calcule puis applique la différence avec ce qui existe. GitOps maintient le cluster identique à ce que contient un dépôt Git. Un pipeline de déploiement transforme chaque changement en une même suite d'étapes, testée et reproductible. L'infrastructure immuable remplace au lieu de réparer. Dans chaque cas, une personne modifie une description, qui est revue, versionnée et auditable, et les machines font le reste. C'est ce qui permet à une petite équipe d'exploiter un grand système en toute sécurité : le changement devient routinier, et le rétablissement devient une simple réexécution plutôt qu'un acte d'héroïsme.

La seconde idée est que chaque option proposée dans ce chapitre est un compromis, et que le bon choix dépend de la charge de travail et de l'organisation. Kubernetes est excellent pour de nombreux services et de nombreuses équipes, et un gaspillage pour trois services et deux développeurs. L'actif-actif multirégion achète des minutes de temps de reprise pour un multiple du coût. Une équipe plateforme n'aide que si les développeurs veulent ce qu'elle construit. Le cloud est le moins cher pour une charge variable et peut être battu par du matériel en propre pour une charge importante et régulière. La dépendance au fournisseur est un coût à mettre en balance avec le coût de la portabilité. Le chapitre vous donne donc des guides de décision, des exemples commentés chiffrés et les arguments des deux camps, plutôt qu'une pile unique recommandée.

Comment utiliser ce chapitre

Si l'exploitation est nouvelle pour vous, lisez d'abord les encadrés pour débutants et la section du chapitre 0 sur les serveurs et le cloud, et ne cherchez pas à mémoriser les noms d'outils : les produits changent tous les quelques ans, alors que les mécanismes (réconciliation, immuabilité, état et dérive, livraison progressive, objectifs de reprise, coûts unitaires) durent.

Les ingénieurs expérimentés voudront peut-être commencer par les exemples commentés et les guides de décision, puis lire les sections dont ils contestent les compromis. Tout au long du chapitre, des fournisseurs comme AWS, Microsoft Azure, Google Cloud et Cloudflare sont cités à titre d'exemples, non de recommandations.

Le chapitre 11 traite ensuite de l'évaluation, de la gouvernance et de la modernisation : comment juger une architecture, infrastructure comprise, et comment la faire évoluer sans risque dans le temps.

SECTION 10.1 SUR 10.8 · INTERMÉDIAIRE · 34 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Modèles de services et de déploiement cloud

AVANT DE LIRE

Cette section s'appuie sur :

- 0.7 Serveurs et cloud
- 2.5 Disponibilité et fiabilité

Termes déjà vus qu'elle utilise : Machine virtuelle (VM), Conteneur, Élasticité, Failover (basculement)

L'IDÉE EN BREF

Pensez à la manière de trouver un logement. Vous pouvez acheter un terrain et bâtir une maison, louer un appartement vide et le meubler, louer un appartement meublé ou séjourner à l'hôtel. Chaque étape confie davantage de travail à quelqu'un d'autre et vous laisse moins de choix ; l'hôtel est merveilleux pour une semaine et ruineux pour dix ans. Faire tourner un logiciel, c'est pareil : vous pouvez posséder des serveurs, louer des machines nues, louer une plateforme prête à l'emploi ou louer un logiciel fini. Quel que soit votre choix, c'est toujours vous qui fermez votre porte à clé : vos données et les personnes qui peuvent y accéder restent votre affaire.

CE QU'IL FAUT SAVOIR

- La publication NIST SP 800-145 définit trois modèles de services (IaaS, PaaS, SaaS) et quatre modèles de déploiement (public, privé, communautaire, hybride) ; les conteneurs et les fonctions en tant que service comblent les vides entre eux.
- Chaque étape de l'IaaS vers le SaaS retire du travail d'exploitation et du contrôle ; la question utile est de savoir quelles tâches concrètes (correctifs, passage à l'échelle, sauvegardes, basculement, mises à niveau) restent les vôtres.
- Dans le modèle de responsabilité partagée, le fournisseur sécurise le cloud lui-même et le client sécurise ce qu'il y met ; les données, les identités et la configuration relèvent toujours du client.
- Le cloud privé, hybride et le multicloud ont des usages légitimes (réglementation, charge régulière, migrations, services distinctifs, plans de sortie), mais rendre chaque charge de travail portable entre plusieurs clouds coûte cher.
- Répartissez les charges de travail sur au moins deux zones de disponibilité, de préférence trois ; sachez si chaque dépendance est zonale, régionale ou mondiale.

- La CNCF définit le cloud native comme un ensemble de pratiques permettant de construire de façon programmatique et reproductible des systèmes faiblement couplés, sécurisés, résilients, gérables, durables et observables ; il s'agit du comment, pas du où.
- Le cloud est le moins cher pour une charge variable ; pour une charge importante et régulière, du matériel en propre peut revenir moins cher, comme l'a soutenu 37signals, une fois la marge de capacité, le personnel et les remises d'engagement honnêtement comptés.

D'une définition à de nombreux modèles

Le chapitre 0 a présenté les serveurs et le cloud : la définition du NIST (Special Publication 800-145, 2011), avec ses cinq caractéristiques essentielles (libre-service à la demande, large accès réseau, mutualisation des ressources, élasticité rapide et service mesuré), et l'idée que les modèles de services confient une part plus ou moins grande de la pile à un fournisseur. Cette section approfondit ces idées autant qu'un architecte en a besoin pour choisir où un système s'exécute et pour défendre ce choix.

Le NIST définit trois modèles de services (infrastructure, plateforme et logiciel en tant que service) et quatre modèles de déploiement : **public** (ouvert à tous, détenu par un fournisseur), **privé** (réservé à une seule organisation, quel que soit son emplacement et quel que soit son exploitant), **communautaire** (partagé par des organisations ayant des préoccupations communes, comme des organismes publics) et **hybride** (deux ou plusieurs de ces modèles, reliés de sorte que les données et les applications puissent passer de l'un à l'autre). Le secteur a depuis ajouté les fonctions et les conteneurs en tant que service aux modèles de services, et le multicloud au vocabulaire du déploiement.

L'endroit où s'exécute un logiciel fait partie de ce que le chapitre 1 appelait la structure d'allocation d'une architecture (définir l'architecture) : la correspondance entre le logiciel et le matériel, les réseaux et les organisations. Il est significatif au sens du chapitre 1 parce qu'il coûte cher à défaire, et parce qu'il fixe une grande partie du coût d'un système, sa disponibilité, sa frontière de sécurité et la vitesse à laquelle ses équipes peuvent livrer.

Modèles de services : qui gère quoi

Les modèles de services forment un continuum. À une extrémité, vous louez de la capacité brute et faites vous-même tout ce qui se trouve au-dessus ; à l'autre, vous louez une application finie et vous vous contentez de la configurer. Chaque pas vers la droite retire du travail d'exploitation et retire des choix. Le tableau ajoute les deux modèles postérieurs au document du NIST : les conteneurs en tant que service et les fonctions en tant que service.

Comparaison des modèles de services

Modèle	Vous déployez et exploitez	Le fournisseur exploite	Exemples	Usage typique
IaaS	Machines virtuelles : système d'exploitation, correctifs, environnement d'exécution, application, règles de mise à l'échelle	Matériel, réseau, hyperviseur, centre de données	Amazon EC2, Azure Virtual Machines, Google Compute Engine	Logiciels qui exigent un système d'exploitation ou un noyau particulier, applications existantes migrées telles quelles, contrôle total
CaaS (conteneurs en tant que service)	Images de conteneurs et leur configuration ; avec un Kubernetes managé, également le choix et la mise à niveau des nœuds de travail	Plan de contrôle des conteneurs, souvent aussi les nœuds ; avec les conteneurs serverless, tous les serveurs	Amazon EKS et ECS, Azure Kubernetes Service et Container Apps, Google Kubernetes Engine et Cloud Run	Nombreux services construits sous forme de conteneurs, équipes qui veulent un empaquetage portable
PaaS	Code applicatif et sa configuration	Système d'exploitation, environnement d'exécution, mise à l'échelle, équilibrage de charge, souvent la journalisation	Heroku, Azure App Service, Google App Engine, AWS Elastic Beanstalk	Applications web et API standard qui respectent les conventions de la plateforme
FaaS	Fonctions individuelles et événements qui les déclenchent	Tout le reste, y compris la réduction à zéro ; facturation à l'invocation et à la durée	AWS Lambda, Azure Functions, Google Cloud Run functions, Cloudflare Workers	Traitements pilotés par les événements, en pics ou à faible volume, en étapes courtes
SaaS	Configuration, utilisateurs, autorisations et vos données	L'application entière	Microsoft 365, Salesforce, Workday, Zendesk	Capacités génériques qui ne différencient pas l'entreprise

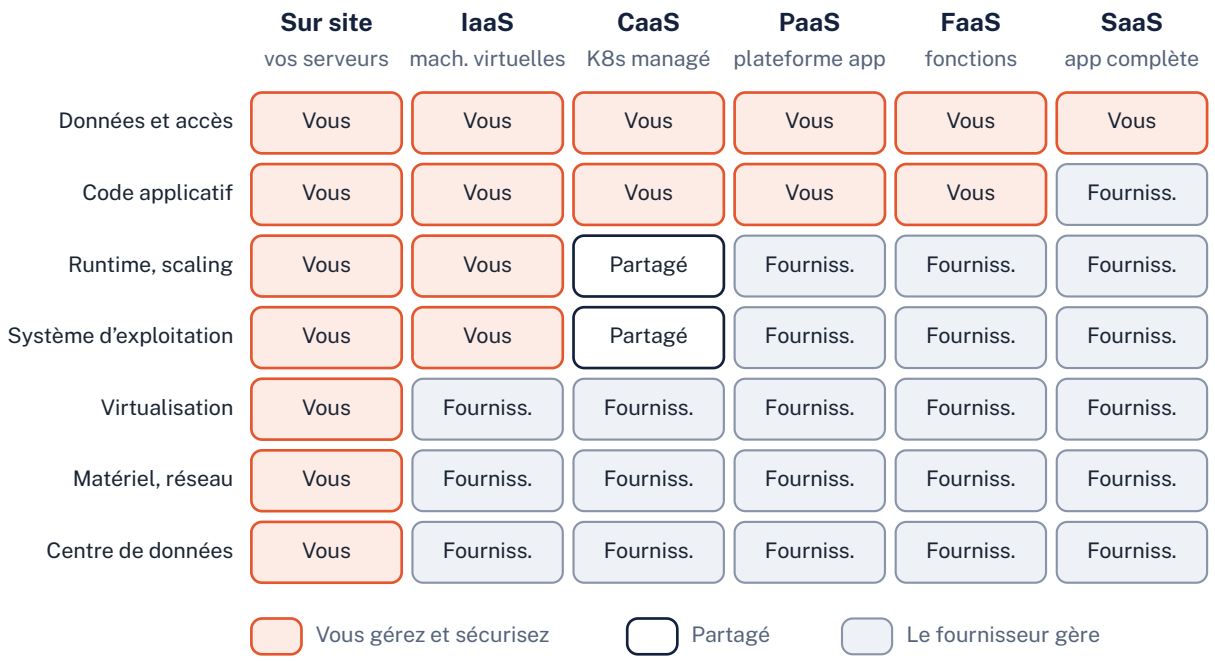
En pratique, les frontières s'estompent. Les plateformes de conteneurs serverless comme Google Cloud Run et Azure Container Apps exécutent une image de conteneur comme un PaaS exécute du code, en descendant à zéro comme une fonction. Un Kubernetes managé peut aller d'un plan de contrôle hébergé, où vous continuez à appliquer les correctifs et les mises à niveau aux machines de travail, jusqu'à un mode entièrement managé où le fournisseur exploite aussi les nœuds. La question utile n'est jamais de savoir quelle étiquette s'applique, mais quelles tâches concrètes restent les vôtres : correctifs, décisions de mise à l'échelle, planification de la capacité, sauvegardes, basculement, mises à niveau et astreinte.

Une seconde distinction utile est l'unité de déploiement et de facturation. L'IaaS facture les machines à la seconde ou à l'heure, qu'elles soient occupées ou non ; le FaaS facture les invocations et le temps d'exécution et ne coûte rien au repos ; le PaaS et les conteneurs serverless se situent entre les deux. Le chapitre 6 explique pourquoi la tarification à la requête est moins chère qu'une capacité toujours active en dessous d'un volume de bascule, et plus chère au-dessus (architectures serverless). C'est la forme de la charge, et non sa taille, qui décide de quel côté du point de bascule elle se trouve.

Le modèle de responsabilité partagée

Chaque grand fournisseur publie un modèle de responsabilité partagée. AWS le résume ainsi : le fournisseur est responsable de la sécurité *du* cloud, et le client de la sécurité *dans* le cloud. Le fournisseur sécurise les bâtiments, le matériel, le réseau, l'hyperviseur et les services qu'il exploite ; le client sécurise ce qu'il construit et configure par-dessus. L'emplacement de la frontière dépend du modèle de services, comme le montre le diagramme.

Qui est responsable de quoi, selon le modèle de services



Aller vers la droite confie les couches basses au fournisseur, mais les données, les identités et la configuration des accès restent au client dans tous les modèles.

Trois choses ne se transfèrent jamais. Le client détient toujours ses données (ce qui est stocké, où, pendant combien de temps, qui peut le lire), ses identités et ses autorisations, et la configuration des services qu'il utilise. De nombreuses violations dans le cloud ont résulté d'erreurs de configuration du client plutôt que de défaillances du fournisseur : stockage objet laissé lisible par tous, clés d'administrateur commitées dans des dépôts publics, autorisations trop larges. En 2019, par exemple, un attaquant a exploité un pare-feu applicatif web mal configuré chez Capital One pour obtenir des identifiants cloud et copier des données concernant plus de 100 millions de personnes aux États-Unis et au Canada ; l'infrastructure du fournisseur s'est comportée comme prévu.

Pour un architecte, le modèle est une liste de contrôle des tâches à attribuer à quelqu'un. Si une équipe déplace la base de données des commandes de la plateforme de livraison de repas d'une machine virtuelle vers un service de base de données managé, l'application des correctifs au moteur de base de données passe au fournisseur, mais pas le choix d'une fenêtre de maintenance, le test des mises à niveau de version majeure, l'octroi d'accès selon le moindre privilège, l'activation du chiffrement ni la vérification que les sauvegardes peuvent être restaurées. Le chapitre 8 traite en profondeur des principes de sécurité qui sous-tendent ces responsabilités (principes de sécurité pour les architectes).

Les modèles de déploiement et leurs vraies raisons

Les organisations choisissent rarement un modèle de déploiement pour une seule raison, et la raison affichée n'est pas toujours la vraie. Il est utile de distinguer les motivations qui résistent à l'examen de celles qui n'y résistent pas.

Public, privé, hybride et multicloud

1. **Cloud public**
   Choisi pour l'élasticité, la rapidité (un nouvel environnement en quelques minutes, et non un achat de matériel en plusieurs mois), un vaste catalogue de services managés et une présence mondiale. Il convient à une charge variable, aux nouveaux produits dont la demande est inconnue et aux équipes qui veulent consacrer leurs efforts au produit plutôt qu'aux serveurs.

2. **Cloud privé**
   Choisi pour un contrôle réglementaire ou contractuel, une charge régulière et prévisible, des investissements existants dans un centre de données ou une latence très faible vers des équipements sur site. Un cloud privé désigne un provisionnement en libre-service, piloté par API, sur une infrastructure dédiée (par exemple avec OpenStack, VMware ou l'édition sur site du cloud public d'un fournisseur), et non une simple salle de serveurs virtualisée où les demandes passent encore par une file de tickets.

3. **Cloud hybride**
   Résulte généralement d'une migration progressive, de données qui doivent rester sur site (un grand livre sur mainframe, des systèmes d'usine ou d'hôpital) ou de sites en périphérie comme des magasins et des entrepôts. Le débordement d'un centre de données privé vers le cloud public aux heures de pointe est souvent promis mais rarement praticable, car les données dont la charge de travail a besoin restent derrière et les copier coûte du temps et de l'argent.

4. **Multicloud**
   Utiliser deux fournisseurs publics ou plus. Parmi les raisons défendables : utiliser un service distinctif de chaque fournisseur, des acquisitions qui arrivent sur un autre cloud, un levier de négociation et une réglementation qui exige des plans de sortie crédibles. Dans l'UE, le règlement sur la résilience opérationnelle numérique (DORA), applicable depuis janvier 2025, impose aux entités financières de disposer de stratégies de sortie pour leurs prestataires de services TIC critiques. Rendre chaque charge de travail portable entre plusieurs clouds est un objectif différent et bien plus coûteux.

L'erreur courante consiste à adopter le multicloud comme une couverture abstraite. Faire tourner une charge de travail à l'identique chez deux fournisseurs implique de n'utiliser que les services que proposent les deux, de construire et tester deux jeux de code d'infrastructure, d'entretenir les compétences pour les deux, et souvent de payer des frais de transfert de données entre eux. C'est le piège du plus petit dénominateur commun : vous payez chaque jour le coût de la portabilité en échange d'une sortie dont vous ne vous servirez peut-être jamais. La section 10.8 traite la dépendance au fournisseur comme un compromis à chiffrer, et non comme un péché à éviter à tout prix (FinOps, architecture des coûts et dépendance au fournisseur).

Les régions et les zones de disponibilité comme données de conception

Une région est une zone géographique dotée de son propre ensemble indépendant de centres de données ; une zone de disponibilité est un ou plusieurs centres de données d'une région, avec une alimentation électrique, un refroidissement et un réseau distincts, suffisamment éloignés des autres pour qu'un incendie, une inondation ou une panne de courant locale n'en touche qu'un seul, et suffisamment proches pour des allers-retours de quelques millisecondes au plus. Le chapitre 0 a expliqué l'idée. Pour la conception, trois points supplémentaires comptent.

Premièrement, chaque service managé a un périmètre de défaillance : les ressources *zonales* (une machine virtuelle isolée, un disque) disparaissent avec leur zone ; les services *régionaux* (la plupart des bases de données managées en mode multizone, le stockage objet, les files d'attente) sont répliqués entre zones par le fournisseur ; quelques services *mondiaux* (DNS, identité, certains services de diffusion de contenu et d'équilibrage de charge) couvrent plusieurs régions, ce qui rend leurs rares pannes très étendues. Connaître le périmètre de chaque dépendance est la première étape de toute analyse de disponibilité (voir disponibilité et fiabilité).

Deuxièmement, la répartition entre zones est le choix par défaut, peu coûteux et très rentable. Faire tourner les services sans état de la plateforme de livraison de repas dans trois zones derrière un équilibreur de charge, avec une base de données qui garde une instance de secours synchrone dans une deuxième zone, permet de survivre à la perte d'un centre de données avec un basculement automatique et un léger coût en latence. Trois zones plutôt que deux comptent pour tout ce qui utilise des quorums, comme etcd dans Kubernetes ou une base de données fondée sur un consensus, car une majorité doit survivre. Le trafic entre zones n'est pas toujours gratuit pour autant : la plupart des grands fournisseurs facturent les données qui circulent entre zones, si bien que des services bavards répartis entre zones ajoutent une ligne visible à la facture.

Troisièmement, les régions ne sont pas identiques. Les régions plus récentes ou plus petites peuvent manquer de certains services, certaines ont moins de zones, les prix varient d'une région à l'autre, et la juridiction applicable suit la région. La section 10.6 traite des régions multiples, de la reprise après sinistre et de la résidence des données (multirégion, reprise après sinistre et edge computing).

SI VOUS DÉBUTEZ

Pas à pas : ce qui s'arrête quand une zone tombe en panne

L'idée la plus difficile de cette section est le périmètre de défaillance : chaque partie d'un système vit dans une zone, à l'échelle d'une région ou à l'échelle du monde, et cela détermine ce qui s'arrête quand quelque chose tombe en panne. Suivez la panne d'une zone à travers la plateforme de livraison de repas.

- La plateforme fait tourner ses services sans état dans trois zones de disponibilité derrière un équilibreur de charge, avec une base de données des commandes managée qui garde une instance de secours synchrone dans une deuxième zone. Les clients la trouvent grâce au DNS.
- Chaque élément a un périmètre. Une machine virtuelle isolée est zonale : elle vit dans une seule zone. La base de données multizone est régionale : le fournisseur la garde copiée entre plusieurs zones. Le DNS est mondial : il couvre plusieurs régions.
- Une inondation met une zone hors ligne pendant le coup de feu du soir. Les copies des services situées dans cette zone s'arrêtent avec elle. L'équilibreur de charge cesse de leur envoyer des requêtes, et les copies des deux autres zones continuent, en se multipliant si nécessaire.
- Si la copie principale de la base de données se trouvait dans la zone inondée, le fournisseur bascule sur l'instance de secours après une courte pause : c'est un failover (basculement) automatique. Comme l'instance de secours est synchrone, chaque écriture lui est parvenue avant que la base de données ne la confirme, comme l'a expliqué le chapitre 7 ; les commandes confirmées survivent donc. Les commandes prises en cours de route pendant la pause peuvent échouer et devoir être relancées.
- Les services mondiaux tombent en panne différemment. Leurs pannes sont rares, mais la répartition entre zones, ou même entre régions, ne vous en protège pas, car toutes les zones et toutes les régions dépendent du même service.
- **La leçon :** connaissez le périmètre de chaque dépendance avant de faire confiance à la conception. Les éléments zonaux ont besoin de copies dans plus d'une zone ; les éléments régionaux survivent à la perte d'une zone mais pas à celle de toute leur région ; les éléments mondiaux tombent rarement en panne, mais partout à la fois.

Services managés ou autogérés

Un service managé est un composant (base de données, file d'attente, cache, moteur de recherche, plan de contrôle Kubernetes) que le fournisseur installe, exploite, corrige, sauvegarde et met à l'échelle pour vous. L'alternative consiste à exploiter vous-même le même logiciel open source sur des machines virtuelles ou dans des conteneurs. Le compromis oppose la charge de travail et le risque d'un côté, au contrôle, au coût et à la portabilité de l'autre.

Les services managés l'emportent quand le composant n'est pas différenciant : toute entreprise de livraison de repas a besoin d'une base de données PostgreSQL fiable, mais aucune ne gagne de clients grâce à sa façon d'appliquer les correctifs à sa base de données. En termes de conception pilotée par le domaine, ce sont des sous-domaines génériques, ceux qu'il vaut mieux acheter que construire. Une base de données managée offre un basculement multizone, une restauration à un instant donné et l'application des correctifs de versions mineures, qui exigeraient sinon une équipe compétente d'astreinte. L'autogestion l'emporte quand vous avez besoin d'une version, d'une extension ou d'un paramètre de réglage que le fournisseur ne propose pas, quand les limites ou quotas du fournisseur ne conviennent pas, quand le surcoût du service managé devient important à grande échelle, ou quand la portabilité entre environnements compte plus que la commodité.

Managé ne veut pas dire sans maintenance. Le fournisseur imposera des mises à niveau de version majeure selon son propre calendrier quand les anciennes versions arriveront en fin de support, modifiera des valeurs par défaut, imposera des quotas que vous découvrirez au pire moment et subira des pannes que vous ne pourrez pas réparer, seulement attendre qu'elles passent. Vous restez responsable de la conception du schéma, des performances des requêtes, des choix de capacité, du coût et du test régulier qui prouve qu'une restauration fonctionne. Traitez chaque service managé comme une dépendance qui a ses propres modes de défaillance et son propre coût de sortie.

Les fondations du cloud : landing zones et hiérarchie de comptes

Avant l'arrivée de la première charge de travail, un parc cloud a besoin de fondations, et elles sont bien plus faciles à poser tôt qu'à ajouter après coup. Une **landing zone** est l'environnement préparé dans lequel les charges de travail se déploient. AWS la décrit comme un environnement multicompte, bien architecturé, scalable et sécurisé. La landing zone Azure de Microsoft sépare une *landing zone de plateforme*, le socle central de gouvernance, de sécurité et de ressources partagées, des *landing zones de charge de travail*, où les équipes produit déploient à l'intérieur de ces garde-fous. Dans les deux cas, elle doit être construite et modifiée sous forme de code (section 10.4), et c'est souvent la première chose dont une équipe plateforme est propriétaire (section 10.7).

La décision la plus difficile à changer plus tard est la hiérarchie. Chaque fournisseur a une unité d'isolation de base et une arborescence qui regroupe ces unités : les comptes AWS dans des unités organisationnelles ; les abonnements Azure sous des groupes d'administration ; les projets Google Cloud, éventuellement dans des dossiers, sous une organisation. L'arborescence compte parce que les politiques s'y rattachent. Les politiques de Google Cloud se transmettent par héritage vers le bas de l'arborescence, et les abonnements Azure héritent des politiques définies sur leurs groupes d'administration. Les service control policies d'AWS fixent les permissions maximales disponibles dans les comptes membres ; elles n'accordent rien par elles-mêmes, et elles n'affectent pas le compte de gestion, ce qui est une raison de garder les charges de travail hors de celui-ci.

Un compte ou un abonnement constitue une frontière bien plus solide qu'une étiquette ou une convention de nommage. Sur AWS, un compte n'a par défaut aucun accès à un autre, les quotas s'appliquent par compte et les coûts s'y rattachent directement. Séparez donc par charge de travail et par environnement dès le départ, car déplacer ensuite un système en production d'un compte à l'autre est un véritable projet de migration. AWS facture les ressources utilisées, et non le nombre de comptes, même si de nombreux comptes exigent bel et bien de l'automatisation.

Pour la plateforme de livraison de repas, une arborescence de départ raisonnable, adaptée de la structure type d'AWS, comporte quatre branches. *Sécurité* contient un compte d'archivage des journaux que les équipes produit ne peuvent pas modifier, ainsi qu'un compte d'audit. *Infrastructure* contient le réseau et les services partagés. *Charges de travail* contient des comptes de production et de hors-production distincts pour les commandes, l'affectation et les paiements, avec des politiques plus strictes sur la branche des paiements. *Sandbox* sert aux expériences sans risque. Les garde-fous placés au sommet de l'arborescence peuvent interdire les régions non approuvées et empêcher quiconque de désactiver la journalisation.

La liste des domaines de conception d'Azure fait une liste de contrôle utile, même sur d'autres clouds : facturation et locataires, identité, organisation des ressources, réseau, sécurité, gestion, gouvernance et automatisation. Vous n'avez pas besoin de tout construire d'un coup ; AWS note que vous pouvez commencer avec une partie de ses unités types et en ajouter d'autres plus tard. L'échec habituel est l'inverse d'un manque de structure : des garde-fous si serrés, et de nouveaux comptes si longs à obtenir, que les équipes les contournent. Proposez les nouveaux comptes en libre-service via la plateforme, avec les garde-fous déjà en place.

Les fondations réseau : VPC, connectivité privée et sortie de données (egress)

Un **cloud privé virtuel** (VPC ; un réseau virtuel sur Azure) est votre réseau privé chez un fournisseur. Planifiez d'abord ses adresses. Sur AWS, la plage avec laquelle un VPC est créé ne peut être ni redimensionnée ni supprimée, seulement complétée, et deux VPC dont les plages se chevauchent ne peuvent pas être appairés. Azure donne le même conseil pour ses réseaux et pour les sites sur site. Attribuez donc à chaque réseau une plage sans chevauchement, issue d'un plan central unique, avec de la marge pour grandir.

Les sous-réseaux découpent un VPC et, sur AWS, chacun se trouve dans une seule zone de disponibilité : la résilience impose donc de les répéter par zone. Ce qui rend un sous-réseau public ou privé, c'est son routage : un sous-réseau public a une route vers une passerelle internet, un sous-réseau privé n'en a pas et a besoin d'un dispositif NAT pour joindre l'extérieur, et un sous-réseau isolé n'a aucune route au-delà du VPC. La plateforme de livraison de repas peut exécuter trois niveaux sur trois zones : les équilibreurs de charge dans des sous-réseaux publics, les services dans des sous-réseaux privés et les bases de données dans des sous-réseaux isolés. Les sous-réseaux ne sont que des zones grossières ; un contrôle plus fin vient des méthodes d'identité et de segmentation de la section 8.5.

Vient ensuite la connectivité. L'appairage (peering) relie deux réseaux, mais il n'est pas transitif : si A est appairé à B et à C, B ne peut toujours pas joindre C en passant par A. À mesure que les réseaux se multiplient, les équipes passent à un hub, tel qu'AWS Transit Gateway ou la conception hub-spoke d'Azure, où un réseau central héberge les passerelles, le pare-feu et le DNS partagés. Le hub devient alors une dépendance partagée, et Microsoft en recommande donc un par région. Pour joindre un centre de données, un VPN site à site fait passer des tunnels chiffrés par internet, tandis qu'AWS Direct Connect ou Azure ExpressRoute offre une liaison dédiée ; AWS note que Direct Connect n'est pas chiffré par défaut. Pour joindre un service plutôt qu'un réseau entier, utilisez AWS PrivateLink, Azure Private Link ou Google Cloud Private Service Connect, qui l'exposent à une adresse privée sans appairage ni accès à internet.

Enfin, contrôlez ce qui sort. De nombreux parcs filtrent soigneusement le trafic entrant et laissent tout sortir, alors que l'accès sortant est le chemin par lequel partent les données volées, y compris au moyen d'instructions d'IA injectées (section 12.6). Une passerelle NAT permet aux services privés d'ouvrir des connexions sortantes et empêche l'extérieur d'en ouvrir vers eux, mais elle ne fait que traduire les adresses et ne choisit pas les destinations. Pour les choisir, il faut un pare-feu de sortie, comme AWS Network Firewall ou Azure Firewall avec des règles de domaine, placé de façon que tout le trafic sortant le traverse. Le service de commandes de la plateforme ne joint alors que ses fournisseurs de paiement, de cartographie et de messagerie, et tout le reste est refusé et journalisé.

Les chemins de sortie ont aussi leurs aspects de fiabilité et de coût. Une passerelle NAT AWS standard réside dans une seule zone : si des charges de travail d'autres zones la partagent, elles perdent l'accès à internet quand cette zone tombe en panne ; AWS conseille d'en avoir une par zone et a ajouté une option régionale, vérifiez donc la version actuelle. Les passerelles NAT sont facturées à l'heure et au gigaoctet, tandis que les points de terminaison de passerelle AWS pour S3 n'entraînent aucuns frais horaires ni frais de traitement des données : y faire passer le trafic de stockage est donc une économie facile (section 10.8).

Ce que signifie cloud native

La Cloud Native Computing Foundation (CNCF), qui fait partie de la Linux Foundation et héberge Kubernetes, publie la définition la plus citée du cloud native. La version 1.0 (2018) décrivait des technologies permettant aux organisations de construire et d'exploiter des applications scalables dans des environnements dynamiques comme les clouds publics, privés et hybrides, et citait comme exemples typiques les conteneurs, les service meshes, les microservices, l'infrastructure immuable et les API déclaratives. La version 1.1, approuvée en 2024, déplace l'accent des technologies vers les pratiques : développer, construire et déployer des charges de travail de façon programmatique et reproductible à grande échelle, ce qui se caractérise par «des systèmes faiblement couplés qui interopèrent de manière sécurisée, résiliente, gérable, durable et observable».

Deux conséquences en découlent. Le cloud native décrit *comment* un système est construit et exploité, et non *où* : un cluster Kubernetes dans un centre de données privé, provisionné à partir de code et déployé de nombreuses fois par jour, est cloud native ; une machine virtuelle configurée à la main et copiée telle quelle dans un cloud public ne l'est pas. Et le cloud native est un moyen plutôt qu'une fin. Ses pratiques (automatisation, composants jetables, configuration déclarative, élasticité, observabilité) sont rentables quand un système a besoin de changements fréquents, d'une échelle variable et d'une haute disponibilité ; elles ajoutent du coût et de la complexité pour un outil interne stable utilisé par vingt personnes.

Les migrations sont souvent décrites à l'aide des « sept R » (voir la section 11.6), qu'il vaut la peine de connaître parce qu'ils fixent les attentes : *rehost* (« lift and shift », sans changement), *relocate* (déplacement de machines virtuelles ou de plateformes entières vers le cloud sans les modifier), *replatform* (petits changements comme le passage à une base de données managée), *refactor* ou réarchitecture (refonte pour un fonctionnement cloud native), *repurchase* (remplacement par un SaaS), *retain* (laisser en place) et *retire* (arrêter le système). Le rehosting est rapide, mais il apporte généralement peu des avantages du cloud et, exécuté sur des machines toujours actives dimensionnées pour le pic, peut coûter plus cher que le centre de données qu'il a quitté.

Quand le cloud n’est pas moins cher

Le contre-exemple le plus connu est 37signals, l’entreprise derrière Basecamp et HEY. En octobre 2022, son directeur technique, David Heinemeier Hansson, a annoncé qu’elle quittait le cloud, en citant une facture AWS annuelle de plus de 3 millions USD pour des charges de travail à la charge régulière et prévisible. Elle a acheté pour environ 700 000 USD de serveurs Dell, les a installés dans un espace de colocation loué et a retiré ses applications du calcul AWS au cours de 2023 ; elle a indiqué que cela réduisait sa facture cloud d’environ 2 millions USD par an. En 2025, elle a annoncé la seconde étape : déplacer environ 18 pétaoctets de fichiers d’Amazon S3 vers des baies de stockage d’environ 1,5 million USD, pour remplacer une facture S3 d’environ 1,5 million USD par an, avec l’objectif déclaré de fermer son compte AWS. L’entreprise a projeté des économies d’environ 10 millions USD sur cinq ans. Sortir ainsi des charges de travail du cloud s’appelle le rapatriement depuis le cloud (cloud repatriation).

Les deux camps méritent d’être entendus. En faveur du raisonnement de 37signals : sa charge est régulière, elle gagnait donc peu à l’élasticité ; elle employait déjà des ingénieurs d’exploitation expérimentés et a indiqué n’avoir eu besoin d’aucun personnel supplémentaire ; les fournisseurs de colocation s’occupent de l’alimentation, du refroidissement et de la sécurité physique ; et les serveurs peuvent servir cinq ans ou plus, si bien que leur coût par année de calcul est très inférieur à la location à la demande. De l’autre côté : 37signals est une entreprise mature à la demande prévisible, ce que ne sont pas la plupart des nouveaux produits ; la comparaison dépend fortement de ce qu’aurait été la facture cloud avec des remises d’engagement et un dimensionnement ajusté ; posséder du matériel implique des dépenses d’investissement, des délais de livraison de plusieurs semaines ou mois et une capacité achetée à l’avance ; et une équipe plus petite, sans compétences d’exploitation poussées, devrait les recruter, ce qui peut effacer l’économie. Les critiques soulignent aussi que les chiffres d’économies proviennent de l’entreprise elle-même.

EXEMPLE COMMENTÉ

Exemple commenté : louer ou posséder la capacité de calcul de la plateforme de livraison de repas

Tous les prix sont des hypothèses illustratives choisies pour montrer la méthode, et non de vrais devis. La capacité cloud à la demande coûte £0,040 par vCPU-heure ; un engagement sur trois ans coûte £0,024 (une remise de 40 %) ; le matériel en propre en colocation coûte £0,012 par vCPU-heure, alimentation et espace compris, mais exige 30 % de marge de capacité et du temps de personnel. La plateforme a besoin de 40 vCPU toute la journée, et de 120 vCPU pendant les quatre heures du coup de feu du soir (environ 122 heures par mois). Un mois compte 730 heures.

A : cloud, dimensionné pour le pic, toujours actif	120 vCPU × 730 h × £0,040 = **£3504 par mois**
B : cloud, base engagée plus pic en autoscaling	40 × 730 × £0,024 = £701 ; plus 80 × 122 × £0,040 = £390 ; total **£1091 par mois**
C : matériel en propre dimensionné pour le pic	120 × 1,3 × 730 × £0,012 = £1367 ; plus un quart d'ingénieur à £80 000 par an = £1667 ; total **£3034 par mois**
Passons à plus grande échelle : 2000 vCPU en continu, sans heure de pointe	Cloud avec engagement : 2000 × 730 × £0,024 = £35 040. En propre : 2000 × 1,3 × 730 × £0,012 = £22 776 plus un ingénieur (£6667) = **£29 443**, soit environ 16 % de moins

Pour une petite charge en pics, l'élasticité domine : l'option B coûte moins d'un tiers de l'option A et environ un tiers de l'option C, car elle ne paie la capacité de pointe que pendant le coup de feu.

Pour une charge importante et plate, les coûts fixes de la propriété se répartissent sur davantage de capacité et la propriété commence à l'emporter : c'est la situation de 37signals. L'économie est réelle, mais plus faible que ne le suggèrent les comparaisons spectaculaires, une fois comptés la marge de capacité, le personnel et l'effort de migration.

La méthode compte plus que les chiffres : modélisez la forme de la charge, incluez le personnel et la marge de capacité, comparez avec le meilleur prix cloud (avec engagements) et non avec le prix catalogue, et refaites le calcul à mesure que l'activité évolue.

GUIDE DE DÉCISION

Guide de décision : choisir un modèle d'hébergement pour une charge de travail

Appliquez-le charge de travail par charge de travail, et non une fois pour toute l'organisation : une même entreprise utilise judicieusement un SaaS pour la paie, des fonctions pour le redimensionnement d'images et des conteneurs pour ses services cœurs.

Q1 **La capacité est-elle générique, avec un produit sur le marché qui convient suffisamment (e-mail, paie, CRM, paiements) ?**

- **Oui** Achetez un SaaS et intégrez-le via son API ; consacrez l'effort de conception à l'intégration et au plan de sortie.
- **Non** Continuez : vous devez exploiter votre propre logiciel.

Q2 **La loi, un contrat ou l'emplacement physique excluent-ils le cloud public pour cette charge de travail (règles de souveraineté strictes, équipements sur site, boucles de contrôle à la milliseconde) ?**

- **Oui** Utilisez un cloud privé, la colocation ou un déploiement en périphérie sur site, relié aux services du cloud public en mode hybride lorsque c'est permis.
- **Non** Continuez avec les options de cloud public.

Q3 **Le traitement est-il piloté par les événements, de courte durée et en pics ou à faible volume (webhooks, traitement de fichiers, tâches planifiées) ?**

- **Oui** Utilisez des fonctions en tant que service ou une plateforme de conteneurs serverless ; vérifiez les limites de démarrage à froid, de durée et de coût à grand volume.
- **Non** Continuez : il s'agit d'un service de longue durée.

Q4 **Le modèle d'un PaaS ou d'une plateforme de conteneurs serverless convient-il (service HTTP sans état, environnement d'exécution standard, peu de besoins particuliers) ?**

- **Oui** Utilisez-le. C'est ce qui retire le plus de travail d'exploitation à une petite équipe.
- **Non** Continuez : vous avez besoin de plus de contrôle.

Q5 **Exploitez-vous de nombreux services répartis entre plusieurs équipes, et avez-vous (ou financerez-vous) une équipe capable d'exploiter Kubernetes ?**

- **Oui** Utilisez un Kubernetes managé (CaaS), idéalement derrière une plateforme interne (section 10.7).
- **Non** Utilisez des machines virtuelles avec des bases de données managées et des groupes d'autoscaling, construits à partir de code d'infrastructure.

Q6 **La charge est-elle importante, régulière et prévisible pour des années, avec des compétences d'exploitation en interne ?**

Oui Comparez le coût de matériel en propre ou en colocation à celui d'une capacité cloud avec engagement ; le rapatriement peut être rentable.

Non Restez sur de la capacité louée et utilisez des remises d'engagement pour la base régulière.

ATTENTION

Erreurs courantes dans le choix des modèles cloud

- Migrer tout tel quel sur des machines virtuelles toujours actives dimensionnées pour le pic, puis en conclure que le cloud coûte cher.
- Adopter le multicloud comme une couverture générale, en payant le coût de la portabilité sur chaque charge de travail au lieu de prévoir une sortie pour les quelques-unes qui en ont besoin.
- Supposer que le fournisseur sauvegarde, chiffre et restreint l'accès à vos données par défaut ; dans tous les modèles, ce sont vos décisions.
- Choisir Kubernetes avant d'avoir une équipe pour l'exploiter ou assez de services pour le justifier.
- Ignorer les frais de transfert de données, en particulier entre zones, régions et fournisseurs, lorsqu'on dessine le diagramme de déploiement.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Une équipe déplace sa base de données d'une VM qu'elle exploitait elle-même vers un service de base de données managé. Citez deux responsabilités qui passent au fournisseur et deux qui restent à l'équipe.

Le fournisseur prend en charge les correctifs du moteur, le matériel et généralement les sauvegardes automatisées et le basculement entre zones. L'équipe garde la conception du schéma et des requêtes, les autorisations d'accès, les choix de chiffrement, la capacité et le coût, le test des restaurations et la planification des mises à niveau de version majeure.

▼ Pourquoi un système fondé sur un quorum, comme etcd, préfère-t-il trois zones de disponibilité à deux ?

Un quorum a besoin d'une majorité. Avec deux zones, perdre la zone qui détient la majorité arrête le système ; avec trois, n'importe quelle zone peut tomber en panne et une majorité survit.

▼ **Une machine virtuelle construite à la main et copiée telle quelle dans un cloud public est-elle cloud native ?**

Non. Le cloud native désigne des pratiques comme l'automatisation, la configuration déclarative, les composants jetables et l'observabilité, et non un emplacement. Le rehosting déplace le système sans adopter ces pratiques.

▼ **Pourquoi une même organisation peut-elle judicieusement utiliser à la fois un SaaS, des fonctions et Kubernetes ?**

Le modèle d'hébergement doit se choisir charge de travail par charge de travail : les capacités génériques s'achètent en SaaS, les tâches en pics pilotées par les événements conviennent aux fonctions, et les services cœurs de longue durée peuvent justifier des conteneurs sur Kubernetes.

▼ **À vous : Prix indicatifs : à la demande £0,05 par vCPU-heure, avec engagement £0,03. Il vous faut 20 vCPU en permanence, plus 60 autres cinq heures par jour, sur un mois de 30 jours. Comparez tout à la demande, la base engagée et l'engagement de tout pour le pic.**

Base : 20 x 720 = 14 400 vCPU-heures ; supplément de pointe : 60 x 150 = 9000. Tout à la demande : 23 400 x £0,05 = £1170. Engagement sur la base : 14 400 x £0,03 + 9000 x £0,05 = £882. Engagement sur 80 vCPU tout le mois : 80 x 720 x £0,03 = £1728. Ne vous engagez que sur la base certaine et laissez le mode à la demande couvrir le coup de feu. Du matériel en propre n'aurait de sens que pour une charge importante et stable.

SOURCES

- NIST SP 800-145: The NIST Definition of Cloud Computing (2011): https://csrc.nist.gov/pubs/sp/800/145/final
- CNCF Technical Oversight Committee: Cloud Native Definition: https://github.com/cncf/toc/blob/main/DEFINITION.md
- AWS: Shared Responsibility Model: https://aws.amazon.com/compliance/shared-responsibility-model/
- Microsoft Learn: Shared responsibility in the cloud: https://learn.microsoft.com/en-us/azure/security/fundamentals/shared-responsibility
- The Register (2025): 37signals is completing its on-prem move: https://www.theregister.com/2025/05/09/37signals_cloud_repatriation_storage_savings
- Designing an AWS Control Tower landing zone (AWS Prescriptive Guidance): https://docs.aws.amazon.com/prescriptive-guidance/latest/designing-control-tower-landing-zone/introduction.html

- Sample account structure and OUs (AWS Prescriptive Guidance): https://docs.aws.amazon.com/prescriptive-guidance/latest/designing-control-tower-landing-zone/account-structure.html
- What is an Azure landing zone? (Microsoft Cloud Adoption Framework): https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/ready/landing-zone/
- Azure landing zone design areas (Microsoft): https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/ready/landing-zone/design-areas
- Organizing your AWS environment using multiple accounts (AWS whitepaper): https://docs.aws.amazon.com/whitepapers/latest/organizing-your-aws-environment/organizing-your-aws-environment.html
- Service control policies (AWS Organizations): https://docs.aws.amazon.com/organizations/latest/userguide/orgs_manage_policies_scps.html
- Resource hierarchy (Google Cloud): https://docs.cloud.google.com/resource-manager/docs/cloud-platform-resource-hierarchy
- VPC CIDR blocks (AWS): https://docs.aws.amazon.com/vpc/latest/userguide/vpc-cidr-blocks.html
- Subnets for your VPC (AWS): https://docs.aws.amazon.com/vpc/latest/userguide/configure-subnets.html
- VPC peering basics (AWS): https://docs.aws.amazon.com/vpc/latest/peering/vpc-peering-basics.html
- What is AWS Transit Gateway?: https://docs.aws.amazon.com/vpc/latest/tgw/what-is-transit-gateway.html
- Hub-spoke network topology in Azure (Microsoft): https://learn.microsoft.com/en-us/azure/architecture/networking/architecture/hub-spoke
- AWS Direct Connect (Amazon VPC connectivity options whitepaper): https://docs.aws.amazon.com/whitepapers/latest/aws-vpc-connectivity-options/aws-direct-connect.html
- What is AWS PrivateLink?: https://docs.aws.amazon.com/vpc/latest/privatelink/what-is-privatelink.html
- Private Service Connect (Google Cloud): https://docs.cloud.google.com/vpc/docs/private-service-connect
- NAT gateways (AWS): https://docs.aws.amazon.com/vpc/latest/userguide/vpc-nat-gateway.html
- NAT gateway basics (AWS): https://docs.aws.amazon.com/vpc/latest/userguide/nat-gateway-basics.html
- Amazon VPC pricing: https://aws.amazon.com/vpc/pricing/
- What is AWS Network Firewall?: https://docs.aws.amazon.com/network-firewall/latest/developerguide/what-is-aws-network-firewall.html

SECTION 10.2 SUR 10.8 · AVANCÉ · 26 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Conteneurs et orchestration avec Kubernetes

AVANT DE LIRE

Cette section s'appuie sur :

- 0.7 Serveurs et cloud
- 9.1 Passage à l'échelle
- 9.2 Équilibrage de charge

Termes déjà vus qu'elle utilise : Conteneur, Autoscaling (mise à l'échelle automatique), Zone de disponibilité, PaaS (plateforme en tant que service)

L'IDÉE EN BREF

Pensez à un grand port. Les marchandises arrivent dans des conteneurs maritimes standard, si bien que chaque grue, chaque navire et chaque camion peut les manipuler sans savoir ce qu'ils contiennent. Le capitaine du port ne déplace pas les caisses lui-même ; il tient un tableau qui indique ce qui doit se trouver où, et son équipe compare sans cesse le quai au tableau et corrige toute différence : une caisse manquante, une grue en panne, un navire en retard. Les conteneurs logiciels sont les caisses standard, et Kubernetes est la capitainerie : vous écrivez ce qui doit tourner, et il vérifie et corrige sans relâche jusqu'à ce que la réalité corresponde.

CE QU'IL FAUT SAVOIR

- Une image de conteneur est un empilement de couches adressées par leur contenu, plus une configuration et un manifeste ; les tags peuvent changer, les digests non, donc la production doit exécuter des images épinglées par digest.
- Les spécifications OCI de runtime, d'image et de distribution permettent de stocker dans n'importe quel registre des images construites par n'importe quel outil et de les exécuter avec n'importe quel runtime conforme.
- Kubernetes comporte un plan de contrôle (serveur d'API, etcd, ordonnanceur, gestionnaire de contrôleurs) et des nœuds de travail (kubelet, runtime de conteneurs, kube-proxy) ; seul le serveur d'API communique avec etcd, et les nœuds tirent leur travail.
- Les contrôleurs exécutent des boucles de réconciliation déclenchées par niveau qui comparent l'état réel à l'état souhaité et agissent sur la différence, ce qui rend le système robuste face aux pannes et aux événements manqués.

- Les pods sont jetables ; les Deployments les déploient, les Services leur donnent des noms stables, Gateway API y fait entrer le trafic, les StatefulSets apportent identité et stockage, les Jobs s'exécutent jusqu'à leur terme et le HPA ajuste le nombre de répliques.
- Les requêtes pilotent la planification et l'autoscaling, les limites plafonnent la consommation (le CPU est bridé, un conteneur qui dépasse sa mémoire est tué) ; les sondes de vivacité ne doivent pas vérifier les dépendances partagées.
- Kubernetes est rentable avec de nombreux services, plusieurs équipes et une équipe plateforme ; pour quelques services sans état, un PaaS ou une plateforme de conteneurs serverless est généralement le meilleur choix.

Images, couches et registres

Le chapitre 0 a présenté les conteneurs comme des processus isolés qui partagent le noyau de l'hôte, empaquetés sous forme d'images construites à partir d'un Dockerfile (les serveurs et le cloud). Un architecte a besoin d'un peu plus de détails, car la conception des images influe sur la vitesse de build, sur la sécurité et sur la fiabilité avec laquelle le même logiciel atteint chaque environnement.

Une image de conteneur est un empilement de *couches* en lecture seule, chacune étant un ensemble compressé de modifications de fichiers, plus un petit document de configuration (la commande à exécuter, les variables d'environnement, l'utilisateur) et un *manifeste* qui les répertorie. Chaque couche et chaque manifeste sont identifiés par un *digest*, une empreinte SHA-256 de leur contenu. Les couches identiques ne sont stockées et téléchargées qu'une fois, si bien qu'une centaine d'images construites sur la même base partagent ses couches. Comme chaque instruction d'un Dockerfile produit une couche et que les outils de build réutilisent les couches en cache tant que rien ne change, on ordonne les instructions de la moins souvent modifiée à la plus souvent modifiée : image de base, puis dépendances, puis code applicatif. Les *builds multi-étapes* compilent dans une étape et ne copient que le résultat dans une petite image finale, en laissant derrière eux compilateurs et outils de build.

Les images résident dans un *registre* : Docker Hub, GitHub Container Registry, Amazon ECR, Azure Container Registry, Google Artifact Registry, ou un registre auto-hébergé comme Harbor. Un *tag* comme `orders:2.4.1` est une étiquette mobile ; toute personne disposant des droits de push peut la faire pointer vers une autre image dès demain. Un digest comme `orders@sha256:3f1c...` ne peut jamais changer de sens. Les déploiements en production doivent donc résoudre les tags en digests, pour que ce qui a été testé soit exactement ce qui s'exécute, et ne jamais utiliser `latest`.

Les petites images se téléchargent plus vite et comportent moins de composants à corriger. La pratique courante consiste à partir d'une base minimale (une distribution allégée ou une image distroless ne contenant que l'environnement d'exécution), à s'exécuter sous un utilisateur non root, à reconstruire régulièrement pour intégrer les correctifs de sécurité et à analyser les images dans le pipeline pour y détecter les vulnérabilités connues. La signature des images et l'ajout d'une nomenclature logicielle relèvent de la sécurité de la chaîne d'approvisionnement, traitée au chapitre 8 (sécurité de la chaîne d'approvisionnement logicielle).

Les normes OCI

Docker a popularisé les conteneurs à partir de 2013 et, en 2015, Docker et d'autres acteurs ont fondé l'Open Container Initiative (OCI) sous l'égide de la Linux Foundation pour éviter la fragmentation du format. L'OCI maintient trois spécifications. La *spécification de runtime* indique comment exécuter un conteneur décompressé (espaces de noms, groupes de contrôle, montages, processus à lancer) ; `runc` en est l'implémentation de référence. La *spécification d'image* définit les couches, les manifestes et la configuration. La *spécification de distribution* définit l'API HTTP que les registres exposent pour le push et le pull. La version 1.1 des spécifications d'image et de distribution (2024) a ajouté des façons standard de stocker d'autres artefacts dans les registres et de les lier à une image : c'est ainsi que les signatures et les nomenclatures logicielles voyagent désormais avec les images qu'elles décrivent.

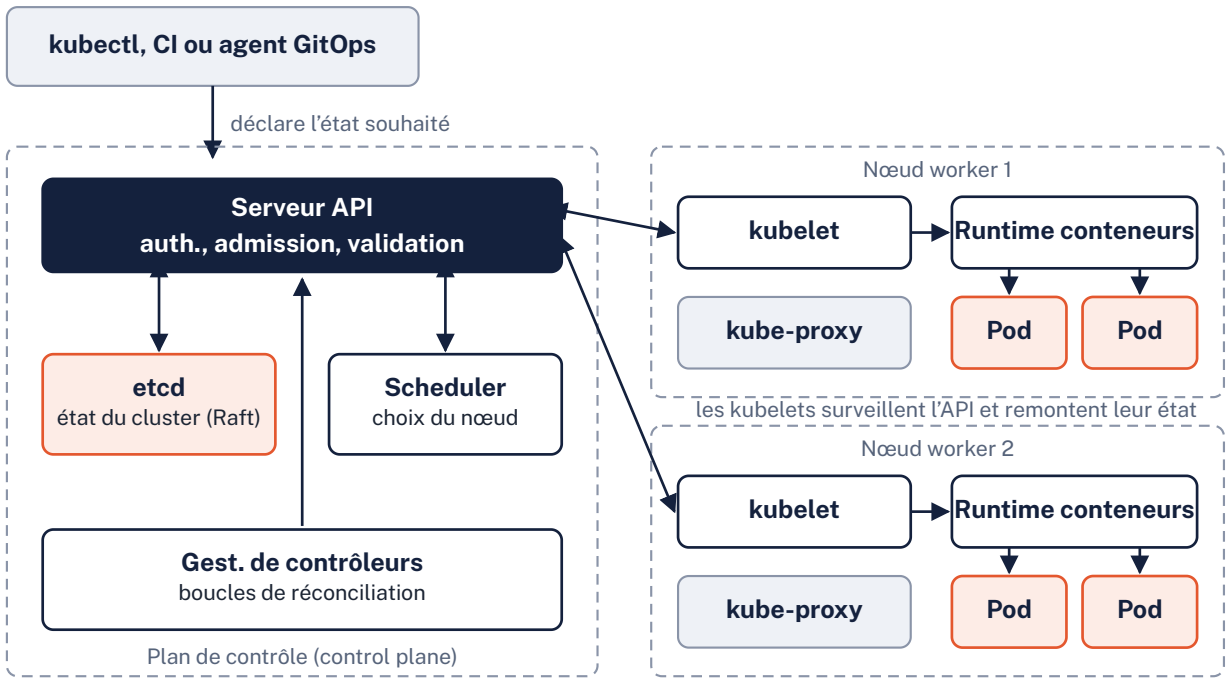
Le résultat pratique est l'interchangeabilité. Une image construite par Docker, Podman, Buildah ou un service de build cloud peut être stockée dans n'importe quel registre conforme et exécutée par n'importe quel runtime conforme : containerd et CRI-O sur un nœud Kubernetes, une plateforme de conteneurs serverless ou l'ordinateur portable d'un développeur. C'est pourquoi les conteneurs, contrairement à la plupart des formats d'empaquetage qui les ont précédés, sont devenus une langue commune entre le développement, l'exploitation et tous les grands clouds.

Pourquoi l'orchestration est nécessaire

Exécuter un conteneur tient en une seule commande. Faire tourner la plateforme de livraison de repas, non : quarante services, de trois à trente copies chacun, répartis sur des dizaines de machines dans trois zones de disponibilité, avec une nouvelle version de quelque chose mise en production toutes les heures. Il faut que quelqu'un décide sur quelle machine tourne chaque copie, redémarre les copies qui plantent, remplace les machines qui meurent, ajoute des copies pour le coup de feu du soir et les retire ensuite, déploie les nouvelles versions sans perdre de requêtes, permette aux services de se trouver mutuellement alors que les copies vont et viennent, leur fournisse configuration et secrets, et attache du stockage aux quelques-uns qui en ont besoin.

L'orchestration de conteneurs est le logiciel qui fait tout cela à partir d'une déclaration de ce qui doit tourner. Google a utilisé pendant plus de dix ans un système interne appelé Borg à cette fin ; en 2014, l'entreprise a annoncé Kubernetes, un système open source qui s'inspire de cette expérience, a publié la version 1.0 en juillet 2015 et l'a donné à la Cloud Native Computing Foundation, alors tout juste créée, où il est devenu en 2018 le premier projet à atteindre le statut « graduated ». Il existe des alternatives (HashiCorp Nomad, Amazon ECS, les plateformes de conteneurs serverless), mais Kubernetes est désormais le socle commun proposé par tous les grands fournisseurs et par la plupart des plateformes sur site.

Architecture de Kubernetes



Chaque composant communique avec le serveur d'API ; seul le serveur d'API communique avec etcd, et les nœuds tirent leur travail au lieu de recevoir des commandes.

Comment fonctionne Kubernetes : plan de contrôle et nœuds

Un cluster Kubernetes comporte un plan de contrôle (control plane), qui prend les décisions, et des nœuds de travail (worker nodes), qui exécutent les charges de travail. Le **serveur d'API** est la porte d'entrée : chaque client, du `kubectl` d'un développeur aux propres composants du cluster, lit et écrit des objets via son API REST ; il authentifie l'appelant, vérifie les autorisations et exécute des contrôles d'*admission* qui peuvent valider ou modifier chaque requête (par exemple, rejeter les conteneurs qui s'exécutent en root). C'est le seul composant qui communique avec **etcd**, un magasin clé-valeur fortement cohérent qui réplique l'état du cluster avec le protocole de consensus Raft sur, en général, trois ou cinq membres, afin qu'une majorité survive à la perte d'un ou deux d'entre eux (cohérence, CAP et PACELC).

L'**ordonnanceur (scheduler)** surveille l'apparition de nouveaux pods sans nœud attribué, écarte les nœuds qui ne peuvent pas les exécuter (pas assez de CPU ou de mémoire non réservés, mauvaise zone, matériel manquant, taints que le pod ne tolère pas), note les autres (en répartissant les copies entre zones, en les regroupant efficacement) et enregistre son choix via le serveur d'API. Le **gestionnaire de contrôleurs (controller manager)** exécute des dizaines de contrôleurs, chacun responsable d'un type d'objet : le contrôleur ReplicaSet maintient le bon nombre de pods, le contrôleur Node repère les nœuds qui cessent de se signaler, le contrôleur Job exécute des tâches jusqu'à leur terme. Dans un cloud, un gestionnaire de contrôleurs cloud crée aussi des équilibreurs de charge et attache des disques via l'API du fournisseur.

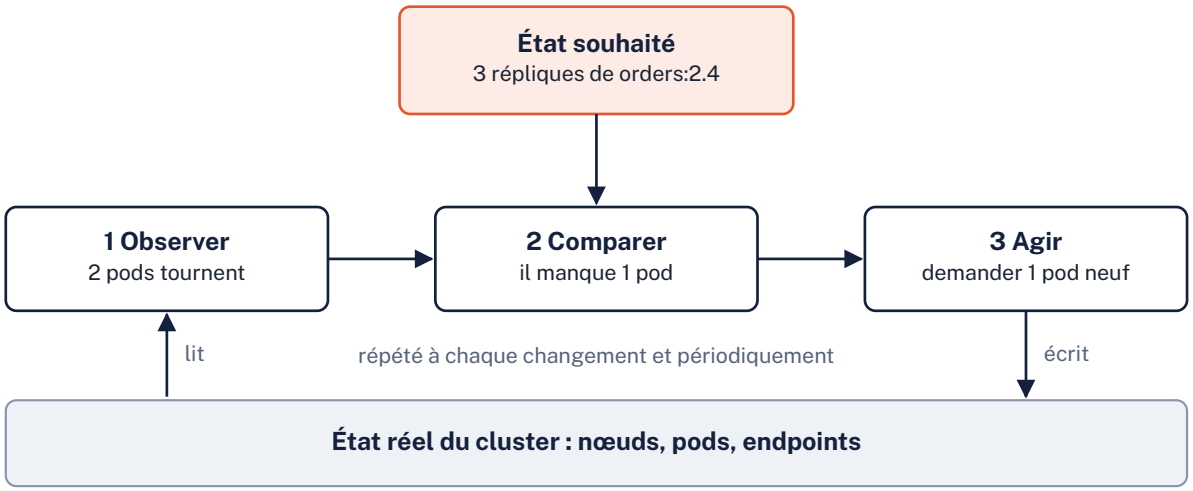
Chaque nœud de travail exécute un **kubelet**, un agent qui surveille le serveur d'API pour repérer les pods attribués à son nœud, demande au **runtime de conteneurs** (généralement containerd ou CRI-O, via la Container Runtime Interface ; la couche d'adaptation intégrée pour Docker Engine a été supprimée dans la version 1.24 en 2022) de les démarrer, exécute leurs sondes de santé et renvoie leur état. **kube-proxy** programme les règles de transfert de paquets du nœud pour que le trafic envoyé à l'adresse virtuelle d'un Service atteigne l'un de ses pods ; certains plug-ins réseau, comme Cilium, le remplacent par un plan de données eBPF. Un plug-in réseau donne à chaque pod sa propre adresse IP, et un service DNS de cluster (généralement CoreDNS) permet aux pods de trouver les Services par leur nom.

Remarquez le sens des flèches. Le plan de contrôle ne se connecte pas à un nœud pour y lancer des charges de travail ; les nœuds surveillent l'état dans lequel ils doivent se trouver et convergent vers lui. (Les commandes à la demande sont l'exception : pour `kubectl exec`, `logs` et `port-forward`, le serveur d'API appelle le kubelet sur le nœud.) Cette conception en mode pull, fondée sur la surveillance et la convergence, est ce qui rend Kubernetes robuste : un nœud déconnecté pendant dix minutes se remet simplement à niveau à son retour. Avec un Kubernetes managé (Amazon EKS, Azure Kubernetes Service, Google Kubernetes Engine et d'autres), le fournisseur exploite et met à niveau le plan de contrôle ; le client choisit toujours les types de nœuds, met à niveau les pools de nœuds et reste responsable de tout ce qui est déployé dans le cluster.

État souhaité déclaratif et boucles de réconciliation

L'idée centrale de Kubernetes est la configuration déclarative. Vous ne dites pas au cluster de démarrer trois conteneurs ; vous enregistrez un objet indiquant que trois copies d'une image donnée doivent exister : c'est l'état souhaité. Des contrôleurs exécutent alors des boucles de réconciliation : observer l'état réel, le comparer à l'état souhaité et agir pour supprimer la différence, indéfiniment.

La boucle de réconciliation



Les contrôleurs sont déclenchés par niveau : ils agissent sur la différence actuelle, et non sur une suite d'événements mémorisée, si bien que les événements manqués et les redémarrages ne causent aucun dommage durable.

Suivez ce qui se passe quand un nœud de la zone B tombe en panne pendant le coup de feu du soir. Le contrôleur Node cesse de recevoir des heartbeats et marque le nœud comme non prêt ; après un délai d'expiration, ses pods sont marqués pour éviction. Le contrôleur ReplicaSet du service des commandes observe deux pods en cours d'exécution là où trois sont souhaités et crée un nouvel objet pod. L'ordonnanceur voit un pod non planifié et l'attribue à un nœud sain, en privilégiant les nœuds survivants de la zone B ou une autre zone selon les règles de répartition. Le kubelet de ce nœud voit l'attribution, télécharge l'image et démarre le conteneur. Quand sa sonde de disponibilité réussit, les endpoints derrière le Service des commandes sont mis à jour et le trafic lui parvient. Aucun composant n'a dit à un autre quoi faire ; chacun a réagi à l'état qu'il observait.

Comme les contrôleurs sont *déclenchés par niveau* (ils réagissent à la différence actuelle) plutôt que *déclenchés par front* (en réagissant à des événements individuels), un contrôleur qui plante, manque une notification ou s'exécute deux fois converge tout de même vers le bon résultat. Le même patron peut être étendu : les *définitions de ressources personnalisées* (custom resource definitions) ajoutent de nouveaux types d'objets, et le patron *opérateur*, introduit par CoreOS en 2016, empaquette un contrôleur qui encode un savoir-faire d'exploitation, par exemple créer, sauvegarder et faire basculer un cluster PostgreSQL à partir d'un seul objet déclaré. La section 10.4 montre la même boucle appliquée à des environnements entiers grâce à GitOps.

Le contrôle déclaratif a un coût : vous devez raisonner en termes d'états finaux et renoncer à un séquencement exact. Quand deux contrôleurs veulent des états souhaités différents pour un même champ, ils se battent. Un exemple classique est un manifeste de Deployment dans Git qui indique `replicas: 3` alors qu'un autoscaler le fixe à neuf ; chaque synchronisation GitOps réduit le service en plein coup de feu. La solution consiste à confier chaque champ à un seul propriétaire, ici en retirant `replicas` du manifeste.

Les objets de base de Kubernetes

Objet	Ce que c'est	Exemple de livraison de repas
Pod	Un ou plusieurs conteneurs planifiés ensemble sur un même nœud, qui partagent une adresse IP et des volumes. Les pods sont jetables : on les remplace, on ne les répare jamais.	Une copie du service des commandes, éventuellement avec un petit sidecar d'expédition des journaux.
ReplicaSet	Maintient en fonctionnement un nombre donné de pods identiques. Rarement écrit à la main.	Créé et géré par le Deployment des commandes.
Deployment	Déclare le modèle de pod et le nombre de répliques d'une application sans état, et déploie les changements en faisant passer les pods d'un ancien ReplicaSet à un nouveau, dans des limites de dépassement et d'indisponibilité.	Services des commandes, de la carte et de la tarification.
Service	Un nom stable et une IP virtuelle pour un ensemble changeant de pods sélectionnés par label, avec une répartition de la charge entre eux.	`orders.default.svc`, utilisé par le service de paiement.
Ingress et Gateway API	Acheminent le trafic HTTP(S) externe vers les Services selon l'hôte et le chemin. Gateway API (disponible en version stable depuis la 1.0 en octobre 2023) est le successeur plus expressif et orienté rôles : l'équipe plateforme possède la Gateway, les équipes applicatives possèdent leurs HTTPRoutes.	`api.example.com/orders` acheminé vers le Service des commandes, avec TLS.
ConfigMap	Configuration non secrète fournie aux pods sous forme de variables d'environnement ou de fichiers.	Réglages de fonctionnalités, délais d'expiration, URL du service de la carte.
Secret	Valeurs sensibles fournies comme les ConfigMaps. Stockées dans etcd seulement encodées en base64, sauf si le chiffrement au repos est configuré ; de nombreuses équipes les synchronisent depuis un gestionnaire de secrets externe.	La clé d'API du prestataire de paiement, le mot de passe de la base de données.
StatefulSet	Pods dotés de noms stables, d'identités réseau stables et de leurs propres volumes persistants, démarrés et arrêtés dans l'ordre.	Un cluster Kafka ou Redis autogéré, si l'on n'utilise pas un service managé.

Objet	Ce que c'est	Exemple de livraison de repas
DaemonSet	Exactement un pod sur chaque nœud (ou sur chaque nœud sélectionné).	Collecteur de journaux ou agent de supervision des nœuds.
Job et CronJob	Pods qui s'exécutent jusqu'à leur terme, une fois ou selon un planning, avec de nouvelles tentatives.	Migration de base de données avant une mise en production ; rapport nocturne des versements aux restaurants.
HorizontalPodAutoscaler	Ajuste le nombre de répliques d'un Deployment pour maintenir une métrique proche d'une cible.	Faire passer le service des commandes de 3 à 12 répliques à mesure que l'utilisation du CPU augmente le soir.

REMARQUE

Les contrôleurs Ingress et le passage à Gateway API

L'objet Ingress ne fait que déclarer des règles de routage ; c'est un *contrôleur Ingress* qui les met en œuvre. En novembre 2025, le projet Kubernetes a annoncé la mise à la retraite du contrôleur communautaire très utilisé ingress-nginx, avec une maintenance au mieux prenant fin en mars 2026 et plus aucun correctif de sécurité ensuite, et a recommandé Gateway API pour les nouveaux travaux. L'API Ingress elle-même fonctionne toujours avec d'autres contrôleurs, mais les équipes qui utilisaient ingress-nginx ont dû planifier une migration : un rappel que même un composant standard de fait d'une plateforme a un cycle de vie qu'il faut suivre.

Un Deployment et un Service en YAML

Les manifestes ci-dessous déclarent le service des commandes : configuration issue d'un Secret, requêtes de ressources et sondes, un Service stable placé devant et un autoscaler qui maintient entre trois et douze répliques. Le Deployment n'a délibérément pas de champ `replicas`, afin que l'autoscaler soit le seul propriétaire de ce nombre. Ils sont appliqués avec `kubectl apply` ou, mieux, par un pipeline ou un agent GitOps à partir d'un dépôt.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: orders
spec:
  # no replicas field: the autoscaler below owns the count
  selector:
    matchLabels: {app: orders}
  template:
    metadata:
      labels: {app: orders}
    spec:
      containers:
      - name: orders
        image: registry.example.com/orders:2.4.1
        ports: [{containerPort: 8080}]
        env:
        - name: DATABASE_URL
          valueFrom: {secretKeyRef: {name: orders-db, key: url}}
        resources:
          requests: {cpu: 500m, memory: 512Mi}
          limits: {memory: 512Mi}  # no CPU limit: may burst
        readinessProbe:
          httpGet: {path: /ready, port: 8080}
          periodSeconds: 5
        livenessProbe:
          httpGet: {path: /live, port: 8080}
          periodSeconds: 10
          failureThreshold: 3
```

Un Deployment pour le service des commandes. En production, l'image serait épinglée par digest plutôt que par tag.

```
apiVersion: v1
kind: Service
metadata:
  name: orders
spec:
  selector: {app: orders}
  ports: [{port: 80, targetPort: 8080}]
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: orders
spec:
  scaleTargetRef: {apiVersion: apps/v1, kind: Deployment, name: orders}
  minReplicas: 3
  maxReplicas: 12
  metrics:
  - type: Resource
    resource:
      name: cpu
      target: {type: Utilization, averageUtilization: 60}
```

Un Service qui donne aux pods une adresse stable unique, et un autoscaler qui vise une utilisation moyenne de 60 % du CPU demandé.

Requêtes de ressources, limites et qualité de service

Les requêtes et limites (requests and limits) sont les lignes les plus lourdes de conséquences d'un manifeste. Une *requête* est ce que l'ordonnanceur réserve : un pod qui demande 500 millicœurs (un demi-CPU) et 512 Mio de mémoire n'est placé que sur un nœud disposant d'autant de capacité non réservée, et cette part lui est garantie en cas de contention. Une *limite* est un plafond imposé par le noyau : un conteneur qui dépasse sa limite de CPU est bridé (ralenti), et celui qui dépasse sa limite de mémoire est tué puis redémarré, avec le statut bien connu `OOMKilled`. Kubernetes déduit de ces réglages une classe de qualité de service : *Guaranteed* (requêtes égales aux limites pour chaque ressource), *Burstable* (requêtes définies, limites plus élevées ou absentes) et *BestEffort* (rien de défini, premier à être évincé quand un nœud manque de ressources).

Les praticiens ne s'accordent pas sur les limites de CPU. Un camp les définit partout, pour la prévisibilité et l'équité entre équipes ; l'autre définit des requêtes de CPU précises et aucune limite de CPU, car le bridage provoque des pics de latence même quand le nœud a du CPU inactif, et les requêtes garantissent déjà sa part à chaque pod. Pour la mémoire, le débat est mince : fixez la limite égale à la requête, car la mémoire ne peut pas être reprise à un processus sans le tuer, et un pod qui utilise plus que sa requête est le premier candidat à l'éviction. Dans

tous les cas, les requêtes doivent reposer sur des mesures ; des requêtes gonflées gaspillent de l'argent (le nœud est réservé mais inactif), et des requêtes sous-estimées provoquent des problèmes de voisins bruyants et un autoscaling trompeur.

Le HorizontalPodAutoscaler calcule *répliques souhaitées = plafond(répliques actuelles × métrique actuelle ÷ cible)*, en mesurant l'utilisation du CPU en pourcentage de la requête. Il augmente rapidement le nombre de répliques mais, par défaut, attend une fenêtre de stabilisation de cinq minutes avant de le réduire, pour éviter les oscillations. Quand les nouveaux pods ne tiennent pas sur les nœuds existants, ils restent *Pending*, et un autoscaler de nœuds (le Cluster Autoscaler, ou Karpenter chez certains fournisseurs) ajoute des machines, ce qui prend des minutes plutôt que des secondes. L'autoscaling des pods et celui des nœuds sont des boucles distinctes, avec des délais distincts, et toutes deux doivent être prévues pour le coup de feu.

EXEMPLE COMMENTÉ

Exemple commenté : dimensionner le service des commandes pour le coup de feu du soir

Chaque pod du service des commandes demande 500m de CPU et 512 Mio, avec une limite de mémoire de 512 Mio et aucune limite de CPU. Les nœuds ont 4 vCPU et 16 Gio, dont environ 3,8 vCPU et 14,5 Gio sont allouables après les réservations du système (chiffres illustratifs). Le HPA vise 60 % de CPU, avec 3 à 12 répliques.

Pods par nœud	CPU : 3,8 ÷ 0,5 = 7 pods. Mémoire : 14,5 Gio ÷ 0,5 Gio = 29 pods. Le CPU est la contrainte déterminante : **7 pods de commandes par nœud** si rien d'autre n'y tourne.
18:00, début du coup de feu	4 répliques à 90 % en moyenne de leur requête de CPU. Souhaité = plafond(4 × 90 ÷ 60) = **6 répliques**.
18:20, la charge monte encore	6 répliques à 85 %. Souhaité = plafond(6 × 85 ÷ 60) = plafond(8,5) = **9 répliques**, soit 4,5 vCPU demandés au total.
Capacité des nœuds	Si d'autres services remplissent déjà les nœuds, trois des nouveaux pods restent Pending ; l'autoscaler de nœuds ajoute un nœud, qui met peut-être deux à quatre minutes à rejoindre le cluster et à télécharger l'image.
Réponse de conception	Relevez `minReplicas` avant le coup de feu prévisible (mise à l'échelle planifiée), gardez de la capacité de nœuds en réserve et gardez l'image petite pour que les nouveaux pods démarrent vite.
Effet d'une requête erronée	Si chaque pod n'avait réellement besoin que de 250m mais que la requête restait à 500m, la même charge de coup de feu (225m par pod) apparaîtrait à 45 % au lieu de 90 % : le HPA n'augmenterait pas le nombre de répliques avant que l'utilisation dépasse 300m par pod, et le cluster réserverait deux fois le CPU dont il a besoin.

L'autoscaling réagit à une charge déjà arrivée ; pour un pic quotidien prévisible, une mise à l'échelle planifiée à l'avance coûte moins cher que des commandes perdues.

Les requêtes pilotent à la fois l'ordonnanceur et l'autoscaler : les mesurer est donc une tâche architecturale de premier ordre, et non un détail de réglage.

Sondes de vivacité, de disponibilité et de démarrage

Les sondes sont le moyen par lequel Kubernetes s'informe de la santé des conteneurs. Une sonde de **disponibilité** (readiness) indique si ce pod doit recevoir du trafic maintenant ; tant qu'elle échoue, le pod est retiré des endpoints de son Service mais continue de tourner. Une sonde de **vivacité** (liveness) indique si le processus est bloqué sans espoir de rétablissement ;

quand elle échoue de façon répétée, le kubelet redémarre le conteneur. Une sonde de **démarrage** (startup) retient les deux autres jusqu'à ce qu'une application lente à démarrer ait fini de s'initialiser.

L'erreur la plus dommageable est une sonde de vivacité qui vérifie les dépendances. Si l'endpoint de vivacité du service des commandes interroge la base de données, un basculement de trente secondes de la base fait échouer en même temps la sonde de vivacité de tous les pods des commandes, Kubernetes les redémarre tous, ils se reconnectent tous au même instant, et un bref incident devient une panne. La sonde de vivacité ne doit vérifier que la capacité du processus lui-même à répondre ; la sonde de disponibilité peut tenir compte de la capacité du pod à faire un travail utile, même si, là encore, de nombreuses équipes évitent de rendre tous les pods indisponibles à cause d'une dépendance partagée, et préfèrent faire échouer rapidement les requêtes avec un disjoncteur (voir patrons de résilience).

Les sondes fonctionnent de concert avec l'arrêt propre (graceful shutdown). Quand un pod est supprimé pendant un déploiement ou une réduction du nombre de répliques, Kubernetes le retire des endpoints et envoie `SIGTERM` au processus ; celui-ci doit cesser d'accepter du nouveau travail, terminer les requêtes en cours et s'arrêter dans le délai de grâce (30 secondes par défaut) avant d'être tué. Comme le retrait des endpoints se propage de façon asynchrone, un court délai `preStop` évite d'envoyer des requêtes à un pod qui a déjà commencé à s'arrêter. La section 10.3 y revient avec le principe de jetabilité de l'application douze facteurs.

Quand Kubernetes est surdimensionné

Kubernetes est une plateforme généraliste pour construire des plateformes, et il fait payer cette généralité. Il faut que quelqu'un conçoive le réseau, l'entrée du trafic, l'identité, les secrets, les politiques, l'observabilité et la multilocation ; suive trois versions mineures par an, chacune prise en charge pendant environ quatorze mois, ainsi que les extensions qui en dépendent ; et assure l'astreinte pour le cluster en plus des applications. Les plans de contrôle managés retirent une partie de ce travail, pas l'essentiel.

Pour une petite équipe qui exploite une poignée de services web sans état, un PaaS ou une plateforme de conteneurs serverless (Google Cloud Run, Azure Container Apps, Amazon ECS avec Fargate et autres) offre l'essentiel des avantages (conteneurs, autoscaling, déploiements progressifs, TLS managé) pour une fraction de la charge d'exploitation. Kubernetes commence à être rentable quand il y a de nombreux services et plusieurs équipes, une équipe plateforme pour l'exploiter, des charges de travail aux besoins inhabituels (systèmes avec état, GPU, traitements par lots, contrôleurs personnalisés), une obligation de faire tourner la même plateforme sur site et dans plusieurs clouds, ou un écosystème d'outils que l'organisation veut

utiliser. Beaucoup d'organisations l'ont adopté parce qu'il était à la mode, puis ont passé des années à construire, en pratique, un PaaS moins bon par-dessus. La section 10.7 explique comment les équipes plateforme évitent ce sort (ingénierie de plateforme).

SI VOUS DÉBUTEZ

Si vous débutez

Vous n'avez pas besoin de mémoriser chaque objet Kubernetes. Retenez trois idées : vous décrivez ce que vous voulez dans des fichiers et le système le réalise ; de petites unités jetables (les pods) sont remplacées plutôt que réparées ; et des noms stables (les Services) masquent le fait que les copies qui se trouvent derrière changent sans cesse. Tout le reste n'est que détail construit sur ces trois idées.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Pourquoi un déploiement en production doit-il désigner une image par son digest plutôt que par son tag ?**

Un tag est une étiquette mobile qui peut être redirigée vers un autre contenu ; un digest est une empreinte du contenu et identifie toujours exactement l'image qui a été testée.

▼ **Un nœud meurt. Quels composants agissent, et dans quel ordre, pour rétablir la troisième réplique ?**

Le contrôleur Node marque le nœud comme non prêt et ses pods sont évincés ; le contrôleur ReplicaSet constate qu'il manque des pods et en crée un ; l'ordonnanceur l'attribue à un nœud sain ; le kubelet de ce nœud le démarre via le runtime ; quand la sonde de disponibilité réussit, le pod rejoint les endpoints du Service.

▼ **Qu'arrive-t-il à un conteneur qui dépasse sa limite de mémoire, et à un conteneur qui dépasse sa limite de CPU ?**

Dépasser la limite de mémoire entraîne l'arrêt forcé du conteneur (OOMKilled) puis son redémarrage ; dépasser la limite de CPU ne fait que le brider.

▼ **Pourquoi est-il dangereux de vérifier la base de données dans une sonde de vivacité ?**

Un bref problème de base de données fait échouer ensemble la sonde de vivacité de tous les pods, si bien que Kubernetes les redémarre tous en même temps, transformant un court incident de dépendance en panne complète.

▼ **À vous : Chaque pod demande 750m de CPU et 1 Gio. Les nœuds offrent 7,6 vCPU et 28 Gio allouables. Le coup de feu nécessite 24 pods. Combien de pods tiennent par nœud, et de combien de nœuds avez-vous besoin si vous devez survivre à la perte de l'un d'eux ?**

Le CPU permet 7,6 / 0,75 = 10,1, soit 10 pods ; la mémoire en permet 28 : le CPU est donc la limite contraignante. Vingt-quatre pods exigent 2,4, soit 3 nœuds. Survivre à une panne de nœud en demande un de plus : 4 nœuds. Les requests pilotent l'ordonnancement : gardez-les donc proches de l'usage réel. Répartissez les pods entre les zones. Vérifiez que le maximum de l'autoscaler autorise 24 répliques, et que le cluster peut ajouter des nœuds avant le coup de feu.

SOURCES

- Kubernetes documentation: Cluster components: https://kubernetes.io/docs/concepts/overview/components/
- Kubernetes documentation: Resource management for pods and containers: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/
- Kubernetes documentation: Configure liveness, readiness and startup probes: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/
- Open Container Initiative: https://opencontainers.org/
- Kubernetes blog (2025): Ingress NGINX retirement: https://kubernetes.io/blog/2025/11/11/ingress-nginx-retirement/

SECTION 10.3 SUR 10.8 · INTERMÉDIAIRE · 25 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

L’application douze facteurs et les principes cloud native

AVANT DE LIRE

Cette section s’appuie sur :

- 0.8 Comment les logiciels sont construits
- 9.2 Équilibrage de charge
- 10.2 Conteneurs et Kubernetes

Termes déjà vus qu’elle utilise : Image de conteneur, Livraison continue, Idempotent, Scalabilité horizontale (scaling out), Observabilité

L’IDÉE EN BREF

Pensez à une chaîne de cuisines de plats à emporter. Une bonne fiche recette fonctionne dans n’importe quelle succursale : elle énumère chaque ingrédient, ne dit rien de la succursale où elle se trouve, et n’importe quel cuisinier peut la reprendre en plein service, car rien d’important n’est gardé dans la tête d’un seul cuisinier. Les livraisons arrivent dans des cartons étiquetés, et quand la carte change, une nouvelle fiche est imprimée et numérotée au lieu d’être griffonnée. L’application douze facteurs est ce genre de fiche recette pour le logiciel : des règles qui permettent à des machines de démarrer, copier, déplacer, remplacer et mettre à l’échelle la même application sans que personne ait besoin d’en connaître les bizarreries.

CE QU’IL FAUT SAVOIR

- The Twelve-Factor App (Adam Wiggins, Heroku, 2011, passé en open source en 2024) décrit le contrat entre une application et une plateforme moderne, et s’applique aussi bien aux monolithes qu’aux microservices.
- La configuration (III) et les services externes (IV) gardent hors de l’artefact tout ce qui varie d’un déploiement à l’autre, si bien qu’un même artefact tourne dans chaque environnement ; les secrets relèvent désormais d’un gestionnaire de secrets ou de l’identité de charge de travail plutôt que de simples variables d’environnement.
- Build, release, run (V) rend les versions immuables et numérotées, de sorte qu’un retour arrière consiste à exécuter une version antérieure et que personne ne modifie un serveur en fonctionnement.

- Les processus sans état (VI), la concurrence par type de processus (VIII) et la jetabilité (IX) sont ce qui rend sûrs la scalabilité horizontale, les mises à jour progressives et le remplacement des instances défaillantes.
- La parité dev/prod (X), les journaux sous forme de flux (XI) et les processus d'administration exécutés comme des processus ponctuels distincts (XII) réduisent les surprises en production ; les migrations doivent être compatibles avec l'ancien et le nouveau code.
- Au-delà des douze facteurs, les applications cloud native sont conçues pour la jetabilité, l'immuabilité, l'observabilité et la sécurité, et l'extension proposée par Hoffman en 2016 ajoute l'API first, la télémétrie ainsi que l'authentification et l'autorisation.
- Un audit au regard des facteurs doit classer les écarts par risque : d'abord les secrets exposés, puis tout ce qui bloque la plateforme cible, comme l'état local, puis la fiabilité et la commodité.

D'où viennent les douze facteurs

En 2011, Adam Wiggins, cofondateur de Heroku, a publié *The Twelve-Factor App*, un court document web qui condensait ce que l'équipe de Heroku avait appris en observant un très grand nombre d'applications construites, déployées et mises à l'échelle sur sa plateforme. La méthodologie de l'application douze facteurs (twelve-factor app) décrit des applications qui se mettent en place à partir d'étapes déclaratives et automatisées, ont un contrat clair avec la plateforme sous-jacente afin de pouvoir tourner n'importe où, gardent le développement et la production assez proches pour permettre le déploiement continu, et peuvent monter en charge horizontalement sans refonte.

Le document est antérieur à Docker (2013) et à Kubernetes (2014), et pourtant il décrit presque exactement le contrat que ces plateformes ont imposé par la suite : une application est un ensemble de processus sans état, assemblés en un artefact immuable, configurés de l'extérieur, qui écrivent leurs journaux dans un flux et communiquent avec tout le reste par le réseau. C'est pourquoi il reste le vocabulaire commun de la conception d'applications cloud native. En novembre 2024, Heroku a annoncé que la méthodologie devenait open source, maintenue par une communauté de contributeurs issus de plusieurs entreprises, dans le but de l'actualiser selon les pratiques actuelles ; ses douze facteurs d'origine restent le point de référence.

Les facteurs portent sur la relation de l'application avec sa plateforme d'exécution, pas sur son architecture interne. Un monolithe modulaire peut parfaitement respecter les douze facteurs ; un ensemble de microservices peut les enfreindre tous. Ils complètent les préoccupations de déployabilité et d'exploitabilité du chapitre 2 et les principes de mise à l'échelle et d'absence d'état du chapitre 9.

Les douze facteurs, avec leurs réserves actuelles

Facteur	Ce que cela signifie	Pourquoi c'est important	Réserve actuelle
I. Base de code	Une seule base de code sous gestion de versions par application, déployée dans de nombreux environnements.	Chaque environnement exécute une version du même code ; pas de forks par client ni par environnement.	Un monorepo contenant plusieurs applications convient si chacune a une frontière claire ; l'anti-pattern, ce sont les forks par copier-modifier.
II. Dépendances	Déclarez explicitement chaque dépendance et isolez l'application du système hôte.	Les builds sont reproductibles sur n'importe quelle machine ; rien ne repose sur un paquet que quelqu'un a installé à la main.	Les lockfiles et les images en font le comportement par défaut ; les dépendances épinglées doivent tout de même être mises à jour régulièrement et analysées pour détecter les vulnérabilités.
III. Configuration	Gardez tout ce qui varie d'un déploiement à l'autre (URL, identifiants, réglages de fonctionnalités) hors du code, dans l'environnement.	Un même artefact tourne dans chaque environnement ; les secrets n'entrent jamais dans le dépôt.	Les variables d'environnement fuient dans les journaux, les rapports de plantage et les processus enfants ; il vaut mieux lire les secrets depuis un gestionnaire de secrets ou des fichiers montés.
IV. Services externes	Traitez les bases de données, files d'attente, caches et API externes comme des ressources attachées, localisées par la configuration.	Une base de données locale peut être remplacée par une base managée, ou une base défaillante par une copie restaurée, en changeant une URL.	Les substitutions sont rarement transparentes quand la sémantique diffère ; isolez les différences derrière des ports et des adaptateurs.
V. Build, release, run	Séparez l'assemblage d'un artefact, sa combinaison avec la configuration en une version (release), et son exécution.	Les versions sont immuables et numérotées ; toute version peut être réexécutée ; personne ne modifie le code sur un serveur.	Aujourd'hui : un seul build dans la CI, l'image stockée par digest, les versions enregistrées dans Git ou dans l'outil de déploiement.

Facteur	Ce que cela signifie	Pourquoi c'est important	Réserve actuelle
VI. Processus	Exécutez l'application comme un ou plusieurs processus sans état qui ne partagent rien ; gardez l'état dans des services externes.	N'importe quelle copie peut traiter n'importe quelle requête, donc on peut ajouter, retirer et remplacer des copies librement.	Les sessions persistantes masquent le problème au lieu de le résoudre ; le disque local n'est qu'un espace de travail temporaire.
VII. Associations de ports	L'application est autonome et expose son service en écoutant sur un port.	La plateforme lui achemine le trafic sans l'injecter dans un serveur web distinct.	Universel aujourd'hui ; les consommateurs de files d'attente n'ont pas de port, et un service mesh peut se placer devant l'application.
VIII. Concurrence	Montez en charge en exécutant davantage de processus, avec des types de processus différents pour des travaux différents (web, worker).	Chaque type de travail passe à l'échelle indépendamment, selon sa propre charge.	Les autoscalers le font automatiquement, y compris selon la longueur des files ; les threads au sein d'un processus comptent toujours.
IX. Jetabilité	Démarrer vite, s'arrêter proprement sur SIGTERM, survivre à une mort subite.	Mise à l'échelle rapide, déploiements progressifs sûrs et robustesse face aux plantages.	Les runtimes lents à démarrer ont besoin de sondes de démarrage (startup probes) ou de techniques de démarrage plus rapide ; les tâches d'arrière-plan doivent être idempotentes et pouvoir reprendre.
X. Parité dev/prod	Gardez le développement, le staging et la production aussi semblables que possible en matière de délais, de personnes et d'outils.	Moins de surprises qui n'apparaissent qu'en production ; le déploiement continu devient sûr.	La parité totale coûte trop cher ; donnez la priorité aux mêmes types et versions de services externes, en utilisant des conteneurs en local.

Facteur	Ce que cela signifie	Pourquoi c'est important	Réserve actuelle
XI. Journaux	Écrivez les journaux comme un flux d'événements sans tampon sur la sortie standard ; laissez la plateforme les acheminer.	L'application ne gère pas de fichiers ; les journaux de toutes les copies sont collectés de manière centralisée.	Utilisez des journaux structurés, ainsi que des métriques et des traces (par exemple via OpenTelemetry) ; surveillez le volume et le coût des journaux.
XII. Processus d'administration	Exécutez les tâches ponctuelles (migrations, scripts) comme des processus distincts utilisant le même code et la même configuration.	Les tâches d'administration s'exécutent sur exactement la version en production, pas sur une copie périmée.	Exécutez-les comme des étapes du pipeline ou des Jobs Kubernetes ; ne laissez jamais chaque réplica exécuter les migrations au démarrage sans verrou.

Configuration, secrets et services externes

Le facteur configuration a un test simple : la base de code pourrait-elle être rendue publique aujourd'hui sans exposer un seul identifiant ? Si ce n'est pas le cas, la configuration a fui dans le code. Séparer la configuration du code permet à un même artefact de traverser tous les environnements sans modification, ce qui est le fondement du principe build once (construire une seule fois) de la section 10.5 (CI/CD et déploiement). La configuration comprend les URL des bases de données et des services, les identifiants et les réglages propres à chaque environnement ; elle exclut le câblage interne identique partout, par exemple quelle classe traite quelle route, qui relève du code.

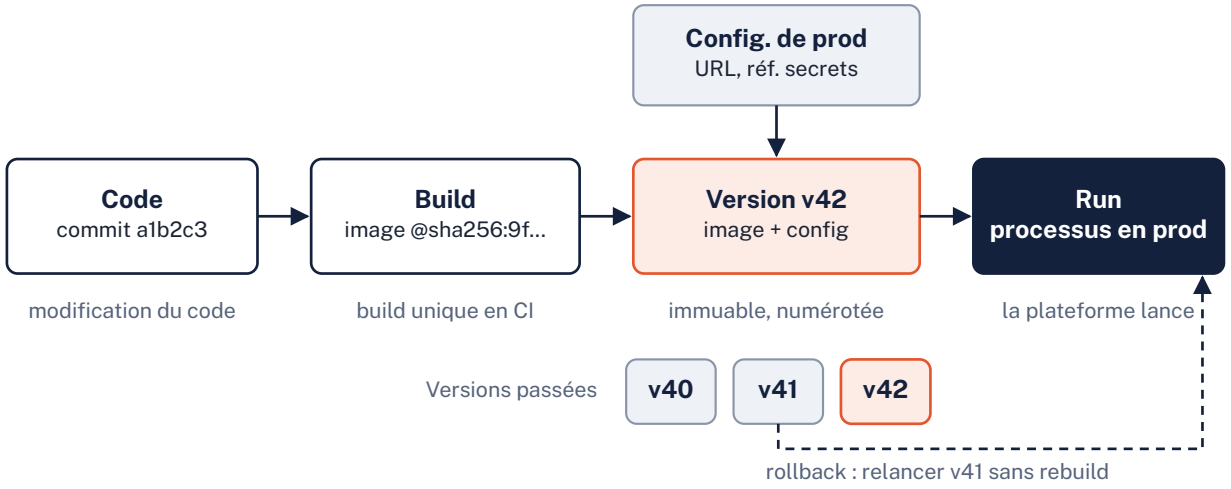
La recommandation d'origine d'utiliser des variables d'environnement a vieilli de façon inégale. Elles sont universelles et indépendantes du langage, et Kubernetes, les plateformes PaaS et les fonctions les prennent toutes en charge. Mais l'environnement d'un processus est visible par les processus enfants, se retrouve souvent recopié dans les rapports de plantage et les pages de débogage, et ne peut pas faire l'objet d'une rotation sans redémarrage. La pratique actuelle pour les secrets consiste à les récupérer au démarrage depuis un gestionnaire de secrets (AWS Secrets Manager, Azure Key Vault, Google Secret Manager, HashiCorp Vault ou OpenBao) ou à les monter sous forme de fichiers que la plateforme peut mettre à jour, et mieux encore à remplacer les secrets à longue durée de vie par l'identité de charge de travail, où la plateforme donne automatiquement à chaque service des identifiants de courte durée. L'intention du facteur, garder la configuration hors de l'artefact, reste inchangée.

Traiter un service externe (backing service) comme une ressource attachée signifie que le code atteint la base de données des commandes, le prestataire de paiement et le courtier de messages de la même façon : par une URL et des identifiants fournis au moment du déploiement. Cela rend les environnements composables et la reprise plus rapide. Cela ne rend pas pour autant les services externes interchangeables par magie ; remplacer un courtier de messages par un autre change la sémantique de livraison. Le style hexagonal du chapitre 5 complète le facteur : la configuration dit où se trouve la ressource, et un adaptateur derrière un port masque son comportement.

Build, release, run (assembler, publier, exécuter)

Le facteur build, release, run (assembler, publier, exécuter) sépare trois étapes souvent confondues. L'étape de *build* transforme un commit en artefact exécutable, aujourd'hui généralement une image de conteneur identifiée par son digest. L'étape de *release* combine cet artefact avec la configuration d'un environnement et donne au résultat une identité, par exemple la version v42 du service des commandes en production. L'étape de *run* démarre des processus à partir d'une version. Les versions sont immuables : une modification du code ou de la configuration produit une nouvelle version, jamais la modification d'une version existante.

Un seul build, une version par environnement, une exécution n'importe où



Comme les versions sont immuables et numérotées, revenir en arrière consiste à exécuter de nouveau une version antérieure, sans rien reconstruire ni modifier.

C'est cette séparation qui rend le déploiement ennuyeux. Personne ne modifie de fichiers sur un serveur, donc les serveurs ne dérivent pas ; l'image qui a passé les tests en staging est celle qui tourne en production, donc un résultat de test a un sens ; et le retour arrière est une opération rapide et rodée plutôt qu'une reconstruction en urgence. C'est aussi une piste d'audit : pour n'importe quel moment, vous pouvez dire exactement quel code et quelle

configuration tournaient. Kubernetes conserve un historique des révisions des ReplicaSets de chaque Deployment, les plateformes PaaS numérotent les versions, et GitOps (section 10.4) fait de l'historique Git du dépôt des versions la référence.

Processus, concurrence et jetabilité

Des processus sans état, qui ne partagent rien, sont la condition préalable de presque tout le reste de ce chapitre : scalabilité horizontale, déploiements progressifs, autoscaling, remplacement des nœuds défaillants et répartition sur plusieurs zones. Tout ce qu'un processus garde uniquement en mémoire ou sur son disque local (la session d'un utilisateur, un fichier envoyé, un rapport en partie construit) est perdu quand ce processus est remplacé, et reste invisible pour les autres copies. Le chapitre 9 traite en détail de la gestion des sessions (équilibrage de charge, absence d'état et sessions). L'état ne disparaît pas ; il est déplacé vers des services externes conçus pour le conserver.

Le facteur concurrence ajoute que les différents types de travail doivent s'exécuter comme des types de processus différents qui passent à l'échelle séparément. Dans la plateforme de livraison de repas, le processus web qui sert l'app client passe à l'échelle selon le débit de requêtes, tandis qu'un processus worker qui envoie les confirmations de commande passe à l'échelle selon la longueur de la file, et un consommateur des positions des livreurs selon le débit d'événements. Les mélanger dans un seul type de processus oblige à tous les mettre à l'échelle pour les besoins d'un seul.

La jetabilité (disposability) exige que les processus démarrent en quelques secondes et s'arrêtent proprement. À la réception de `SIGTERM`, un processus web cesse d'accepter des connexions et termine les requêtes en cours ; un worker remet son message courant dans la file ou le termine dans le délai de grâce. Comme les processus peuvent aussi mourir sans prévenir (une panne de nœud, un arrêt forcé pour mémoire insuffisante), le travail doit pouvoir être répété sans danger : traitement idempotent des messages, comme dans l'architecture pilotée par les événements, et tâches qui reprennent à partir de points de contrôle. George Candea et Armando Fox ont poussé l'idée à l'extrême en 2003 avec le *crash-only software*, qui n'a qu'une façon de s'arrêter (planter) et une façon de démarrer (la reprise), de sorte que le chemin de reprise est exercé en permanence au lieu d'être découvert pendant un incident.

SI VOUS DÉBUTEZ

Pas à pas : un worker à qui l'on demande de s'arrêter

L'idée qui demande le plus d'attention ici est la jetabilité : un processus doit s'attendre à être arrêté à tout moment, poliment ou non, et ne rien perdre quand cela arrive. Suivez un processus worker à travers les deux types d'arrêt.

- La plateforme de livraison de repas exécute un processus worker qui prend des messages dans une file et envoie les confirmations de commande. Lors d'un déploiement de routine, chaque ancienne copie du worker est remplacée par une nouvelle, donc chaque ancienne copie doit s'arrêter.
- La manière polie est un signal appelé SIGTERM, qui signifie « veuillez terminer et quitter », comme l'a décrit la section 10.2. La plateforme attend pendant un délai de grâce avant de forcer l'arrêt du processus.
- Un worker jetable réagit immédiatement. Il cesse de prendre de nouveaux messages, termine la confirmation qu'il est en train d'envoyer ou rend ce message à la file pour qu'une autre copie le reprenne, puis il quitte. Rien n'est perdu.
- Mais un processus peut aussi mourir sans prévenir : sa machine tombe en panne, ou il est tué parce qu'il utilise trop de mémoire. Il n'a aucune possibilité de faire le ménage.
- Supposons que le worker meure juste après avoir envoyé une confirmation, mais avant d'avoir signalé à la file qu'il avait terminé. Le message est proposé de nouveau et une autre copie le traite une seconde fois. Le traitement doit donc être idempotent, comme l'a dit le chapitre 5 : le faire deux fois a le même effet que le faire une fois. On y parvient généralement en donnant à chaque message une clé unique, transmise avec la confirmation, afin qu'une répétition soit reconnue et ignorée.
- **La leçon :** traitez l'arrêt et le plantage comme des événements de tous les jours. Un processus qui démarre vite, s'arrête proprement quand on le lui demande et ne perd rien quand il meurt sans prévenir peut être remplacé, déplacé et mis à l'échelle librement.

Parité dev/prod, journaux et processus d'administration

Le texte d'origine identifie trois écarts entre le développement et la production : le temps (du code écrit des semaines avant d'être déployé), le personnel (les développeurs l'écrivent, les opérateurs le déploient) et les outils (SQLite sur un portable, PostgreSQL en production). La livraison continue comble le premier, les équipes qui exploitent ce qu'elles construisent comblent le deuxième, et les conteneurs comblent une grande partie du troisième : un développeur peut exécuter en local la même version de PostgreSQL, le même courtier de messages et le même cache, et des outils comme Testcontainers démarrent de vraies

dépendances dans les tests d'intégration. Une parité parfaite n'est ni abordable ni nécessaire ; l'essentiel est d'utiliser les mêmes types et versions de services externes, car c'est là que se cachent les différences surprenantes.

Traiter les journaux comme des flux d'événements signifie que l'application écrit chaque événement sur la sortie standard et laisse à la plateforme le stockage, la rotation et l'expédition. Cela reste juste, avec deux ajouts modernes : les journaux doivent être structurés (par exemple en JSON, avec des noms de champs cohérents et un identifiant de requête ou de trace), et les journaux sont l'un des trois signaux, avec les métriques et les traces, qui rendent un système observable (observabilité). Le volume des journaux est aussi un vrai facteur de coût ; la section 10.8 y revient.

Les processus d'administration, comme les migrations de base de données, les réparations de données et les rapports ponctuels, doivent s'exécuter avec la même version (code et configuration) que l'application, sous forme de processus distincts. Dans Kubernetes, cela signifie généralement un Job exécuté par le pipeline avant le déploiement de la nouvelle version. Les migrations demandent un soin particulier : elles doivent être compatibles à la fois avec l'ancienne et la nouvelle version du code qui tournent côte à côte, ce qui correspond à la discipline d'expansion-contraction de l'évolution des schémas.

Au-delà des douze facteurs

Les douze facteurs n'ont jamais prétendu être complets, et des auteurs ultérieurs les ont étendus. *Beyond the Twelve-Factor App* de Kevin Hoffman (O'Reilly, 2016) a porté la liste à quinze, en ajoutant *API first* (concevoir le contrat avant l'implémentation), la *télémétrie* (intégrer métriques, santé et traçage) ainsi que l'*authentification et l'autorisation* (la sécurité comme préoccupation de conception dès le départ). La pratique actuelle ajoute quatre thèmes que toute application cloud native devrait respecter.

La jetabilité, prise au sérieux. Concevez pour être tué à tout moment : opérations idempotentes, délais d'expiration courts pour les requêtes, temps de démarrage borné, drainage en douceur, et aucun processus unique dont la perte compte. **L'immuabilité.** Ne corrigez jamais un conteneur ou un serveur en fonctionnement ; construisez une nouvelle image et remplacez-le (section 10.4 sur l'infrastructure immuable). Chaque instance en cours d'exécution devient ainsi traçable jusqu'à un commit, et la dérive disparaît. **L'observabilité.** Émettez des journaux structurés, des métriques et des traces via un standard comme OpenTelemetry, exposez des endpoints de santé qui reflètent l'état propre du processus, et propagez le contexte de trace d'un appel à l'autre, afin que les opérateurs puissent répondre à des questions qu'ils n'avaient pas anticipées. **La sécurité.** Exécutez sous un utilisateur non root, avec un système de fichiers en lecture seule quand c'est possible ; obtenez les identifiants par

l’identité de charge de travail plutôt que par des clés à longue durée de vie ; appliquez le moindre privilège à chaque service externe ; et construisez à partir d’images minimales, analysées et signées (sécurité de la chaîne d’approvisionnement).

EXEMPLE COMMENTÉ

Exemple commenté : l'audit du service d'inscription des restaurants

Le service d'inscription des restaurants de la plateforme de livraison de repas est une application web vieille de cinq ans, installée sur deux machines virtuelles montées à la main derrière un équilibreur de charge à sessions persistantes. L'équipe veut la déplacer sur la plateforme de conteneurs de l'entreprise. Un audit au regard des facteurs relève neuf problèmes ; chacun reçoit un risque, une correction et une priorité.

1. Mot de passe de la base de données de production dans un fichier de propriétés du dépôt (III)	Risque : toute personne ayant accès en lecture au code, y compris d'anciens prestataires, possède l'identifiant. Correction : renouveler le mot de passe immédiatement, le placer dans le gestionnaire de secrets, l'injecter au moment du déploiement. **Priorité 1**.
2. Cartes et logos envoyés, écrits sur le disque local de la VM (VI, IV)	Risque : les fichiers disparaissent quand une requête arrive sur l'autre VM ou qu'une VM est remplacée ; bloque tout passage à des conteneurs jetables. Correction : stockage objet, avec une copie ponctuelle des fichiers existants. **Priorité 1**.
3. Sessions dans la mémoire du processus, répartition de charge persistante (VI)	Risque : chaque déploiement déconnecte le personnel des restaurants au milieu d'un formulaire. Correction : jetons de session signés ou stockage de sessions partagé. **Priorité 2**.
4. Déploiement par SSH, `git pull` et redémarrage sur chaque VM (V, II)	Risque : les deux VM exécutent un code différent après un pull raté ; pas de retour arrière. Correction : la CI construit une image par commit ; versions numérotées. **Priorité 2**.
5. Migrations exécutées automatiquement au démarrage par les deux instances (XII)	Risque : les deux instances se disputent la même migration. Correction : une étape de migration distincte dans le pipeline, avec des changements en expansion-contraction. **Priorité 2** (priorité 1 si une telle situation de concurrence s'est déjà produite).
6. Arrêt par `kill -9` pendant les déploiements (IX)	Risque : formulaires d'inscription à moitié enregistrés. Correction : gérer SIGTERM, drainer les requêtes, ajouter une sonde de disponibilité (readiness) et un délai pre-stop. **Priorité 2**.
7. Journaux écrits dans des fichiers à rotation sur chaque VM (XI)	Risque : journaux perdus avec la VM ; recherche difficile d'une instance à l'autre. Correction : JSON structuré sur la sortie standard, collecté de manière centralisée. **Priorité 3**, peu coûteux à faire en même temps que le point 4.
8. Rapport nocturne des reversements en tâche cron sur la VM 1 seulement (XII, VIII)	Risque : la tâche s'arrête sans bruit quand la VM 1 est reconstruite, ou s'exécute deux fois si elle est copiée. Correction : une tâche planifiée sur la plateforme, rendue idempotente. **Priorité 3**.

9. Les développeurs utilisent SQLite, la production PostgreSQL (X)	Risque : du SQL qui fonctionne en local échoue en production. Correction : PostgreSQL dans un conteneur pour le développement et les tests. **Priorité 3**.

Les priorités suivent d'abord le risque (un identifiant exposé est un incident en cours), puis ce qui bloque la plateforme cible (l'état local), puis la fiabilité, puis la commodité.

L'équipe n'a pas besoin de tout corriger avant de migrer : les points 1 et 2 sont des prérequis ; les points 3 à 6 peuvent être traités pendant le déplacement ; les points 7 à 9 peuvent suivre.

Un audit comme celui-ci est aussi une bonne matière pour un enregistrement de décision d'architecture : il explique pourquoi chaque changement est fait, dans quel ordre, et ce qui a été délibérément reporté.

ATTENTION

Contresens fréquents sur les douze facteurs

- Sans état ne veut pas dire que le système n'a pas d'état ; cela veut dire que les processus ne le conservent pas. Déplacer l'état vers un cache unique et non répliqué ne fait que déplacer le problème.
- Les douze facteurs ne sont pas une définition des microservices. Ils s'appliquent tout autant à un monolithe bien construit, et un ensemble de services peut les enfreindre.
- Mettre les secrets dans des variables d'environnement respecte la lettre du facteur III, mais pas la pratique de sécurité actuelle ; préférez un gestionnaire de secrets et l'identité de charge de travail.
- La parité dev/prod ne signifie pas une copie complète de la production pour chaque développeur ; elle signifie les mêmes types et versions de dépendances, et de faibles écarts de délai et de personnes.
- L'association de ports et les journaux sur la sortie standard sont des gains faciles ; des équipes s'en contentent parfois et revendiquent la conformité tout en gardant des fichiers sur le disque local.

Questions fréquentes

Toute application doit-elle respecter les douze facteurs ?
Non. Ils forment une liste de contrôle de propriétés qui rendent le déploiement, la mise à l'échelle et la reprise peu coûteux sur une plateforme moderne. Traitez chaque écart comme un risque assorti d'un coût, et corrigez d'abord ceux qui bloquent votre plateforme ou menacent la sécurité, comme dans l'audit ci-dessus.

La méthodologie s'applique-t-elle aux fonctions serverless ?
En grande partie, et la plateforme en impose l'essentiel : les fonctions sont sans état, configurées de l'extérieur, jetables et journalisées par la plateforme. Les facteurs qui demandent encore de l'attention sont la configuration et les secrets, les services externes, l'idempotence et la parité dev/prod, car l'émulation locale des sources d'événements d'un fournisseur est imparfaite.

Les douze facteurs sont-ils encore pertinents maintenant que Kubernetes existe ?
Oui, car Kubernetes les présuppose. Un pod qui garde les sessions en mémoire, écrit des fichiers en local ou met deux minutes à démarrer contrarie les mises à jour progressives, l'autoscaling et la replanification de la plateforme. Les facteurs décrivent ce qu'une application doit faire pour que l'automatisation de la plateforme l'aide au lieu de lui nuire.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Quel test simple montre si la configuration a fui dans le code ?**

Demandez-vous si la base de code pourrait être rendue publique aujourd'hui sans exposer aucun identifiant. Si ce n'est pas le cas, la configuration (au moins les secrets) se trouve dans le code.

▼ **Pourquoi build, release, run rend-il le retour arrière rapide ?**

Chaque version est une combinaison immuable d'un artefact construit et d'une configuration, avec sa propre identité ; revenir en arrière consiste donc à exécuter de nouveau la version précédente, sans rien reconstruire ni modifier.

▼ **Un service stocke les fichiers envoyés sur le disque local de son serveur. Quel facteur cela enfreint-il, et qu'est-ce qui ne fonctionne plus ?**

Les processus (VI), et sans doute les services externes (IV) : les fichiers manquent quand les requêtes atteignent une autre copie, et sont perdus quand l'instance est remplacée, ce qui bloque la scalabilité horizontale et les conteneurs jetables.

▼ **Pourquoi les migrations ne devraient-elles pas s'exécuter automatiquement dans chaque réplica au démarrage ?**

Plusieurs réplicas risquent d'exécuter la même migration en même temps et d'entrer en concurrence, et le démarrage devient lent et fragile. Les migrations doivent s'exécuter une seule fois, comme une étape d'administration distincte dans le pipeline, avec la même version.

▼ **À vous : En auditant un service, vous trouvez : un mot de passe de production dans le dépôt ; des cartes importées enregistrées sur le disque local ; des journaux écrits dans un fichier ; des scripts d'administration ponctuels lancés en se connectant à un serveur. Classez ces éléments en vue d'un passage sur une plateforme de conteneurs.**

D'abord, le mot de passe : il est exposé dès maintenant, alors faites-le tourner et placez-le dans un gestionnaire de secrets ou utilisez l'identité de charge de travail. Ensuite, les fichiers locaux, car ils bloquent la migration : les conteneurs sont jetables, les envois vont donc dans un stockage objet utilisé comme service externe. Puis, les scripts ponctuels : exécutez-les comme processus d'administration séparés issus de la même version, afin qu'ils soient reproductibles et audités. Enfin, les journaux : écrivez-les sur stdout sous forme de flux. Le risque d'abord, puis les blocages, puis le confort.

SOURCES

- The Twelve-Factor App: https://12factor.net/
- The Twelve-Factor App: III. Config: https://12factor.net/config
- Twelve-Factor App methodology is now open source (2024): https://www.12factor.net/blog/open-source-announcement
- Kubernetes documentation: Configure liveness, readiness and startup probes: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

Infrastructure en tant que code et infrastructure immuable

AVANT DE LIRE

Cette section s'appuie sur :

- 0.7 Serveurs et cloud
- 0.8 Comment les logiciels sont construits
- 10.2 Conteneurs et Kubernetes

Termes déjà vus qu'elle utilise : État souhaité, Boucle de réconciliation, Pull request, Rayon d'impact (blast radius)

L'IDÉE EN BREF

Imaginez que vous meubliez un hôtel de deux cents chambres identiques. Vous pourriez dire à chaque décorateur quoi faire et espérer qu'il s'en souvienne, ou bien écrire une spécification précise pour une chambre, la ranger dans un classeur et faire approuver chaque modification par écrit. Avec la spécification, n'importe quelle chambre peut être vérifiée, une chambre endommagée peut être refaite à l'identique, et une nouvelle aile peut être aménagée en quelques jours. L'infrastructure en tant que code est cette spécification pour les ordinateurs, les réseaux et les bases de données sur lesquels tourne un système : écrite, relue, versionnée et appliquée par des machines plutôt que de mémoire.

CE QU'IL FAUT SAVOIR

- L'infrastructure en tant que code définit l'infrastructure dans des fichiers sous gestion de versions appliqués par des outils, ce qui rend les environnements reproductibles, vérifiables, auditables et rapides à créer.
- Les outils déclaratifs décrivent l'état final souhaité et calculent les étapes, ce qui rend les exécutions répétées idempotentes ; les scripts impératifs énumèrent des étapes et sont difficiles à relancer sans risque.
- HashiCorp a fait passer Terraform sous Business Source License en août 2023 ; le fork OpenTofu a suivi sous l'égide de la Linux Foundation (stable en janvier 2024, projet sandbox de la CNCF depuis 2025) ; CloudFormation, Bicep, Pulumi et Crossplane sont les principales alternatives.
- L'état associe le code aux ressources réelles ; il doit être distant, verrouillé, versionné, à accès contrôlé et découpé pour limiter le rayon d'impact, et la dérive doit être détectée par des plans exécutés à intervalles réguliers, puis codifiée ou annulée.

- Des modules versionnés aux interfaces réduites, réutilisés d'un environnement à l'autre avec des entrées différentes, empêchent les environnements copiés-collés de diverger.
- L'infrastructure immuable remplace les serveurs et les conteneurs à partir de nouvelles images au lieu de les modifier sur place ; la gestion de configuration reste utile pour les parcs à longue durée de vie et pour construire les images.
- GitOps conserve l'état souhaité dans Git et laisse des agents internes au cluster (Argo CD, Flux) le récupérer et le réconcilier en continu ; la politique en tant que code et des tests en couches rendent les changements sûrs.

Ce qu'est l'infrastructure en tant que code

L'infrastructure en tant que code (IaC) consiste à définir les réseaux, les machines virtuelles, les clusters, les bases de données, les files d'attente, les enregistrements DNS, les autorisations et tous les autres éléments d'infrastructure dans des fichiers texte conservés sous gestion de versions, et à laisser des outils créer et modifier les ressources réelles à partir de ces fichiers. Au lieu qu'un ingénieur clique dans une console cloud, un changement est un commit : relu dans une pull request, testé dans un pipeline, appliqué automatiquement et enregistré pour toujours.

L'idée est plus ancienne que le cloud public. CFEngine de Mark Burgess (1993) gérait des machines Unix à partir de politiques déclaratives ; Puppet (2005), Chef (2009) et Ansible (2012) ont popularisé la gestion de configuration. Le cloud l'a rendue indispensable, car l'infrastructure elle-même est devenue accessible par des API : AWS CloudFormation (2011) et Terraform de HashiCorp (2014) décrivent des environnements entiers, et pas seulement le logiciel installé sur les serveurs. Le livre de Kief Morris *Infrastructure as Code*, aujourd'hui dans sa troisième édition (2025), est l'ouvrage de référence sur cette pratique et ses principes.

Les bénéfices sont ceux du génie logiciel appliqué à l'infrastructure. Les environnements deviennent reproductibles : le staging peut être construit à partir des mêmes définitions que la production, et une région peut être reconstruite après un sinistre à partir du code plutôt que de mémoire. Les changements deviennent vérifiables et auditables, ce que les régulateurs et les équipes de sécurité apprécient. Le savoir cesse de vivre dans la tête d'un seul ingénieur. Et la rapidité augmente, car créer un environnement pour un nouveau service de la plateforme de livraison de repas devient une pull request plutôt qu'un ticket qui attend deux semaines.

Déclaratif ou impératif

Une approche *impérative* énumère les étapes : créer un réseau, puis un sous-réseau, puis une base de données, puis ouvrir tel port. Exécuter le script deux fois tente de tout créer deux fois. Une approche *déclarative* décrit l'état final, l'état souhaité de la section 10.2 : un réseau avec

ces sous-réseaux, une base de données de cette taille, un pare-feu qui autorise ce port. L'outil compare la description avec ce qui existe et en déduit les étapes, si bien que l'appliquer deux fois ne change rien la seconde fois. Cette propriété, l'*idempotence*, est ce qui permet d'exécuter du code d'infrastructure de façon répétée et en automatisation sans risque.

La plupart des outils matures sont déclaratifs au fond, mais ils diffèrent par le langage utilisé pour écrire la déclaration et par l'entité qui tient le registre de ce qui existe. Le tableau résume les principales familles.

Approches du code d'infrastructure

Approche	Vous écrivez	Exemples	Points forts	Faiblesses
Scripts impératifs	Des séquences d'appels CLI ou SDK	Scripts shell appelant les CLI des clouds, programmes SDK	Familiers, souples, rapides pour des tâches ponctuelles	Non idempotents sans grand soin ; aucun registre de ce qui existe ; difficiles à modifier sans risque
Langage déclaratif dédié	Des ressources et leurs relations dans un langage de configuration	Terraform et OpenTofu (HCL), AWS CloudFormation (YAML ou JSON), Azure Bicep, manifestes Kubernetes	Plans lisibles et vérifiables ; idempotents ; vastes écosystèmes de providers	Boucles, conditions et abstraction parfois malcommodes ; un langage à apprendre
Modèle déclaratif, langage généraliste	Du code en TypeScript, Python, Go, C# ou Java qui construit un modèle déclaratif	Pulumi, AWS Cloud Development Kit (CDK)	De vraies abstractions, des outils de test et des éditeurs ; familier pour les développeurs	Facile d'écrire une infrastructure astucieuse et difficile à relire ; le plan est moins visible dans le code
Réconciliation continue	Des ressources de style Kubernetes décrivant l'infrastructure cloud	Crossplane, contrôleurs Kubernetes des fournisseurs cloud	Même boucle de réconciliation et mêmes outils que pour les applications ; dérive corrigée en continu	Nécessite un plan de contrôle Kubernetes ; le débogage s'étend sur deux systèmes
Gestion de configuration	La configuration souhaitée des logiciels sur des serveurs existants	Ansible, Chef, Puppet	Gère les machines à longue durée de vie, les équipements réseau et les parcs sur site	Les serveurs accumulent un historique et dérivent ; plus lent et moins prévisible que de les remplacer

Les outils et le fork OpenTofu

Terraform est devenu l'outil multicloud le plus utilisé parce qu'un seul langage et un seul flux de travail couvrent des milliers de providers : AWS, Azure, Google Cloud, Cloudflare, Kubernetes, GitHub, des services de supervision et bien d'autres. Le 10 août 2023, HashiCorp a fait passer Terraform de la licence open source Mozilla Public License 2.0 à la Business Source License 1.1, qui autorise l'usage interne et commercial mais interdit de le proposer dans des produits concurrents de ceux de HashiCorp. La version 1.5.7 a été la dernière publiée sous l'ancienne licence.

La communauté a répondu par un fork. Un manifeste appelant à un retour à l'open source est paru en quelques jours ; en septembre 2023, la Linux Foundation a accepté le fork sous le nom d'OpenTofu ; sa première version stable, la 1.6.0, a suivi en janvier 2024 ; et en 2025, il a été accepté à la CNCF comme projet sandbox. IBM a finalisé son acquisition de HashiCorp en février 2025. Les deux outils restent largement compatibles par le langage et les providers, mais leurs fonctionnalités divergent : OpenTofu a par exemple ajouté le chiffrement de l'état côté client, tandis que la plateforme commerciale de Terraform a ajouté ses propres capacités. Pour un architecte, l'épisode rappelle que la licence et la gouvernance d'un outil font partie de son profil de risque, et que des standards ouverts et une alternative crédible gardent ce risque maîtrisable.

Parmi les outils propres à un seul cloud, AWS CloudFormation (2011) et Bicep de Microsoft (introduit en 2020, compilé en modèles Azure Resource Manager) sont exécutés par le fournisseur, qui tient le registre de ce qui existe : il n'y a donc pas de fichier d'état à gérer, au prix d'une dépendance à un seul cloud. Pulumi (2018) et l'AWS CDK permettent aux développeurs d'écrire l'infrastructure dans des langages généralistes ; Crossplane étend Kubernetes pour gérer les ressources cloud par réconciliation. Le bon choix dépend des clouds que vous utilisez, des compétences de votre équipe, de la maturité des providers dont vous avez besoin, de la politique en matière de licences et de ce que le reste de l'organisation utilise déjà ; la cohérence entre les équipes vaut généralement plus que les mérites marginaux d'un outil donné.

```
terraform {
  required_version = "≥ 1.6"
  backend "gcs" {                 # remote state; this backend locks natively
    bucket = "fd-infra-state"
    prefix = "orders/prod"
  }
}

variable "environment" { type = string }

module "orders_db" {
  source      = "git::https://git.example.com/modules/postgres.git?ref=v3.2.0"
  name        = "orders-${var.environment}"
  size        = var.environment == "prod" ? "large" : "small"
  high_avail  = var.environment == "prod"   # standby in a second zone
  backup_days = 14
}

output "orders_db_host" {
  value = module.orders_db.host
}
```

OpenTofu ou Terraform (HCL) : état distant et module interne versionné qui masque les détails du provider derrière une interface réduite.

Exécuter `tofu plan` (ou `terraform plan`) montre ce qui changerait avant que quoi que ce soit ne se produise. Le plan est l'artefact le plus important du flux de travail : les relecteurs le lisent dans la pull request, à côté du code, et c'est lui qui est appliqué.

```
  # module.orders_db.google_sql_database_instance.this will be updated in-place
  ~ resource "google_sql_database_instance" "this" {
      ~ settings {
          ~ tier = "db-custom-2-8192" → "db-custom-4-16384"
        }
    }

Plan: 0 to add, 1 to change, 0 to destroy.
```

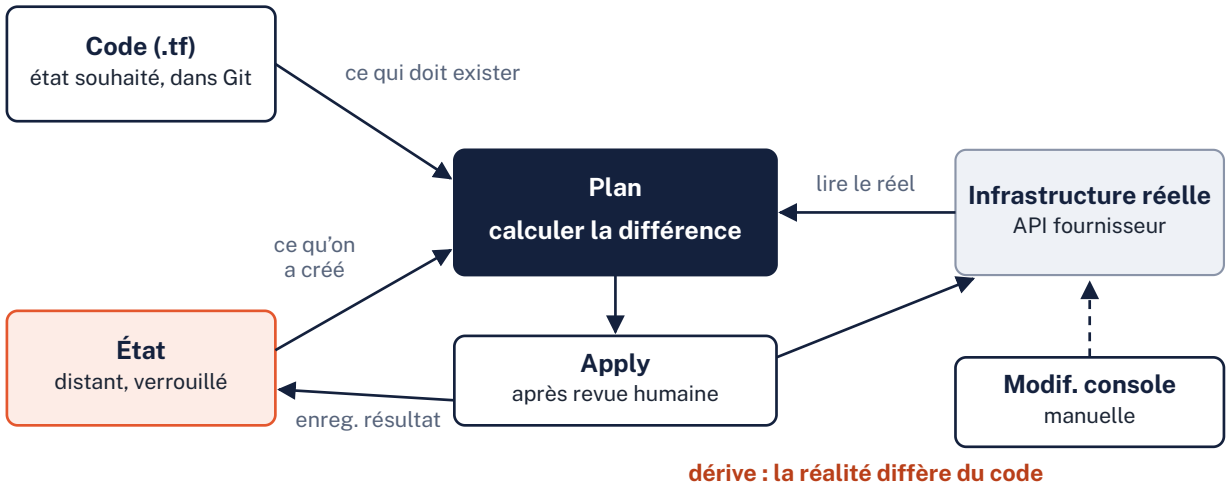
Un plan abrégé pour redimensionner la base de données de production des commandes. Le tilde signale une mise à jour sur place ; un -/+ signifierait détruire et recréer, ce qui, pour une base de données, est un signal d'alarme.

État, verrouillage et dérive

Les outils déclaratifs doivent savoir quelles ressources réelles correspondent à quelles définitions. Terraform et OpenTofu conservent cette information dans l'état : un fichier qui associe chaque ressource du code à l'identifiant de l'objet réel, avec ses derniers attributs

connus. La planification compare trois choses : le code (ce qui devrait exister), l'état (ce que l'outil a créé) et l'infrastructure réelle, lue via l'API du fournisseur (ce qui existe vraiment).

Plan, apply, état et dérive



Un plan compare le code, l'état et la réalité ; une modification manuelle dans la console fait diverger la réalité des deux, et le plan suivant tentera de l'annuler.

L'état demande du soin parce qu'il est à la fois critique et sensible. Il doit être stocké à distance (dans un stockage objet ou un service managé), pas sur un portable, afin que toute l'équipe et le pipeline partagent une seule version. Il doit être *verrouillé* pendant les modifications, pour que deux apply ne puissent pas s'entremêler et le corrompre. Il doit être sauvegardé et versionné, car un fichier d'état perdu laisse l'outil dans l'ignorance des ressources qu'il a créées. Et c'est un secret : l'état contient souvent des mots de passe de bases de données et des clés en clair, donc son accès doit être restreint, et il doit être chiffré au repos (OpenTofu peut aussi le chiffrer côté client).

La dérive de configuration (drift) est toute différence entre le code et la réalité, généralement due à des modifications manuelles faites dans l'urgence, à un autre outil qui touche la même ressource, ou à un échec partiel pendant un apply. La dérive est dangereuse, car le prochain apply peut annuler silencieusement un changement dont quelqu'un dépendait, ou échouer en production sur une différence dont personne ne soupçonnait l'existence. Les équipes la détectent en exécutant un plan à intervalles réguliers (par exemple chaque nuit) et en alertant sur tout changement proposé, et la préviennent en limitant l'accès en écriture à la console à une procédure d'urgence (break-glass) enregistrée.

L'état définit aussi le *rayon d'impact* (blast radius). Un état géant pour toute une entreprise signifie que chaque plan est lent, que chaque apply met tout en jeu et qu'un seul verrou bloque toutes les équipes. Découper l'état par composant et par environnement (réseau, cluster

partagé, ressources de chaque service ; production séparée du staging) garde les plans rapides et les défaillances contenues, au prix de devoir transmettre des sorties, comme les identifiants réseau, d'un état à l'autre. La refactorisation est prise en charge : les blocs `moved` renomment des ressources sans les détruire, et `import` place sous gestion des ressources existantes créées à la main.

EXEMPLE COMMENTÉ

Exemple commenté : une règle de pare-feu qui a dérivé

Pendant un incident un vendredi soir, un ingénieur ouvre, via la console cloud, le pare-feu de la base de données de production des commandes à l'adresse IP d'un prestataire, pour que celui-ci aide à diagnostiquer une requête lente. Personne ne met à jour le code.

Détection	La tâche nocturne de détection de dérive exécute `tofu plan` pour chaque état de production. Le samedi, elle signale *1 to change* pour le pare-feu de la base des commandes et publie le plan dans le canal de l'équipe plateforme.
Diagnostic	Le plan montre une adresse source autorisée en plus. Le journal d'audit du cloud indique qui l'a ajoutée, quand et dans le cadre de quel incident.
Décision	L'accès était temporaire et l'incident est clos : c'est donc le code qui a raison et la réalité qui a tort. Appliquer le plan supprime la règle. Si le changement avait été nécessaire de façon permanente, la correction aurait consisté à l'ajouter au code par une pull request relue.
Prévention	L'accès en écriture à la console de production est remplacé par un rôle d'urgence (break-glass) qui exige une justification et déclenche une alerte à chaque utilisation. Une règle de politique (voir plus bas) rejette toute règle de pare-feu de base de données ouverte à tout l'internet.
Ce que cela montre	La dérive n'est pas qu'une question de rangement : un trou non documenté dans un pare-feu est une exposition de sécurité, et sans détection, le prochain apply sans rapport l'aurait supprimé sans que personne sache pourquoi le prestataire avait perdu son accès.

La règle empirique : la réalité peut différer du code pendant quelques minutes lors d'une urgence, mais la différence doit être soit codifiée, soit annulée au plus tard le jour ouvré suivant.

SI VOUS DÉBUTEZ

Pas à pas : pourquoi l'outil tient un registre

Le point qui fait le plus souvent trébucher ici est la raison pour laquelle des outils comme OpenTofu et Terraform conservent un état. Le code dit ce qui devrait exister, mais pas à quelle ressource réelle du compte cloud il fait référence. Suivez la base de données de production des commandes tout au long de sa vie.

- L'équipe décrit orders-prod dans le code, à l'aide du module interne de base de données. Rien n'existe encore : l'état est donc vide, et le premier plan propose de l'ajouter.
- Lors de l'apply, l'outil demande au fournisseur de créer la base de données. Le fournisseur lui attribue un identifiant, et l'outil enregistre dans l'état que cette ressource du code est la base de données réelle portant cet identifiant, avec ses paramètres.
- Des mois plus tard, l'équipe modifie la taille dans le code. Le plan lit le code, trouve dans l'état de quelle base de données réelle il s'agit, demande au fournisseur ses paramètres actuels et propose une seule modification sur place.
- Sans l'état, l'outil n'aurait aucune trace du fait que cette base de données appartient à ce code, et proposerait de la créer à nouveau.
- L'état révèle aussi ce que le code ne mentionne plus. Si l'équipe supprime le module du code, le plan propose de détruire la base de données que l'état répertorie encore.
- C'est pourquoi l'état est partagé, verrouillé, sauvegardé et gardé confidentiel : deux apply simultanés pourraient le corrompre, le perdre laisserait l'outil dans l'ignorance de ce qu'il a créé, et il contient souvent des mots de passe.
- **La leçon :** le code dit ce qui devrait exister, l'infrastructure réelle est ce qui existe, et l'état relie les deux. Lisez chaque plan en gardant les trois à l'esprit, et traitez un plan qui détruit une base de données comme un signal d'alerte.

Modules et environnements

Un module est une unité réutilisable de code d'infrastructure, avec des entrées et des sorties : une base de données PostgreSQL avec sauvegardes et supervision, le réseau et l'équilibreur de charge d'un service, un namespace Kubernetes standard avec ses quotas et ses politiques. Un bon module applique le masquage de l'information du chapitre 3 : une interface réduite (nom, taille, haute disponibilité ou non) devant les nombreux détails propres au fournisseur qu'une équipe de service ne devrait pas avoir à connaître. Les modules sont versionnés et publiés comme des bibliothèques, et les environnements font référence à une version précise, comme le `ref=v3.2.0` de l'exemple, afin qu'un changement de module n'atteigne la production que lorsque quelqu'un en fait délibérément la mise à niveau.

Les environnements doivent être construits à partir des mêmes modules avec des entrées différentes, et non à partir de code copié. Une organisation courante consiste en un répertoire (ou un dépôt) par environnement et par composant, chacun avec son propre état, qui appelle des modules partagés et versionnés ; un changement est promu en mettant à jour la version du module ou les entrées en staging, puis en production. Les environnements copiés-collés divergent en quelques mois, jusqu'à ce que le staging ne permette plus de prédire la production. Les workspaces, qui donnent plusieurs états à une même configuration, conviennent mieux aux copies éphémères, comme les environnements de test par branche, qu'au staging et à la production, qui durent.

Méfiez-vous de la sur-abstraction. Un module avec quarante entrées booléennes qui essaie de servir toutes les équipes est aussi difficile à modifier que le code qu'il a remplacé. Préférez quelques modules aux choix affirmés qui encodent les standards de l'organisation : c'est exactement ce qu'une équipe plateforme propose sous forme de golden path à la section 10.7.

Infrastructure immuable ou gestion de configuration

Les outils de gestion de configuration maintiennent des serveurs à longue durée de vie dans un état décrit : Ansible se connecte par SSH et pousse les changements sans agent ; Chef et Puppet font tourner des agents qui récupèrent leur configuration et y convergent périodiquement. Ils fonctionnent, mais chaque serveur accumule un historique : paquets installés pendant un incident, fichiers modifiés à la main, mises à jour appliquées à moitié. Martin Fowler a appelé le résultat un *snowflake server* (serveur flocon de neige), unique et impossible à reproduire, et a proposé comme remède le *phoenix server* (serveur phénix), régulièrement brûlé et reconstruit de zéro.

L'infrastructure immuable, terme popularisé par Chad Fowler en 2013, pousse ce remède jusqu'au bout : ne jamais modifier un serveur en fonctionnement. Chaque changement, qu'il s'agisse d'une mise en production applicative ou d'un correctif du système d'exploitation, passe par la construction d'une nouvelle image (une image de conteneur, ou une image de machine avec un outil comme Packer), son test et le remplacement des anciennes instances, généralement par un déploiement progressif ou bleu-vert. Les conteneurs sur Kubernetes sont immuables par défaut. Les avantages sont l'absence de dérive, des instances identiques, des environnements trivialement reproductibles et un retour arrière par redéploiement de l'image précédente ; les coûts sont un pipeline d'images, le besoin de processus jetables et sans état (section 10.3) et un délai plus long pour un changement urgent isolé.

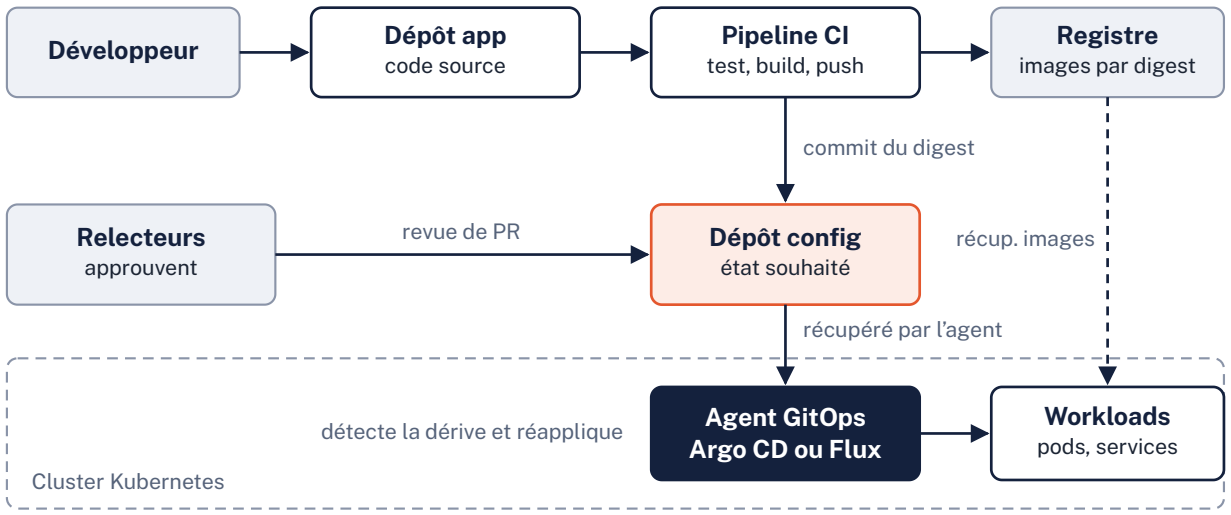
La gestion de configuration n'a pas disparu. Elle reste le bon outil pour les parcs qui ne peuvent pas être remplacés en bloc : serveurs sur site, équipements réseau, bases de données sur du matériel dédié, et les images de base elles-mêmes, souvent construites en exécutant Ansible

pendant la construction de l'image. La règle d'architecture est de repousser la mutabilité aussi bas et aussi tôt que possible : configurer pendant la construction des images, puis traiter ce qui tourne comme étant en lecture seule.

GitOps

GitOps applique la boucle de réconciliation de la section 10.2 à des environnements entiers. Le terme a été forgé en 2017 par Alexis Richardson, de Weaveworks, l'entreprise à l'origine de l'outil Flux. Le projet OpenGitOps, un groupe de travail de la CNCF, définit quatre principes : l'état souhaité est *déclaratif* ; il est *versionné et immuable*, avec un historique complet ; il est *récupéré automatiquement* par des agents logiciels ; et il est *réconcilié en continu*, des agents observant l'état réel et appliquant l'état souhaité. Les deux principaux agents, Argo CD et Flux, sont tous deux des projets diplômés (graduated) de la CNCF ; Flux a continué comme projet communautaire après la fermeture de Weaveworks en 2024.

Un flux de livraison GitOps



La CI construit et publie les images ; Git contient l'état souhaité ; un agent dans le cluster le récupère, l'applique et corrige sans cesse la dérive.

Le modèle pull a un avantage de sécurité sur un pipeline qui pousse les changements dans le cluster : le système de CI n'a besoin d'aucun identifiant pour la production, car l'agent à l'intérieur du cluster va chercher ce qu'il doit exécuter. Git devient le journal d'audit et l'interface du changement, y compris pour l'exploitation : mettre un service à l'échelle ou revenir en arrière est un commit relu ou un revert. La dérive est corrigée automatiquement, si bien qu'un `kubectl edit` manuel ne dure que jusqu'à la synchronisation suivante.

GitOps soulève ses propres questions de conception. Les secrets ne doivent pas se trouver en clair dans Git ; les équipes les chiffrent (SOPS, Sealed Secrets) ou stockent des références que l'External Secrets Operator résout à partir d'un gestionnaire de secrets. La promotion entre environnements demande une convention, par exemple un répertoire par environnement, avec des pull requests qui copient un nouveau digest d'image du staging vers la production. La correction automatique de la dérive doit être délibérément suspendue lors de certaines urgences, et la structure du dépôt doit refléter la responsabilité, afin que les équipes puissent modifier leurs propres services sans toucher à ceux des autres. GitOps est le plus naturel pour Kubernetes, mais les mêmes principes s'appliquent à l'infrastructure cloud via Crossplane ou via des pipelines qui planifient et appliquent le code d'infrastructure à partir de commits fusionnés.

Politique en tant que code (policy as code)

La politique en tant que code (policy as code) exprime les règles sur l'infrastructure sous forme de code vérifiable par machine au lieu d'une page de wiki : les buckets de stockage ne doivent pas être publics, les bases de données doivent être chiffrées et sauvegardées, chaque ressource doit porter une étiquette de responsable et de centre de coûts, seules les régions approuvées peuvent être utilisées, les conteneurs ne doivent pas s'exécuter en root, les images de production doivent provenir du registre de l'entreprise. Les vérifications s'exécutent à plusieurs moments : sur le plan dans la pull request (Open Policy Agent avec Conftest, Checkov, HashiCorp Sentinel dans ses produits commerciaux), à l'admission dans un cluster Kubernetes (OPA Gatekeeper, Kyverno) et en continu sur les ressources en fonctionnement (services de politiques des fournisseurs cloud).

La politique en tant que code fait passer la gouvernance d'une réunion de revue à un garde-fou rapide et cohérent qui s'explique de lui-même dans la pull request. Elle fonctionne au mieux quand les politiques sont peu nombreuses, clairement justifiées, versionnées et testées comme n'importe quel code, quand elles commencent en mode avertissement avant de bloquer, et quand il existe un processus d'exception documenté. Le chapitre 11 explique comment ces vérifications automatisées s'intègrent à la gouvernance de l'architecture (la gouvernance dans les organisations agiles).

Tester le code d'infrastructure, du plus rapide au plus lent

1. **Vérifications statiques à chaque commit**
   Le formatage, `validate`, le linting (par exemple TFLint) et les scanners de sécurité détectent en quelques secondes les erreurs de syntaxe, les arguments obsolètes et les mauvaises configurations évidentes.

2. **Vérifications de politiques sur le plan**
   Produisez le plan en JSON et évaluez-le au regard des politiques de l'organisation ; les relecteurs voient les violations dans la pull request.

3. **Tests de modules**
   `terraform test` et `tofu test` exécutent des assertions sur un module, avec des providers simulés pour des tests rapides de type unitaire, ou réels pour des tests d'intégration.

4. **Tests d'intégration dans des environnements éphémères**
   Appliquez le module ou l'environnement dans un compte jetable, vérifiez qu'il fonctionne (une base de données accepte des connexions, un service répond), puis détruisez-le. Des outils comme Terratest automatisent cela ; c'est lent et cela coûte de l'argent, donc réservez-le aux modules partagés.

5. **Revue du plan et déploiement par étapes**
   Appliquez d'abord en staging, puis en production, à partir des mêmes versions ; une personne lit chaque plan de production, en particulier les destructions et les remplacements.

6. **Détection de la dérive et exercices de restauration**
   Des plans exécutés à intervalles réguliers détectent la dérive ; reconstruire périodiquement un environnement à partir du code prouve que c'est le code, et non la mémoire, qui fait foi.

Liste de contrôle : un code d'infrastructure prêt pour la production

Utilisez-la pour relire le dépôt d'infrastructure d'une équipe ou un nouveau module de plateforme.

- [ ] **Tout est dans le code**
  Aucune ressource de production n'existe simplement parce que quelqu'un a cliqué ; les ressources existantes créées à la main ont été importées.
- [ ] **État distant, verrouillé et chiffré**
  Backend partagé avec verrouillage, gestion de versions et accès restreint ; état découpé par composant et par environnement pour limiter le rayon d'impact.
- [ ] **Plans relus**
  Chaque changement est une pull request qui montre le plan ; les destructions et remplacements de ressources avec état exigent une approbation explicite.
- [ ] **Modules versionnés**
  Les environnements épinglent les versions des modules ; le staging et la production utilisent les mêmes modules avec des entrées différentes, pas des copies.
- [ ] **Politiques appliquées automatiquement**
  Les règles de chiffrement, d'exposition réseau, d'étiquetage et de région sont vérifiées sur chaque plan et à l'admission dans le cluster.
- [ ] **Secrets tenus hors du code et de Git**
  Les identifiants proviennent d'un gestionnaire de secrets ou de l'identité de charge de travail ; l'accès à l'état est traité comme un accès à des secrets.
- [ ] **Tests dans le pipeline**
  Vérifications statiques et vérifications de politiques à chaque changement ; tests de modules pour les modules partagés ; tests d'intégration pour les modules critiques.
- [ ] **Dérive détectée et traitée**
  Des plans exécutés à intervalles réguliers alertent sur les différences ; l'accès en écriture à la console est une procédure d'urgence (break-glass) enregistrée.
- [ ] **Immuable quand c'est possible**
  Les serveurs et les conteneurs sont remplacés à partir d'images au lieu d'être corrigés sur place ; la configuration se fait au moment de la construction de l'image.
- [ ] **Reprise testée en exercice**
  Un environnement a été reconstruit à partir du code dans le délai de reprise convenu, restauration des données comprise (voir la section 10.6).

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Pourquoi l'idempotence est-elle importante pour le code d'infrastructure ?**

Parce que l'automatisation réexécute souvent le code. Un outil déclaratif et idempotent ne change rien quand la réalité correspond déjà au code, donc le réexécuter est sans risque ; un script non idempotent risque de tenter de tout recréer.

▼ **Citez trois choses qu'un plan compare, et ce que chacune représente.**

Le code (ce qui devrait exister), l'état (ce que l'outil a créé et suit) et l'infrastructure réelle lue via l'API du fournisseur (ce qui existe vraiment).

▼ **Pourquoi l'état ne devrait-il pas être conservé dans un seul fichier pour toute une organisation ?**

Un seul état signifie des plans lents, un verrou unique qui bloque toutes les équipes et un rayon d'impact qui couvre tout ; le découper par composant et par environnement garde les changements rapides et les défaillances contenues.

▼ **Quel avantage de sécurité le modèle pull de GitOps a-t-il sur un pipeline qui pousse vers le cluster ?**

Le système de CI n'a besoin d'aucun identifiant de production, car un agent à l'intérieur du cluster récupère l'état souhaité dans Git et l'applique.

▼ **Quand la gestion de configuration reste-t-elle un bon choix ?**

Pour les machines à longue durée de vie qui ne peuvent pas simplement être remplacées, comme les serveurs sur site et les équipements réseau, et pour configurer les images de base pendant leur construction.

▼ **À vous : Une équipe garde son fichier d'état d'infrastructure sur l'ordinateur portable d'un ingénieur, avec tous les environnements dans un seul état, et a créé le staging en copiant le code de production et en le modifiant. Énumérez les risques et dites comment vous restructureriez cela.**

L'état pourrait être perdu, personne d'autre ne peut appliquer des changements sans risque, et deux personnes pourraient entrer en collision. Un état unique signifie aussi qu'une seule erreur peut endommager tous les environnements. Le code copié va dériver. Déplacez l'état vers un magasin distant, verrouillé, versionné et à accès contrôlé, découpé par environnement et par composant pour limiter le rayon d'impact. Construisez des modules versionnés à interfaces étroites, et utilisez-les pour chaque environnement avec des entrées différentes. Ajoutez des plans programmés pour détecter la dérive.

SOURCES

- OpenTofu: https://opentofu.org/
- Terraform documentation: State: https://developer.hashicorp.com/terraform/language/state
- Terraform documentation: Tests: https://developer.hashicorp.com/terraform/language/tests
- OpenGitOps: GitOps principles: https://opengitops.dev/
- Martin Fowler: PhoenixServer: https://martinfowler.com/bliki/PhoenixServer.html

SECTION 10.5 SUR 10.8 · INTERMÉDIAIRE · 25 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Pipelines CI/CD et stratégies de déploiement

AVANT DE LIRE

Cette section s'appuie sur :

- 0.8 Comment les logiciels sont construits
- 2.6 Modifiabilité et déployabilité
- 7.7 Évolution des schémas

Termes déjà vus qu'elle utilise : Image de conteneur, Expansion-contraction (expand and contract), Objectif de niveau de service (SLO), Monolithe distribué

L'IDÉE EN BREF

Pensez à une chaîne de boulangeries qui lance un nouveau pain. Elle cuit une fournée d'essai, la vérifie dans la cuisine de la boulangerie, puis la vend dans une seule boutique pendant une matinée avant de l'envoyer dans toutes les boutiques. Si des clients se plaignent, seuls les clients d'une boutique ont été touchés, et l'ancien pain est toujours en rayon. Un pipeline logiciel suit la même routine, automatisée : chaque changement est construit une seule fois, vérifié par étapes, essayé sur quelques utilisateurs, et diffusé plus largement seulement quand les éléments recueillis montrent qu'il est bon. Un interrupteur distinct décide du moment où les clients voient réellement une nouvelle fonctionnalité.

CE QU'IL FAUT SAVOIR

- Un pipeline de déploiement (Humble et Farley, 2010) fait passer chaque commit par des étapes automatisées, les plus rapides d'abord, et un échec à n'importe quelle étape arrête ce changement et en avertit l'auteur.
- Build once, deploy many : empaquetez chaque changement dans un artefact unique et immuable, stockez-le dans un dépôt d'artefacts et promouvez ce même artefact dans chaque environnement, en ne changeant que la configuration.
- Le développement basé sur le tronc garde des branches de courte durée et des changements petits, ce qui rend l'intégration continue réelle et chaque changement plus facile à relire, tester et annuler.
- Les déploiements par recréation, progressifs, bleu-vert, canari et fantômes échangent une infrastructure et une complexité supplémentaires contre une moindre exposition des utilisateurs et un retour arrière plus rapide ; tous, sauf la recréation, exigent que l'ancienne et la nouvelle version soient compatibles.

- Les feature flags séparent le déploiement de la mise en production ; les catégories de Hodgson sont les toggles de mise en production, d'expérimentation, opérationnels et de permission, et chaque flag a un coût tant qu'il n'est pas supprimé.
- Les changements de base de données doivent être rétrocompatibles (expansion-contraction) et s'exécuter comme des étapes distinctes du pipeline, car les données ne peuvent pas faire l'objet d'un retour arrière comme le code.
- La livraison progressive automatise les décisions sur le canari en le comparant statistiquement à une référence neuve ; face à une mauvaise version, la réponse par défaut est de revenir en arrière d'abord et de diagnostiquer ensuite.

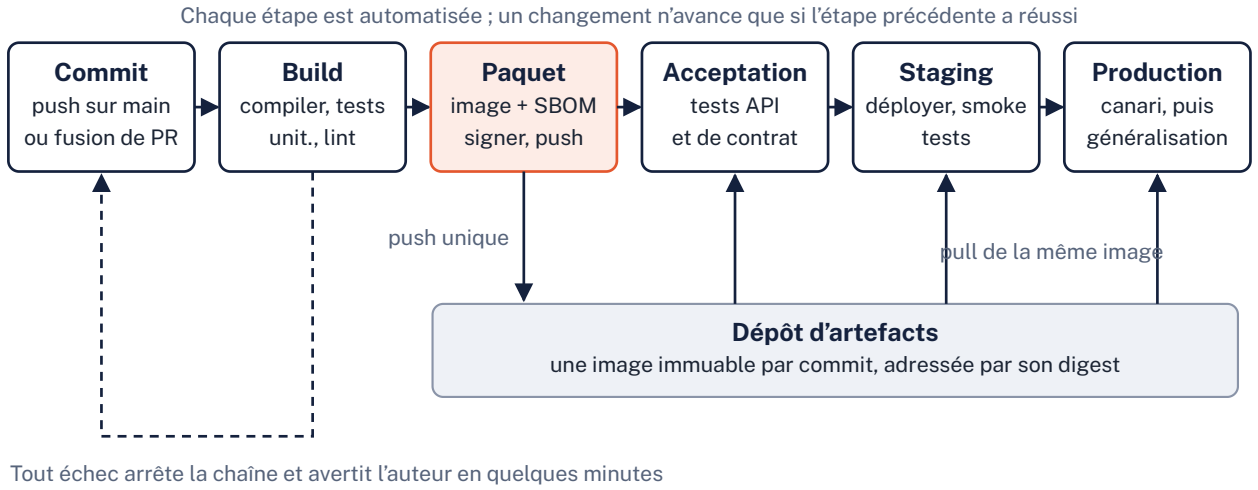
Du commit à la production : à quoi sert un pipeline

La section 0.8 (comment les logiciels sont construits) a présenté l'intégration continue (chaque changement est fusionné et vérifié automatiquement), la livraison continue (la branche principale peut toujours être mise en production) et le déploiement continu (chaque changement qui passe les contrôles va en production sans décision humaine). Cette section les traite comme de l'architecture : la conception du chemin que suit un changement, et les stratégies qui l'amènent en production sans nuire aux utilisateurs.

L'idée centrale est le pipeline de déploiement, décrit par Jez Humble et David Farley dans *Continuous Delivery* (2010). Chaque commit déclenche une séquence automatisée d'étapes. Les premières étapes sont rapides et peu coûteuses, et détectent la plupart des problèmes ; les suivantes sont plus lentes, plus réalistes et plus coûteuses. Un changement ne passe à l'étape suivante que si la précédente a réussi : plus un build avance, plus il a gagné la confiance. Un échec, où qu'il se produise, arrête ce build et revient directement à la personne qui a fait le changement, pendant qu'elle l'a encore bien en tête.

Le pipeline compte pour les architectes pour trois raisons. D'abord, c'est là que la déployabilité, un attribut de qualité vu en section 2.6, est réellement obtenue ou perdue. Ensuite, sa vitesse limite la rapidité avec laquelle l'organisation peut réagir à une évolution du marché ou à une vulnérabilité de sécurité : si un correctif d'une ligne met deux jours à atteindre la production, l'architecture a deux jours de retard, quel que soit l'aspect du code. Enfin, la forme du système contraint la forme du pipeline : des services déployables indépendamment peuvent avoir des pipelines indépendants, mais un monolithe distribué impose des mises en production coordonnées, aussi astucieux que soient les outils.

Les étapes d'un pipeline typique



Un artefact est construit et poussé une seule fois ; chaque étape suivante déploie exactement cet artefact, et tout échec arrête la chaîne.

Ce que fait chaque étape et combien de temps elle devrait prendre

1. **Étape de commit (objectif : moins de 10 minutes)**
   Compiler, exécuter les tests unitaires, les linters, l'analyse statique et l'analyse des vulnérabilités des dépendances. C'est l'étape que les développeurs attendent, elle doit donc être rapide ; les tests lents viennent plus tard.

2. **Empaquetage**
   Construire l'artefact déployable (généralement une image de conteneur), générer une nomenclature logicielle (SBOM), la signer et la pousser vers le dépôt d'artefacts. Les contrôles de la chaîne d'approvisionnement sont traités en section 8.6.

3. **Acceptation et intégration**
   Déployer l'artefact dans un environnement de test temporaire ou partagé et exécuter des tests d'API, des tests de contrat pilotés par le consommateur et quelques parcours de bout en bout. Les tests de performances et de sécurité peuvent s'exécuter ici en parallèle ou selon un calendrier.

4. **Staging**
   Déployer dans un environnement configuré comme la production, appliquer les migrations de base de données, exécuter les smoke tests. Sa valeur dépend de sa ressemblance avec la production en matière de configuration, de forme des données et de dépendances.

5 **Production**
Déployer avec une stratégie qui limite l'exposition (progressive, bleu-vert ou canari), vérifier avec de vraies métriques par rapport aux objectifs de niveau de service, puis terminer ou revenir en arrière automatiquement.

Build once, deploy many, et le dépôt d'artefacts

Une règle de *Continuous Delivery* encore enfreinte étonnamment souvent est build once, deploy many (construire une fois, déployer partout) : compiler et empaqueter chaque changement exactement une fois, puis promouvoir ce même artefact dans chaque environnement. Si une équipe reconstruit pour le staging, puis à nouveau pour la production, ce qu'elle a testé n'est pas ce qu'elle a livré. Un correctif différent du compilateur, une dépendance résolue pendant la nuit vers une version plus récente ou une image de base modifiée peuvent faire diverger les deux builds, et les tests ne prouvent rien sur le second.

Ce qui diffère d'un environnement à l'autre, c'est la configuration, fournie au moment du déploiement ou du démarrage (adresses de bases de données, identifiants, réglages des feature flags), jamais intégrée à l'artefact. C'est le facteur de configuration de l'application douze facteurs. Une image de conteneur étiquetée `orders:4f1c2ab` (le hash du commit) est déployée en test, en staging et en production ; seuls ses variables d'environnement et ses secrets montés changent.

Le dépôt d'artefacts est le stockage entre le build et le déploiement : un registre de conteneurs (Amazon ECR, Azure Container Registry, Google Artifact Registry, GitHub Container Registry, Harbor) ou un dépôt universel comme JFrog Artifactory ou Sonatype Nexus pour les paquets et les binaires. Traitez-le comme une infrastructure critique. Déployez par digest immuable (`sha256:...`) plutôt que par un tag modifiable comme `latest`, qui peut être déplacé pour pointer vers une autre image ; conservez les artefacts qui sont ou ont été en production aussi longtemps que vous pourriez avoir besoin d'y revenir ou de les examiner ; et enregistrez la provenance, c'est-à-dire le commit, l'exécution du pipeline et les dépendances qui ont produit chaque artefact.

Développement basé sur le tronc et petits lots

La façon dont le code est réparti en branches détermine la fréquence à laquelle il est réellement intégré. Dans le développement basé sur le tronc, chacun intègre de petits changements dans une seule branche principale (le tronc) au moins une fois par jour, soit en faisant des commits directement, soit par des branches de courte durée et des pull requests qui vivent quelques heures, pas des semaines. Le programme de recherche DORA a associé à

plusieurs reprises les branches de courte durée et les fusions fréquentes à une meilleure performance de livraison, et c'est une condition préalable à l'intégration continue au sens littéral : du code qui reste trois semaines sur une branche n'est pas intégré en continu.

Les branches de fonctionnalité de longue durée semblent plus sûres, parce que le travail inachevé est tenu à l'écart, mais le risque n'est que reporté : plus deux branches divergent longtemps, plus la fusion finale est grosse et risquée, et plus longtemps le pipeline voit du code qui n'est pas celui qui sera livré. GitFlow, un modèle de branches avec des branches develop et release de longue durée publié par Vincent Driessen en 2010, a été largement adopté ; en 2020, Driessen a ajouté une note indiquant que les équipes qui livrent en continu, comme pour les applications web, devraient envisager un flux de travail plus simple. Les équipes qui doivent travailler sur de gros changements dans le tronc utilisent deux techniques : les feature flags, qui masquent un comportement inachevé, et le branch by abstraction, qui introduit une interface devant l'ancienne implémentation, construit la nouvelle derrière elle et bascule quand elle est prête.

Les praticiens sont réellement en désaccord à la marge. Les projets open source qui ont de nombreux contributeurs non fiables s'appuient sur des pull requests de plus longue durée et sur la relecture, et les équipes réglementées ont parfois besoin de branches de version explicites pour l'audit. Le principe qui survit au désaccord est la taille des lots : plus chaque changement est petit, plus il est facile à relire, tester, déployer, diagnostiquer et annuler.

```
name: delivery
on:
  push:
    branches: [main]
env:
  IMAGE: registry.example.com/food/orders
jobs:
  build:
    runs-on: ubuntu-latest
    outputs:
      digest: ${{ steps.push.outputs.digest }}
    steps:
      - uses: actions/checkout@v4
      - run: make lint test              # fast feedback first
      - id: push
        run: |
          docker build -t "$IMAGE:${{ github.sha }}" .
          docker push "$IMAGE:${{ github.sha }}"
          ref=$(docker inspect --format '{{index .RepoDigests 0}}' "$IMAGE:${{ github.sha
}}")
          echo "digest=${ref#*@}" >> "$GITHUB_OUTPUT"   # the image's sha256 digest
  staging:
    needs: build
    runs-on: ubuntu-latest
    environment: staging
    steps:
      - uses: actions/checkout@v4
      - run: ./deploy.sh staging "$IMAGE@${{ needs.build.outputs.digest }}" && ./smoke-
test.sh staging
  production:
    needs: [build, staging]
    runs-on: ubuntu-latest
    environment: production            # can require an approval
    steps:
      - uses: actions/checkout@v4
      - run: ./deploy.sh production "$IMAGE@${{ needs.build.outputs.digest }}" --canary 5
```

Un flux de livraison continue minimal dans la syntaxe de GitHub Actions : construire et tester une fois, pousser une image étiquetée avec le commit, puis déployer cette image exacte, par son digest, en staging, puis en production comme canari. GitLab CI, Azure Pipelines, Jenkins et d'autres expriment les mêmes étapes. La connexion au registre et les scripts de déploiement sont omis.

Stratégies de déploiement

Une stratégie de déploiement décide comment la nouvelle version remplace l'ancienne en production, et donc combien d'utilisateurs une mauvaise version peut toucher avant d'être arrêtée. Les stratégies diffèrent par l'infrastructure supplémentaire qu'elles exigent, la rapidité

de leur retour arrière et le fait que deux versions tournent ou non en même temps. Chaque fois que deux versions tournent en même temps, elles doivent être compatibles avec les données et les messages l'une de l'autre : c'est le prix de toutes les stratégies sauf la recréation.

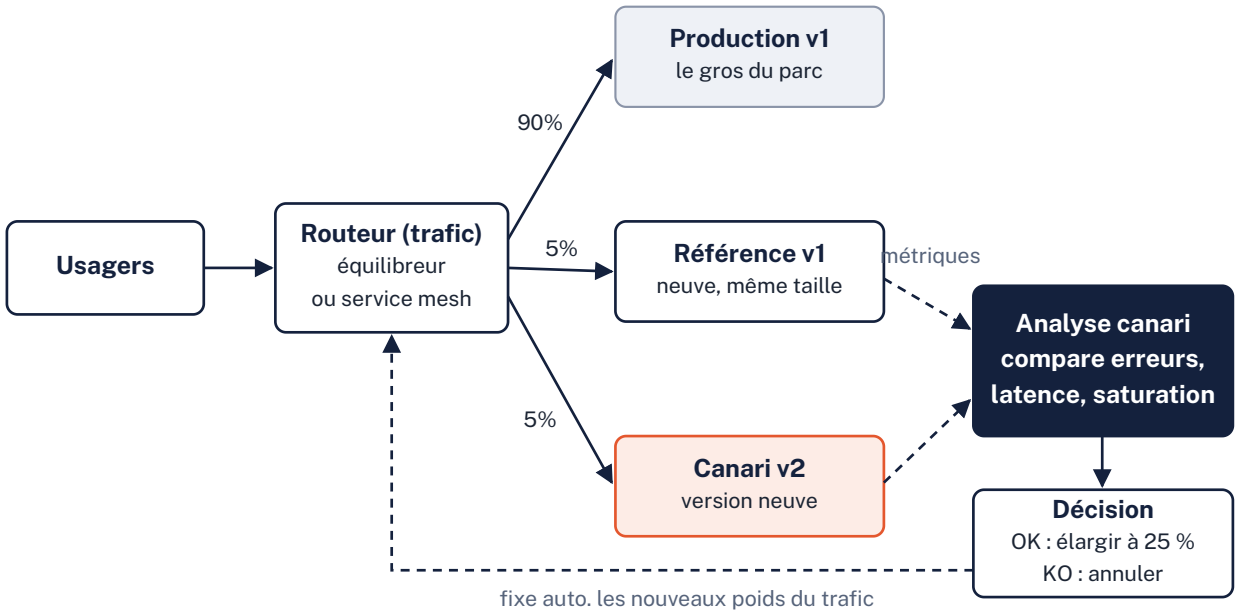
Comparaison des stratégies de déploiement

Stratégie	Comment ça marche	Risque pour les utilisateurs	Retour arrière (rollback)	Coût et complexité supplémentaires
Recréation	Arrêter toutes les anciennes instances, puis démarrer les nouvelles.	Interruption de service pendant la bascule ; chaque utilisateur rencontre une mauvaise version en même temps.	Redéployer l'ancienne version, avec une nouvelle interruption.	Aucun. Acceptable pour les outils internes, les traitements par lots et les fenêtres de maintenance.
Progressif	Remplacer les instances quelques-unes à la fois (le comportement par défaut de Kubernetes pour les Deployments, avec maxSurge et maxUnavailable valant tous deux 25 % par défaut).	L'exposition augmente à mesure que le déploiement avance ; les contrôles de santé détectent les plantages, mais pas les erreurs subtiles.	Revenir en arrière en redéployant progressivement l'ancienne version, ce qui prend autant de temps que le déploiement.	Faible. L'ancienne et la nouvelle version tournent côte à côte, elles doivent donc être compatibles.
Bleu-vert	Faire tourner un nouvel environnement complet (vert) à côté de l'environnement actif (bleu), le tester, puis basculer tout le trafic au niveau du routeur.	Tous les utilisateurs basculent en même temps, mais le nouvel environnement a été vérifié avant la bascule.	Rapide : rebasculer le routeur, tant que le bleu tourne encore.	Jusqu'au double de capacité pendant la bascule ; les bases de données partagées doivent fonctionner avec les deux versions.

Stratégie	Comment ça marche	Risque pour les utilisateurs	Retour arrière (rollback)	Coût et complexité supplémentaires
Canari	Envoyer une petite part du trafic (disons 1 à 5 %) vers la nouvelle version, comparer ses métriques, puis élargir par paliers.	Seule la part des utilisateurs dirigée vers le canari rencontre une mauvaise version, et seulement jusqu'à ce que l'analyse l'arrête.	Rapide : ramener le poids du canari à zéro.	Exige un routage pondéré, de bonnes métriques et assez de trafic pour une comparaison significative.
Fantôme (dark launch)	Copier les vraies requêtes vers la nouvelle version ; ses réponses sont comparées ou jetées, jamais renvoyées.	Aucun directement, à condition que les effets de bord (paiements, e-mails, écritures) soient supprimés ou simulés.	Rien à annuler ; les utilisateurs n'en ont jamais dépendu.	Double charge sur le chemin dupliqué et gestion soigneuse des effets de bord ; idéal pour les chemins de lecture et les réécritures.

Le déploiement bleu-vert a été décrit par Martin Fowler en 2010, qui en attribue le mérite à Dan North et Jez Humble ; le déploiement canari, nommé d'après les canaris que les mineurs emportaient pour détecter le gaz, a été présenté par Danilo Sato sur le même site en 2014. Dans Kubernetes, les mises à jour progressives sont intégrées ; les déploiements bleu-vert et canari viennent généralement d'un contrôleur de livraison progressive (Argo Rollouts, Flagger) ou d'un service mesh et du routage pondéré de la Gateway API. Sur les plateformes managées, les fonctionnalités équivalentes s'appellent traffic splitting (répartition du trafic) ou deployment slots (emplacements de déploiement).

Déploiement canari avec analyse automatisée



Comparer le canari à une référence neuve de l'ancienne version, plutôt qu'au parc qui tourne depuis longtemps, élimine les différences dues uniquement à la durée de fonctionnement et aux caches chauds.

SI VOUS DÉBUTEZ

Pas à pas : une mauvaise version, quatre façons de la déployer

L'idée la plus difficile de cette section est que la stratégie de déploiement décide combien de clients une mauvaise version atteint avant d'être arrêtée, et à quelle vitesse on peut l'annuler. Suivez une version défectueuse du service de commandes à travers quatre stratégies.

- La nouvelle version a un bug subtil : elle ne plante pas, mais facture de mauvais frais de livraison sur certaines commandes. Les contrôles de santé réussissent, car le processus lui-même est en bonne santé.
- Recréation : toutes les anciennes copies s'arrêtent, puis les nouvelles démarrent. Après une courte panne, chaque client rencontre la version défectueuse en même temps. Pour l'annuler, il faut redéployer l'ancienne version, avec une nouvelle panne.
- Déploiement progressif : les copies sont remplacées quelques-unes à la fois. Comme rien ne plante, les contrôles de santé laissent le déploiement continuer jusqu'à ce que toutes les copies soient défectueuses. Revenir en arrière prend autant de temps que déployer.
- Bleu-vert : la nouvelle version tourne comme un second environnement complet et est testée avant que tout le trafic n'y soit basculé. Si les tests ratent le bug, chaque client bascule en même temps, mais l'annulation est rapide : rebasculer le routeur vers l'ancien environnement, qui tourne toujours.
- Canari : 5 % du trafic va vers la nouvelle version, et ses métriques sont comparées à celles d'une copie neuve de l'ancienne version. Si la comparaison inclut un signal métier comme les frais facturés, elle peut repérer la différence, et la part du canari est ramenée à zéro. Seuls les clients de ces 5 % ont rencontré le bug, et seulement jusqu'à ce que l'analyse l'arrête.
- **La leçon :** choisissez une stratégie selon le nombre de clients que vous pouvez vous permettre d'exposer et la vitesse à laquelle vous devez annuler une erreur, et assurez-vous que vos contrôles peuvent voir les défaillances que vous redoutez, pas seulement les plantages.

Séparer le déploiement de la mise en production avec des feature flags

Le **déploiement** installe une nouvelle version du logiciel dans un environnement ; la **mise en production** rend un comportement accessible aux utilisateurs. Les traiter comme un seul événement fait de chaque déploiement un événement métier, avec toute la nervosité que cela entraîne. Les feature flags (feature toggles) les séparent : le nouveau code est déployé désactivé, et le comportement est mis en production plus tard, progressivement et de façon

réversible, en changeant la configuration. L'article de Pete Hodgson *Feature Toggles (aka Feature Flags)* sur martinfowler.com (2017) distingue quatre catégories selon leur durée de vie et le caractère plus ou moins dynamique de leur décision.

Les quatre catégories de feature toggles selon Hodgson

Catégorie	Objectif	Durée de vie typique	Degré de dynamisme
Toggles de mise en production	Livrer dans le tronc des fonctionnalités incomplètes ou pas encore mises en production, désactivées.	De quelques jours à quelques semaines ; à supprimer après la mise en production.	Plutôt statique pour une version donnée.
Toggles d'expérimentation	Tests A/B ou multivariés qui orientent chaque utilisateur vers une cohorte.	Assez longtemps pour obtenir des résultats statistiquement solides, puis supprimés.	À chaque requête, selon la cohorte de l'utilisateur.
Toggles opérationnels	Commandes d'exploitation, comme la désactivation d'un panneau de recommandations coûteux en cas de forte charge (kill switches, ou interrupteurs d'arrêt d'urgence).	Plutôt courte ; quelques kill switches sont conservés à long terme.	Modifiés à l'exécution, souvent pendant un incident.
Toggles de permission	Fonctionnalités réservées à certains utilisateurs : offres premium, bêta-testeurs, personnel interne.	Peuvent vivre des années.	À chaque requête, selon l'utilisateur.

Les flags ont un coût de possession. Chaque flag double les chemins dans le code qu'il protège, ajoute des combinaisons à tester et laisse du code mort une fois sa décision arrêtée. Hodgson recommande de traiter les flags comme un stock : ajouter une tâche de suppression quand un toggle de mise en production est créé, donner aux flags des dates d'expiration, ajouter des tests qui échouent dès qu'un flag a dépassé sa date, et limiter le nombre de flags qui peuvent exister en même temps. La perte de Knight Capital en 2012, décrite en section 1.3, impliquait un flag réaffecté qui a réactivé du vieux code sur un serveur oublié lors d'un déploiement manuel : un rappel de ne jamais réutiliser les noms de flags et de supprimer le code mort derrière les flags dont la décision est arrêtée. À grande échelle, les flags sont servis par un service dédié (des options open source comme Unleash et Flagsmith, des offres commerciales comme LaunchDarkly, ou l'API OpenFeature, indépendante des fournisseurs) afin qu'un changement se propage en quelques secondes et soit tracé.

Les changements de base de données dans le pipeline

Le code peut revenir en arrière en quelques secondes ; les données, non. Un déploiement qui renomme une colonne, puis fait l'objet d'un retour arrière, laisse l'ancien code face à un schéma qu'il ne comprend pas. La réponse, traitée en détail dans l'évolution des schémas, consiste à rendre chaque changement de schéma rétrocompatible avec la version du code en cours d'exécution, au moyen de l'expansion-contraction : ajouter d'abord la nouvelle structure, déployer du code qui écrit dans les deux et lit l'une ou l'autre, faire le backfill, basculer les lecteurs, et ne supprimer l'ancienne structure que dans une version ultérieure, une fois que plus rien ne l'utilise.

Dans le pipeline, cela signifie que les migrations sont des artefacts versionnés qui s'exécutent comme une étape à part, avant le déploiement du code qui en a besoin, et qu'elles sont testées sur une copie des données de la taille de la production, afin qu'un verrou sur une grande table soit découvert en staging plutôt qu'à l'heure du déjeuner. Les étapes destructrices (suppression de colonnes ou de tables) sont différées et exécutées séparément, car c'est le seul type de changement qu'un retour arrière ne peut pas annuler. Un contrôle utile dans le pipeline fait échouer le build quand une migration n'est pas additive, sauf si elle est explicitement marquée comme une étape de contraction.

Livraison progressive et analyse canari automatisée

La livraison progressive, un terme popularisé en 2018 par James Governor, du cabinet d'analystes RedMonk, combine les canaris, les feature flags et la vérification automatisée : un changement atteint des publics de plus en plus larges, et à chaque étape, c'est le système, et non une personne qui surveille des tableaux de bord, qui décide de continuer ou non. La décision repose sur l'analyse canari automatisée : les métriques du canari (taux d'erreur, percentiles de latence, saturation, signaux métier comme les commandes passées) sont comparées statistiquement à celles d'une référence. Netflix et Google ont publié en open source leur moteur d'analyse, Kayenta, en 2018 ; Argo Rollouts et Flagger fournissent des étapes d'analyse similaires dans Kubernetes.

Trois détails décident si l'on peut se fier à l'analyse. Comparez ce qui est comparable : une référence neuve de l'ancienne version recevant la même part de trafic évite d'imputer à la nouvelle version des différences dues à des caches froids ou à des redémarrages récents. Ayez assez de trafic : à 5 % d'un service qui traite dix requêtes par minute, une heure de canari voit trente requêtes, trop peu pour détecter un taux d'erreur de 1 % ; les services à faible trafic ont donc besoin de paliers plus longs, d'une charge synthétique ou d'une autre stratégie. Choisissez des métriques liées aux objectifs de niveau de service et aux résultats métier, car une version peut être rapide et sans erreur tout en facturant silencieusement de mauvais frais de livraison.

Retour arrière ou roll forward

Quand une mise en production tourne mal, il y a deux façons d'en sortir. Le retour arrière (rollback) revient à l'artefact précédent, ce qui est rapide et ne demande aucun nouveau code, à condition que la version précédente fonctionne toujours avec les données actuelles et avec les versions des autres services. Le roll forward (correction vers l'avant) fait passer un correctif par le pipeline, ce qui est plus lent mais parfois la seule option, par exemple après une migration de données irréversible ou un changement dont d'autres services dépendent déjà.

Une bonne règle par défaut est «revenir en arrière d'abord, diagnostiquer ensuite» : rétablir le service avec l'action sûre la plus rapide, puis enquêter sans pression. Corriger vers l'avant sous pression tend à contourner les contrôles du pipeline, et un correctif précipité sur un incident de production est une source classique de l'incident suivant. Le retour arrière n'est une vraie option que s'il est prévu dès la conception : artefacts précédents conservés, changements de schéma compatibles dans les deux sens pendant au moins une version, messages lisibles par les anciens comme par les nouveaux consommateurs, et chemin de retour arrière lui-même exercé régulièrement. Une équipe qui n'est jamais revenue en arrière devrait supposer qu'elle en est incapable.

EXEMPLE COMMENTÉ

Exemple commenté : mettre en production une nouvelle règle de frais de livraison

Le service de commandes de la plateforme de livraison de repas change sa façon de calculer les frais de livraison, en utilisant la distance et la demande au lieu d'un forfait. L'équipe pratique le développement basé sur le tronc, un déploiement canari et un flag de mise en production.

Changement	Nouveau calculateur de frais derrière un flag de mise en production `dynamic-delivery-fee`, désactivé par défaut ; une migration d'expansion seule ajoute une colonne pouvant être nulle, `fee_rule_version`.
Du commit à l'artefact	Fusionné dans main à 10:02. Le lint, les tests unitaires et les analyses prennent 6 minutes ; l'image `orders:4f1c2ab`, avec sa SBOM et sa signature, est poussée à 10:09.
Migration	La migration additive s'exécute à 10:15 comme une étape distincte du pipeline. La version en cours d'exécution ignore la nouvelle colonne, donc rien ne casse.
Staging	La même image est déployée à 10:16 ; les tests de contrat avec les services de paiement et des restaurants, ainsi que les smoke tests, réussissent à 10:31.
Canari	À 10:35, 5 % du trafic de production va vers la nouvelle version, flag toujours désactivé. Une analyse sur 20 minutes compare la latence p99, le taux de 5xx et le taux de commandes réussies à une référence neuve : réussite.
Déploiement	Les poids passent à 25 %, 50 % puis 100 % par paliers de 15 minutes ; terminé à 11:40. Le déploiement est fini, et pourtant aucun client n'a vu les nouveaux frais.
Mise en production	Le lendemain, la responsable produit active le flag pour 10 % des clients d'une ville et surveille la conversion des commandes et les réclamations, puis élargit à toutes les villes en une semaine.
Nettoyage	Le flag et le code des frais forfaitaires sont supprimés au sprint suivant ; une migration de contraction ultérieure rend la colonne obligatoire.

Le risque de déploiement (le code fonctionne-t-il ?) et le risque de mise en production (les clients apprécient-ils les nouveaux frais ?) ont été traités séparément, chacun avec son propre petit rayon d'impact.

À chaque étape jusqu'à la migration de contraction, la reprise passait par un changement de configuration ou un poids du routeur, pas par un correctif d'urgence. Du commit au déploiement complet en production, il a fallu environ 1 heure 40 minutes.

Mesurer le pipeline

Les métriques DORA de livraison logicielle présentées en section 2.6 (délai de mise en œuvre des changements, fréquence de déploiement, taux d'échec des changements, temps de rétablissement après un déploiement en échec et taux de reprise des déploiements) sont en grande partie des propriétés du pipeline et de l'architecture réunis. Les petits lots et le développement basé sur le tronc raccourcissent le délai et augmentent la fréquence ; les tests automatisés et l'analyse canari font baisser le taux d'échec des changements ; un retour arrière rapide et les flags raccourcissent le temps de rétablissement. Mesurez-les par service, observez les tendances plutôt que des valeurs isolées, et n'en faites jamais des objectifs individuels, ce qui ne fait qu'inviter à tricher.

ATTENTION

Erreurs courantes dans les pipelines

- Reconstruire l'artefact pour chaque environnement, si bien que la production exécute quelque chose qui n'a jamais été testé.
- Une étape de commit qui prend 40 minutes, si bien que les développeurs regroupent leurs changements pour éviter d'attendre, ce qui va à l'encontre du but recherché.
- Des tests instables relancés jusqu'à ce qu'ils passent au vert, ce qui apprend à tout le monde à ignorer les builds rouges.
- Des validations manuelles à chaque étape, qui ajoutent des jours d'attente sans apporter d'information.
- Des migrations intégrées au démarrage de l'application, si bien que chaque instance se précipite pour les appliquer et qu'un retour arrière se heurte à un schéma incompatible.
- Des feature flags jamais supprimés, qui laissent des centaines de combinaisons non testées.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Pourquoi reconstruire l'artefact séparément pour la production pose-t-il problème, même si le code source est identique ?**

Les dépendances, les images de base ou les outils de build peuvent avoir changé entre les builds : l'artefact de production n'est donc pas celui qui a été testé. Construisez une seule fois et promouvez le même artefact.

▼ **Quelle est la différence entre déployer une fonctionnalité et la mettre en production ?**

Le déploiement installe le code dans un environnement ; la mise en production rend le comportement accessible aux utilisateurs. Les feature flags permettent de déployer le code désactivé et de le mettre en production plus tard, progressivement et de façon réversible.

▼ **Pourquoi un canari sur un service à faible trafic donne-t-il souvent des résultats peu fiables ?**

Une petite part d'un trafic faible produit trop peu de requêtes pour détecter avec une confiance statistique des différences de taux d'erreur ou de latence : il faut donc des paliers plus longs, un trafic synthétique ou une autre stratégie.

▼ **Quand le roll forward peut-il être la seule option ?**

Quand la mise en production comprenait un changement irréversible, comme une migration destructrice, ou que d'autres services dépendent déjà du nouveau comportement, si bien que la version précédente ne peut plus fonctionner correctement.

▼ **À vous : Une équipe reconstruit l'image pour chaque environnement, déploie en arrêtant tout et en redémarrant à l'heure du déjeuner, et la nouvelle version supprime une colonne de base de données dans la même mise en production. Trouvez trois problèmes et donnez une meilleure approche.**

Reconstruire signifie que la production exécute quelque chose qui n'a jamais été testé : construisez donc une seule fois et promouvez un artefact immuable unique. Un Recreate à l'heure du déjeuner provoque une interruption en pleine pointe : utilisez donc un déploiement progressif (rolling) ou canary, avec une comparaison automatique à une référence. Supprimer la colonne dans cette version rend le retour arrière impossible, et l'ancienne et la nouvelle version ne peuvent pas fonctionner ensemble : utilisez donc l'expansion-contraction, en la supprimant dans une version ultérieure. Ajoutez un feature flag pour séparer le déploiement de la mise en production.

SOURCES

- Pete Hodgson: Feature Toggles (aka Feature Flags), martinfowler.com: https://martinfowler.com/articles/feature-toggles.html
- Martin Fowler: BlueGreenDeployment: https://martinfowler.com/bliki/BlueGreenDeployment.html
- Danilo Sato: CanaryRelease: https://martinfowler.com/bliki/CanaryRelease.html
- Trunk Based Development: https://trunkbaseddevelopment.com/
- DORA: software delivery performance metrics: https://dora.dev/guides/dora-metrics-four-keys/

SECTION 10.6 SUR 10.8 · AVANCÉ · 24 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Multirégion, reprise après sinistre et edge computing

AVANT DE LIRE

Cette section s'appuie sur :

- 2.5 Disponibilité et fiabilité
- 6.6 CAP et cohérence
- 7.4 Réplication et partitionnement

Termes déjà vus qu'elle utilise : Failover (basculement), Retard de réplication (replication lag), PACELC, Rayon d'impact (blast radius)

L'IDÉE EN BREF

Une chaîne de magasins qui garde tout son stock dans un seul entrepôt est en difficulté si cet entrepôt est inondé. Elle pourrait se contenter d'une liste d'inventaire et d'un double des clés d'un bâtiment vide ailleurs (bon marché, mais lent à remettre en route), garder un petit second entrepôt approvisionné et doté de personnel (plus cher, mais rapide), ou exploiter en permanence deux entrepôts complets (le plus cher, presque sans interruption). Elle peut aussi installer de petits kiosques près des clients pour les articles les plus demandés. La conception multirégion et en périphérie (edge) fait les mêmes choix pour le logiciel, en arbitrant entre ce que l'on paie d'avance et le temps que les clients peuvent attendre.

CE QU'IL FAUT SAVOIR

- Ne fonctionnez sur plusieurs régions que pour la latence, la reprise après sinistre ou la résidence des données ; plusieurs zones de disponibilité dans une même région constituent la base normale et coûtent beaucoup moins cher.
- Le RTO (combien de temps vous pouvez être indisponible) et le RPO (combien de données vous pouvez perdre) viennent du métier et déterminent la stratégie de reprise après sinistre et son coût.
- Les stratégies AWS Well-Architected, sauvegarde et restauration, pilot light, warm standby et multisite actif-actif, offrent des RTO et RPO de plus en plus bas pour un coût permanent et une complexité de plus en plus élevés.
- Les données interrégions sont la partie difficile : la réplication synchrone ajoute de longs allers-retours aux écritures, le retard de la réplication asynchrone devient le RPO effectif, et la réplication copie les suppressions et les corruptions : les sauvegardes restent donc indispensables.

- Le basculement est piloté par le DNS ou par des équilibreurs de charge globaux anycast ; les conceptions statiquement stables conservent une capacité provisionnée à l'avance et évitent de dépendre du plan de contrôle de la région défaillante.
- L'architecture cellulaire et le shuffle sharding limitent le rayon d'impact des mauvais déploiements, des surcharges et des entrées empoisonnées, qu'une seconde région subirait aussi.
- Les CDN et les fonctions en périphérie, comme Cloudflare Workers qui s'exécute dans des isolats V8, servent les lectures et une logique légère près des utilisateurs, tandis que les systèmes de référence restent généralement dans une région.

Pourquoi fonctionner sur plus d'une région

La section 0.7 (serveurs et cloud) a présenté les régions (des zones géographiques distinctes) et les zones de disponibilité (des groupes de centres de données indépendants au sein d'une région), et la section 10.1 a montré comment les fournisseurs les organisent. Fonctionner sur plusieurs zones de disponibilité d'une même région est la base normale de tout système de production : cela survit à la perte d'un centre de données pour un faible surcoût, parce que les zones sont assez proches pour une réplication synchrone. Fonctionner sur plusieurs *régions* est un pas beaucoup plus important, et il n'y a que trois bonnes raisons de le franchir.

La latence. La lumière dans la fibre optique se déplace à environ 200 000 km par seconde : un aller-retour entre Londres et Sydney (environ 17 000 km à vol d'oiseau) ne peut donc pas prendre moins d'environ 170 ms, et les itinéraires réels prennent nettement plus de temps. Une requête qui exige plusieurs allers-retours vers une région lointaine paraît lente, aussi rapides que soient les serveurs. Servir les utilisateurs depuis une région proche, ou depuis la périphérie, est la seule solution.

La reprise après sinistre. Des régions entières tombent bel et bien en panne, généralement à cause d'une dépendance partagée plutôt que d'une catastrophe physique. Les 19 et 20 octobre 2025, une condition de concurrence dans l'automatisation qui gère les enregistrements DNS d'Amazon DynamoDB a laissé son endpoint US-EAST-1 avec un enregistrement DNS vide ; les erreurs de DynamoDB ont duré environ trois heures, mais les défaillances en cascade dans le lancement de nouvelles machines virtuelles et dans les contrôles de santé des équilibreurs de charge ont duré jusqu'à l'après-midi suivant. Tous les grands fournisseurs ont connu des incidents régionaux comparables. La reprise après sinistre s'y prépare.

La résidence des données. Les exigences de résidence des données, issues de la loi, des régulateurs ou des contrats clients, peuvent imposer que certaines données soient stockées ou traitées dans des juridictions précises. Le RGPD lui-même restreint les transferts de données à caractère personnel hors de l'Espace économique européen plutôt qu'il n'exige un stockage

local, mais les règles sectorielles, les lois nationales et les contrats d'entreprise vont souvent plus loin (voir protection de la vie privée dès la conception). La résidence impose souvent des conceptions multirégions dans lesquelles les données ne sont délibérément *pas* répliquées partout.

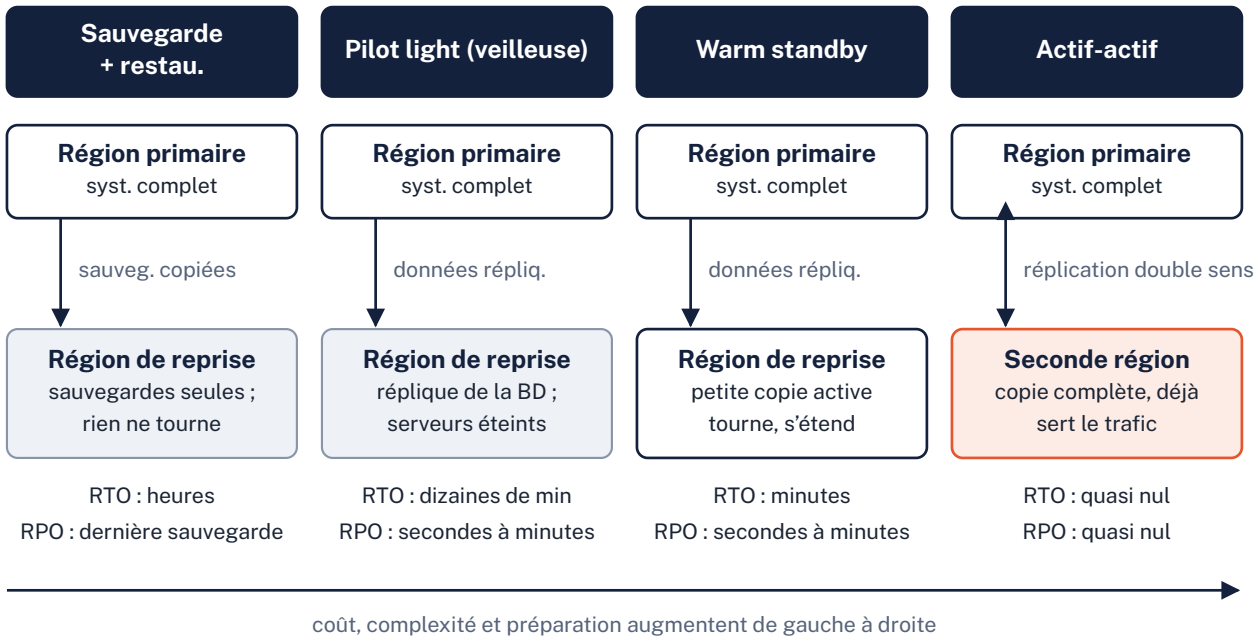
Le multirégion coûte cher : infrastructure dupliquée, frais de transfert de données entre régions, cohérence des données plus difficile, exploitation et tests plus complexes. Beaucoup d'équipes qui pensent en avoir besoin ont en réalité besoin d'une bonne conception multizone, de sauvegardes testées et d'un CDN.

Les objectifs de reprise et les quatre stratégies

La section 2.5 (disponibilité et fiabilité) a défini les deux chiffres qui pilotent toute conception de reprise après sinistre. L'objectif de temps de reprise (RTO) est la durée pendant laquelle le service peut être indisponible ; l'objectif de point de reprise (RPO) est la quantité de données récentes qui peut être perdue, exprimée en temps. Les deux viennent du métier, idéalement du coût d'une heure de panne et d'une commande perdue, et non des préférences des ingénieurs. Des valeurs plus basses coûtent plus cher, et le coût grimpe fortement à l'approche de zéro.

Le livre blanc AWS Well-Architected *Disaster Recovery of Workloads on AWS: Recovery in the Cloud* décrit quatre stratégies, par ordre croissant de coût et de complexité. Leurs noms sont largement utilisés au-delà d'AWS et s'appliquent tout autant à Azure, à Google Cloud ou à une paire de centres de données privés.

Stratégies de reprise après sinistre



Chaque pas vers la droite maintient une plus grande partie du système en fonctionnement dans la seconde région, ce qui achète des RTO et RPO plus bas au prix d'un coût permanent plus élevé.

Comparaison des stratégies de reprise après sinistre

Stratégie	Ce qui tourne dans la seconde région	RTO typique	RPO typique	Coût relatif et principal risque
Sauvegarde et restauration	Rien d'autre que des copies de sauvegarde (instantanés de bases de données, stockage d'objets) et le code d'infrastructure pour reconstruire.	Des heures, selon le volume de données et le degré d'automatisation.	L'intervalle entre sauvegardes (quelques minutes avec des sauvegardes continues des journaux, quelques heures avec des instantanés quotidiens).	Le plus bas. Les restaurations sont lentes et souvent non testées ; sans infrastructure en tant que code, le RTO est une fiction.
Pilot light (veilleuse)	Bases de données répliquées et infrastructure de base ; serveurs d'application définis, mais pas en fonctionnement.	Quelques dizaines de minutes : démarrer et faire monter en capacité le niveau applicatif, puis rediriger le trafic.	Quelques secondes à quelques minutes (retard de réplication).	Faible. Monter en capacité pendant une crise régionale met en concurrence avec tous les autres clients pour obtenir de la capacité.
Warm standby (secours tiède)	Une copie réduite, mais pleinement opérationnelle, de l'ensemble du système.	Quelques minutes : rediriger le trafic, puis monter en capacité.	Quelques secondes à quelques minutes.	Modéré. Le secours peut être testé avec du trafic réel, mais il dérive s'il n'est pas déployé à chaque mise en production.
Multisite actif-actif	Une copie complète qui sert du trafic réel dans chaque région.	Quasi nul : le trafic est redistribué, aucun basculement n'est nécessaire.	Quasi nul pour les données répliquées ; la corruption exige toujours des sauvegardes.	Le plus élevé. Exige une conception des données pour des écritures simultanées dans plusieurs régions.

Le livre blanc lui-même donne un seul chiffre approximatif couvrant à la fois le RTO et le RPO pour chaque stratégie ; le tableau les sépare parce que, avec une réplication continue, le RPO du pilot light et du warm standby est fixé par le retard de réplication plutôt que par la part de l'application qui tourne. Deux mises en garde valent pour toutes les colonnes. La réplication n'est pas une sauvegarde : une suppression accidentelle, une mauvaise migration ou un rançongiciel sont répliqués fidèlement en quelques secondes ; des sauvegardes à un instant donné, idéalement immuables et dans un compte séparé, sont donc nécessaires même dans une conception actif-actif. Et une stratégie qui n'a pas été répétée est un espoir, pas un plan. Les équipes qui font un exercice de basculement régional au moins quelques fois par an découvrent des autorisations manquantes, des endpoints codés en dur, des certificats expirés et des quotas trop petits dans la région de reprise, pendant qu'il est encore peu coûteux de les trouver.

Répliquer les données entre régions

Les serveurs d'application sans état sont faciles à faire tourner à deux endroits ; les données sont la partie difficile, et les compromis sont ceux de la cohérence, du CAP et de PACELC à l'échelle des continents. La réplication synchrone entre régions signifie que chaque écriture attend au moins un long aller-retour, peut-être 70 à 150 ms ou plus, ce qui est acceptable pour quelques écritures critiques et ruineux pour des échanges bavards. La réplication asynchrone garde des écritures rapides, mais la réplique est en retard, et tout ce qui n'est pas arrivé quand la région primaire tombe en panne est perdu : le retard de réplication *est* le RPO effectif.

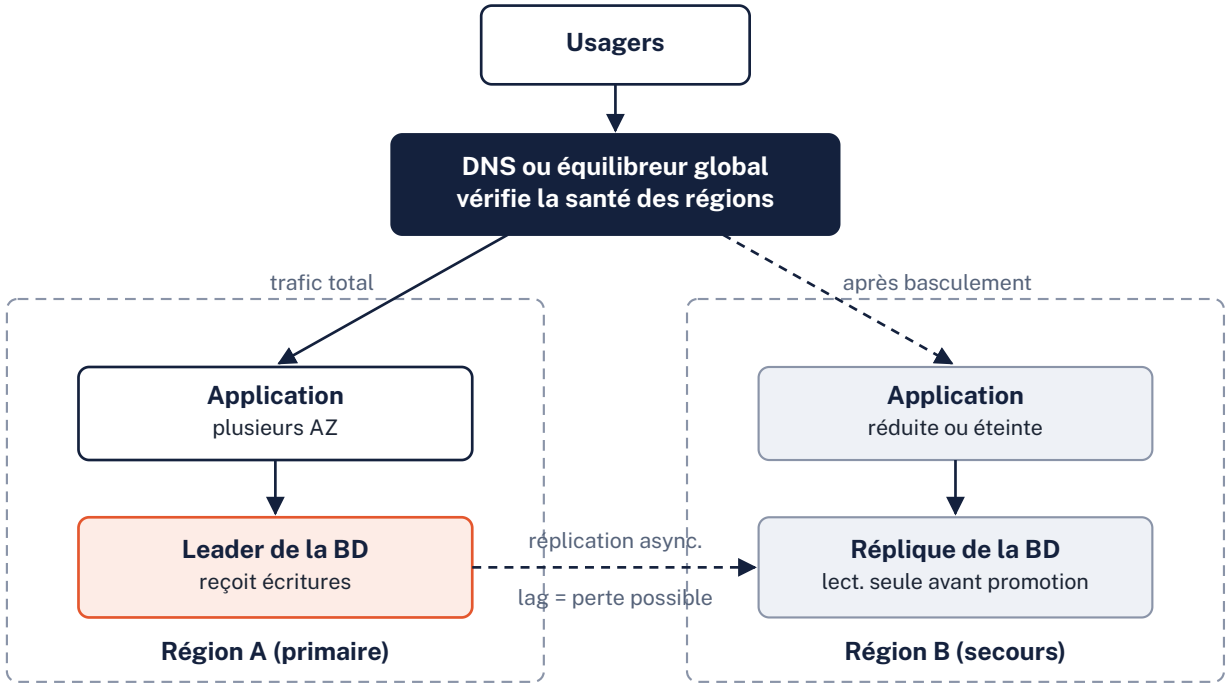
Il existe quatre conceptions de données courantes. **Une seule région primaire** : toutes les écritures vont vers une région, et des répliques ailleurs servent les lectures et la reprise ; c'est simple, mais les utilisateurs lointains paient la latence des écritures, et le basculement implique de promouvoir une réplique (avec les risques de split brain décrits dans réplication et partitionnement). **Une région de rattachement par entité** : chaque client, restaurant ou locataire a une région de rattachement où ses données sont écrites, ce qui garde les écritures locales et favorise la résidence, au prix du routage et de rares opérations interrégions. **Plusieurs écrivains avec résolution des conflits** : chaque région accepte des écritures et les conflits sont fusionnés, par exemple par last-writer-wins (la dernière écriture l'emporte, le comportement par défaut des tables globales DynamoDB, qui offrent aussi depuis 2025 une cohérence forte multirégion) ou par des types de données répliqués sans conflit ; ce n'est sûr que pour les données dont la fusion a un sens, comme un panier, et c'est dangereux pour l'argent ou les stocks. **Les bases de données à cohérence globale**, comme Google Spanner ou CockroachDB, paient la latence du consensus sur les écritures pour offrir une cohérence forte entre régions ; Azure Cosmos DB laisse chaque application choisir parmi cinq niveaux de cohérence, de forte à à terme.

La plupart des systèmes réels combinent ces conceptions selon les jeux de données, exactement comme le recommande la section 6.6 : les paiements et l'état des commandes dans une seule région primaire ou une région de rattachement, les cartes et les informations sur les restaurants répliquées partout pour les lectures, et les données analytiques expédiées de façon asynchrone.

Basculement, DNS et stabilité statique

Le trafic est orienté entre les régions par le DNS ou par un équilibreur de charge global. Avec un basculement (failover) fondé sur le DNS, le service DNS contrôle la santé de chaque région et renvoie l'adresse d'une région saine ; sa faiblesse est la mise en cache, car les résolveurs et les clients peuvent conserver d'anciennes réponses plus longtemps que le TTL de l'enregistrement. Les équilibreurs de charge globaux (par exemple AWS Global Accelerator, Azure Front Door, l'équilibreur de charge externe global de Google Cloud, Cloudflare) utilisent des adresses anycast annoncées depuis de nombreux emplacements : les clients gardent une seule adresse et le fournisseur déplace le trafic en quelques secondes.

Basculement entre une région primaire et une région de secours



Tout ce qui n'a pas été répliqué quand la région A tombe en panne est perdu : le retard de réplication fixe donc le vrai point de reprise.

Décider *quand* basculer est plus difficile que le mécanisme. Le basculement automatique réagit le plus vite, mais il peut être déclenché par une faute de la supervision ou par une brève coupure réseau, et basculer une base de données risque de laisser deux primaires accepter

des écritures. Beaucoup d'organisations automatisent donc la détection et les étapes du basculement, mais conservent une décision humaine pour lancer un basculement régional, avec un runbook et une limite de temps. Le retour à la région d'origine après l'incident, y compris la resynchronisation des données écrites dans la région de reprise, demande lui aussi un plan.

L'incident d'octobre 2025 illustre un principe que l'AWS Builders' Library appelle la stabilité statique : un système devrait continuer à fonctionner pendant une défaillance sans avoir à faire de changements à ce moment-là. Les systèmes qui disposaient déjà d'assez de capacité en fonctionnement ont traversé le problème ; ceux qui devaient lancer de nouvelles instances, ou dont la reprise dépendait d'actions du plan de contrôle dans la région touchée, n'ont pas pu se rétablir avant la région elle-même. Pour la reprise après sinistre, cela plaide pour une capacité provisionnée à l'avance (warm standby plutôt que pilot light, pour les chemins critiques), pour un basculement qui ne dépend que du plan de données de la région saine, et pour un DNS, une gestion des identités et des secrets qui ne résident pas uniquement dans la région que vous quittez.

SI VOUS DÉBUTEZ

Pas à pas : se rétablir sans rien demander

L'idée qui demande le plus d'attention ici est la stabilité statique : un plan de reprise ne vaut que ce que valent les éléments dont il a besoin au moment de la défaillance, et le plan le plus sûr n'a besoin de rien de nouveau. Comparez deux façons de préparer une seconde région pour le cœur de commande de la plateforme de livraison de repas.

- Avec un pilot light, la seconde région détient une copie des données mise à jour en continu et l'infrastructure de base, mais les serveurs d'application sont éteints. Avec un warm standby, une petite copie opérationnelle du cœur de commande y tourne déjà et reçoit un peu de trafic réel.
- À l'heure du dîner, la région primaire tombe en panne. Avec le pilot light, se rétablir signifie faire des changements maintenant : démarrer les serveurs d'application, les faire monter en capacité, puis rediriger le trafic. Chaque démarrage est une requête adressée au plan de contrôle du fournisseur, la mécanique qui crée et modifie les ressources.
- C'est le pire moment pour demander. D'autres clients veulent de la capacité au même moment, et toute étape qui dépend d'un élément conservé uniquement dans la région défaillante, comme le DNS, la gestion des identités ou les secrets, doit attendre que cette région se rétablisse.
- Avec le warm standby, la copie tourne déjà et a fait ses preuves avec du trafic réel. Le trafic y est redirigé immédiatement ; elle sert avec une capacité réduite pendant qu'elle monte en capacité. Si la montée en capacité est lente, les clients obtiennent un service réduit plutôt qu'aucun service.
- La stabilité statique coûte de l'argent : une copie qui tourne en permanence coûte plus cher chaque mois que les serveurs éteints du pilot light, comme le montre l'exemple commenté plus loin dans cette section.
- **La leçon :** pour les chemins critiques, préférez une reprise qui ne s'appuie que sur ce qui tourne déjà dans la région saine. Tout ce qui doit être créé ou modifié pendant le sinistre risque de ne pas être disponible au moment voulu.

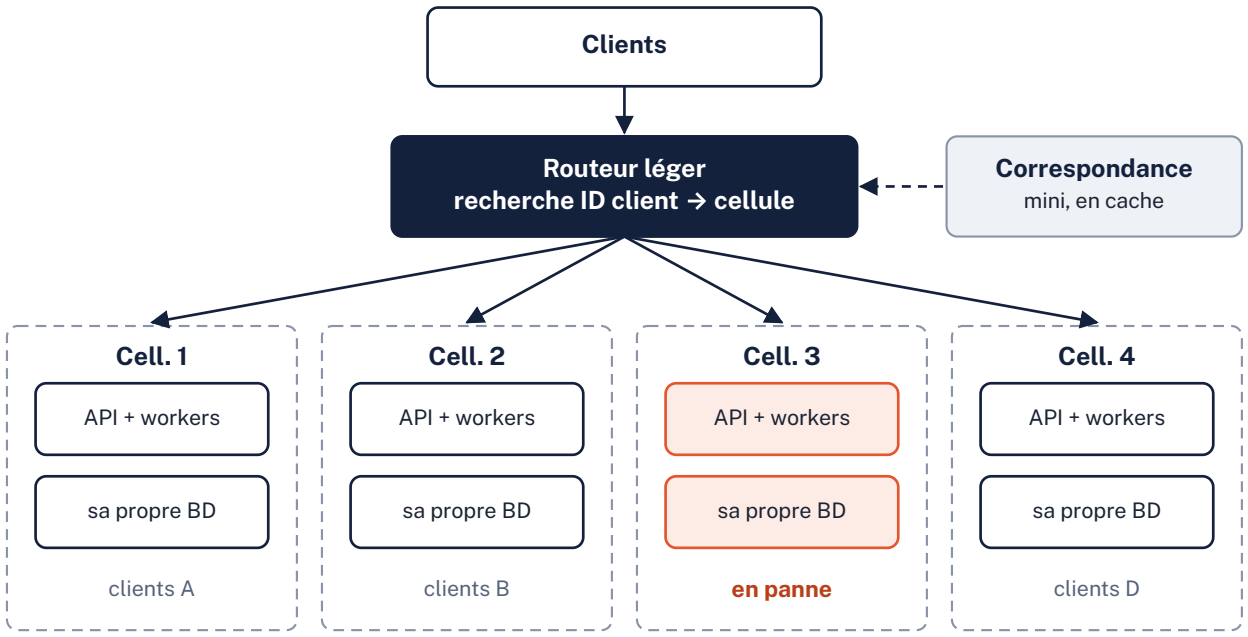
Les cellules : limiter le rayon d'impact

Les régions protègent contre les sinistres régionaux ; la plupart des pannes, cependant, sont causées par les propres changements du système, par des surcharges et par des entrées empoisonnées, qu'une seconde région exécutant le même code et la même configuration subira aussi. L'architecture cellulaire limite le rayon d'impact de ces défaillances en faisant tourner de nombreuses copies complètes et indépendantes du système, appelées cellules, chacune servant un sous-ensemble fixe de clients. Un routeur léger associe chaque requête à

sa cellule, généralement selon l'identifiant du client ou du locataire. Les cellules ne partagent rien à l'exécution : chacune a ses propres instances d'application, ses files et ses bases de données.

Si une cellule subit un mauvais déploiement, un client très actif ou une corruption de données, seuls les clients de cette cellule sont touchés : avec 20 cellules, environ 5 % d'entre eux. Les déploiements se font cellule par cellule, par vagues, si bien qu'une mauvaise version est détectée dans la première cellule. Cette conception est décrite dans le livre blanc AWS Well-Architected *Reducing the Scope of Impact with Cell-Based Architecture* (2023) et est utilisée à grande échelle par les propres services d'AWS et par des entreprises comme Slack. Une technique apparentée, le shuffle sharding, attribue à chaque client une combinaison aléatoire de quelques workers parmi beaucoup (avec 8 workers et 2 par client, il y a 28 combinaisons) : un client dont les requêtes font planter des workers n'abat que sa propre combinaison, et la plupart des autres clients partagent au plus un worker avec lui.

Architecture cellulaire (cell-based architecture)



Faute, mauvais déploiement ou client bruyant dans la cellule 3 : environ un quart des clients touchés, pas tous

Le routeur doit être le composant le plus simple et le plus fiable du système, car c'est la seule chose que toutes les cellules partagent.

Les cellules ont un coût : davantage d'instances et de bases de données à exploiter, des opérations transverses aux cellules (rapports sur l'ensemble des clients, déplacement d'un client d'une cellule à une autre) qui demandent des mécanismes distincts, et un routeur et un service de correspondance qui doivent être extrêmement simples et fiables. Elles conviennent aux grands systèmes multilocataires dotés de clés de partitionnement naturelles ; pour une application interne monolocataire, c'est de la surconception.

La périphérie : CDN et fonctions en périphérie

Entre les utilisateurs et les régions se trouve la périphérie (edge) : les centaines d'emplacements exploités par des réseaux de diffusion de contenu (CDN) comme Akamai, Cloudflare, Fastly, Amazon CloudFront et Azure Front Door. Un CDN met en cache le contenu statique et pouvant être mis en cache (images, scripts, pages de carte) près des utilisateurs, en suivant les règles de mise en cache HTTP décrites dans mise en cache, et absorbe les pics de trafic et de nombreuses attaques par déni de service avant qu'ils n'atteignent l'origine.

Les fonctions en périphérie (edge functions) vont plus loin en exécutant du code à ces emplacements : Cloudflare Workers, Fastly Compute, CloudFront Functions et Lambda@Edge, Akamai EdgeWorkers, et les edge functions de plateformes comme Vercel et Netlify. Cloudflare Workers exécute le code dans des isolats V8 légers, les bacs à sable que les moteurs JavaScript utilisent pour les onglets du navigateur, plutôt que dans un conteneur par fonction : le démarrage est donc très rapide et de nombreux locataires partagent un même processus en toute sécurité. Ce cours lui-même tourne sur Cloudflare Workers. Les bons usages sont le routage des requêtes et l'affectation A/B, les contrôles d'authentification et la validation des tokens, les règles d'en-têtes et de redirection, le redimensionnement d'images, la personnalisation simple, et les proxys d'API qui gardent les clés secrètes hors du navigateur.

Les limites sont architecturales. Les fonctions en périphérie ont des limites strictes de temps CPU et de mémoire ; le stockage en périphérie est généralement à cohérence à terme (Workers KV de Cloudflare peut mettre un certain temps à propager une écriture à l'échelle mondiale) ou n'offre une cohérence forte que pour un seul objet coordinateur, comme le font les Durable Objects ; et le système de référence reste normalement dans une région. Un modèle courant est donc *lire en périphérie, écrire à l'origine* : servir les cartes, les pages des restaurants et les vérifications de session depuis la périphérie, et envoyer les commandes et les paiements vers la région de rattachement.

GUIDE DE DÉCISION

Guide de décision : de combien de multirégion avez-vous besoin ?

Répondez pour chaque parcours utilisateur important, et non pour l'ensemble du système d'un coup.

Q1 **Le métier peut-il tolérer que ce parcours soit indisponible pendant plusieurs heures, le temps de reconstruire une région à partir des sauvegardes ?**

- **Oui** Utilisez plusieurs zones de disponibilité dans une région, plus une sauvegarde et restauration interrégions testée et automatisée. Réexaminez la question si le métier évolue.
- **Non** Continuez : un secours dans une autre région est nécessaire.

Q2 **Un temps de reprise de quelques dizaines de minutes est-il acceptable, et pouvez-vous compter sur l'obtention de capacité pendant un incident régional ?**

- **Oui** Pilot light : répliquez les données en continu, gardez l'infrastructure en tant que code prête et automatisez la montée en capacité et la bascule du trafic.
- **Non** Continuez : la capacité doit déjà être en fonctionnement.

Q3 **Un temps de reprise de quelques minutes est-il acceptable, avec une capacité réduite au début ?**

- **Oui** Warm standby : faites tourner une petite copie opérationnelle, déployez-la à chaque mise en production et envoyez-lui un peu de trafic réel pour prouver qu'elle fonctionne.
- **Non** Actif-actif, avec une conception des données pour les écritures multirégions (régions de rattachement ou base de données à cohérence globale) ; prévoyez un budget pour l'ingénierie, pas seulement pour l'infrastructure.

Q4 **Des utilisateurs éloignés ont-ils besoin d'écritures rapides, ou certaines données doivent-elles rester dans leur juridiction ?**

- **Oui** Partitionnez par géographie : donnez aux clients une région de rattachement et gardez-y leurs données, plutôt que de tout répliquer partout.
- **Non** Servez les lectures des utilisateurs éloignés via un CDN et des fonctions en périphérie, et gardez les écritures dans une seule région primaire.

EXEMPLE COMMENTÉ

Exemple commenté : choisir une stratégie de reprise après sinistre pour la plateforme de livraison de repas

La plateforme fonctionne dans une région, sur trois zones de disponibilité. Le cœur de commande (commandes, paiement, affectation des livreurs) coûte environ 60 000 par mois en infrastructure. Il traite environ 40 000 commandes par jour, avec un pic à 6000 par heure au dîner, et environ 5 de revenus pour la plateforme par commande. La direction demande un plan de reprise en cas de panne régionale. Tous les chiffres sont indicatifs, en livres sterling.

Coût d'une panne	Le chiffre d'affaires est en moyenne d'environ 8000 par heure sur la journée et atteint 30 000 au pic. La perte de confiance des clients, des restaurants et des livreurs double à peu près le coût réel. Les contrats avec les grandes chaînes de restaurants promettent que la prise de commandes est rétablie en moins d'une heure.
Exigence	RTO de 1 heure au plus et RPO de moins de 1 minute pour le cœur de commande ; RTO de 24 heures pour l'analytique, les recommandations et le marketing.
Sauvegarde et restauration	Surcoût d'environ 2000 par mois. Reconstruction mesurée : 4 à 6 heures (la restauration d'une base de données de 2 To domine). Ne respecte pas le RTO de 1 heure.
Pilot light (veilleuse)	Surcoût d'environ 8000 par mois (réplique continue de la base de données, réseau, images prêtes). Basculement répété : 35 à 50 minutes, surtout pour faire monter en capacité le niveau applicatif. Ne respecte le RTO que si la capacité peut être obtenue pendant un événement régional.
Warm standby (secours tiède)	Surcoût d'environ 18 000 par mois (réplique plus application à 20 % de sa taille). Basculement répété : environ 10 minutes, puis montée en capacité tout en servant. Respecte le RTO avec de la marge.
Actif-actif	Surcoût d'environ 45 000 par mois, plus environ six mois-ingénieur estimés pour reconcevoir les données des commandes et des paiements pour des écritures multirégions. RTO quasi nul, bien au-delà de l'exigence.
Décision	Warm standby pour le cœur de commande ; sauvegarde et restauration pour l'analytique, les recommandations et le marketing ; cartes et pages des restaurants servies depuis le CDN pour que la navigation continue pendant le basculement. Surcoût total d'environ 20 000 par mois, soit environ un tiers de l'infrastructure du cœur.
Conditions	Le secours est déployé par le même pipeline à chaque mise en production, reçoit 2 % du trafic réel pour prouver qu'il fonctionne, et un basculement complet est répété chaque trimestre. La décision est consignée dans un ADR.

La perte attendue d'une panne régionale rare ne suffirait pas, à elle seule, à justifier le warm standby ; le RTO contractuel et l'atteinte à la réputation causée par une longue panne à l'heure du dîner, si. La reprise après sinistre s'achète comme une assurance, et le métier fixe la prime qu'il est prêt à payer.

Le découpage par niveaux selon les parcours a nettement réduit le coût : seule la partie du système qui doit se rétablir rapidement paie pour une reprise rapide.

ATTENTION

Les pièges du multirégion

- Une région de reprise qui n'a jamais reçu de trafic : la dérive de configuration, les secrets manquants et les quotas trop faibles n'apparaissent que pendant le sinistre.
- Des outils de basculement, le DNS, la gestion des identités ou le pipeline de déploiement hébergés uniquement dans la région tombée en panne.
- Supposer que la réplication protège contre les suppressions ou la corruption ; elle les copie.
- Des appels interrégions synchrones sur le chemin critique de l'utilisateur, qui ajoutent de longs allers-retours à chaque requête.
- Traiter l'actif-actif comme un réglage d'infrastructure plutôt que comme un problème de conception des données.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi une conception actif-actif a-t-elle toujours besoin de sauvegardes à un instant donné ?

La réplication copie les suppressions, les mauvaises migrations et la corruption dans chaque région en quelques secondes ; seules les sauvegardes permettent de restaurer les données telles qu'elles étaient avant l'erreur.

▼ Dans une conception avec réplication asynchrone vers une région de secours, qu'est-ce qui détermine la quantité de données perdues lors d'une défaillance régionale ?

Le retard de réplication au moment de la défaillance : les écritures qui n'avaient pas encore atteint la réplique sont perdues, donc le retard est le RPO effectif.

▼ **Contre quelles défaillances une architecture cellulaire protège-t-elle, alors qu'une seconde région exécutant le même code ne le fait pas ?**

Les défaillances causées par le système lui-même, comme un mauvais déploiement, une entrée empoisonnée ou un client en surcharge, qui toucheraient toutes les régions, mais une seule cellule.

▼ **Pourquoi le pilot light est-il plus risqué que le warm standby lors d'un incident régional de grande ampleur ?**

Le pilot light doit lancer le niveau applicatif et le faire monter en capacité au moment de la défaillance, quand la capacité et les opérations du plan de contrôle risquent d'être rares ; le warm standby dispose déjà d'une capacité opérationnelle en fonctionnement.

▼ **À vous : Exemple indicatif : la sauvegarde et restauration signifie une panne régionale de 8 heures à £8000 perdues par heure. Un warm standby (secours tiède) coûte £6000 par mois et la réduit à 20 minutes. Une panne régionale est attendue tous les cinq ans. Comparez les coûts annuels et donnez votre conseil.**

Sauvegarde : 8 x £8000 = £64 000 par panne, soit £12 800 par an. Standby : £72 000 par an, plus environ £533 de perte attendue. Sur le seul chiffre d'affaires, ce n'est pas justifié. Vérifiez le RTO et le RPO avec le métier : les contrats, la sûreté ou la perte de clients pourraient changer la réponse, et un pilot light moins cher pourrait convenir. Gardez des sauvegardes testées quel que soit votre choix, car la réplication copie les suppressions et les corruptions.

SOURCES

- AWS Whitepaper: Disaster Recovery of Workloads on AWS, disaster recovery options in the cloud: https://docs.aws.amazon.com/whitepapers/latest/disaster-recovery-workloads-on-aws/disaster-recovery-options-in-the-cloud.html
- AWS Whitepaper: Reducing the Scope of Impact with Cell-Based Architecture: https://docs.aws.amazon.com/wellarchitected/latest/reducing-scope-of-impact-with-cell-based-architecture/reducing-scope-of-impact-with-cell-based-architecture.html
- AWS: Summary of the Amazon DynamoDB service disruption in the Northern Virginia (US-EAST-1) Region: https://aws.amazon.com/message/101925/
- Amazon Builders' Library: Workload isolation using shuffle-sharding: https://aws.amazon.com/builders-library/workload-isolation-using-shuffle-sharding/
- Cloudflare Workers documentation: How Workers works: https://developers.cloudflare.com/workers/reference/how-workers-works/

SECTION 10.7 SUR 10.8 · INTERMÉDIAIRE · 24 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Ingénierie de plateforme et plateformes internes de développement

AVANT DE LIRE

Cette section s'appuie sur :

- 3.7 Loi de Conway
- 10.4 Infrastructure en tant que code
- 10.5 CI/CD et déploiement

Termes déjà vus qu'elle utilise : Kubernetes, Infrastructure en tant que code (IaC), Politique en tant que code (policy as code), DevOps, Observabilité

L'IDÉE EN BREF

Imaginez un grand bureau où chaque département doit acheter ses propres imprimantes, câbler son propre réseau et embaucher son propre agent d'entretien. Le travail se fait, mais tout le monde perd du temps en corvées et certains départements s'en acquittent mal. Une bonne équipe des services généraux propose au contraire des services prêts à l'emploi : réserver une salle de réunion en ligne, commander un ordinateur portable dans une courte liste, demander un bureau en quelques clics. Personne n'est obligé de les utiliser, mais c'est la voie la plus simple. L'ingénierie de plateforme est cette équipe des services généraux pour les équipes logicielles : elle rend rapides, sûres et accessibles en libre-service les corvées courantes et nécessaires de la construction et de l'exploitation des logiciels.

CE QU'IL FAUT SAVOIR

- L'ingénierie de plateforme construit une plateforme interne de développement : des outils, services, connaissances et un support en libre-service qui permettent aux équipes produit de construire et d'exploiter des logiciels sans tickets, en réduisant les efforts en double et l'inégalité de qualité.
- Un golden path (paved road) est une manière d'accomplir une tâche courante qui fait des choix affirmés, bénéficie d'un support, couvre tout de bout en bout et reste facultative, comme créer un nouveau service et le déployer en production.
- Backstage, créé chez Spotify et passé en open source en 2020, est un projet de la CNCF qui offre un catalogue logiciel, des modèles, TechDocs et des plug-ins ; un portail est la porte d'entrée, pas la plateforme elle-même.

- La plateforme est un produit avec des clients internes : elle a besoin d'un responsable produit (product owner), de recherche utilisateur, de documentation, de ses propres SLO et d'une adoption méritée plutôt qu'imposée (le mode X-as-a-Service et la plateforme viable la plus fine de Team Topologies).
- Le libre-service est sécurisé par des garde-fous, c'est-à-dire des vérifications automatiques de politiques avec des messages clairs et une procédure d'exception, plutôt que par des points de validation humains.
- Le succès se mesure aux résultats des autres équipes : adoption, délai jusqu'au premier déploiement, métriques DORA et expérience développeur, en utilisant plusieurs dimensions de SPACE plutôt qu'un seul chiffre.
- Les échecs typiques consistent à construire une plateforme dont personne ne veut, à renommer une équipe d'exploitation pilotée par les tickets, à imposer l'adoption et à créer des abstractions qui cachent trop pour permettre le débogage.

Le problème que résolvent les plateformes

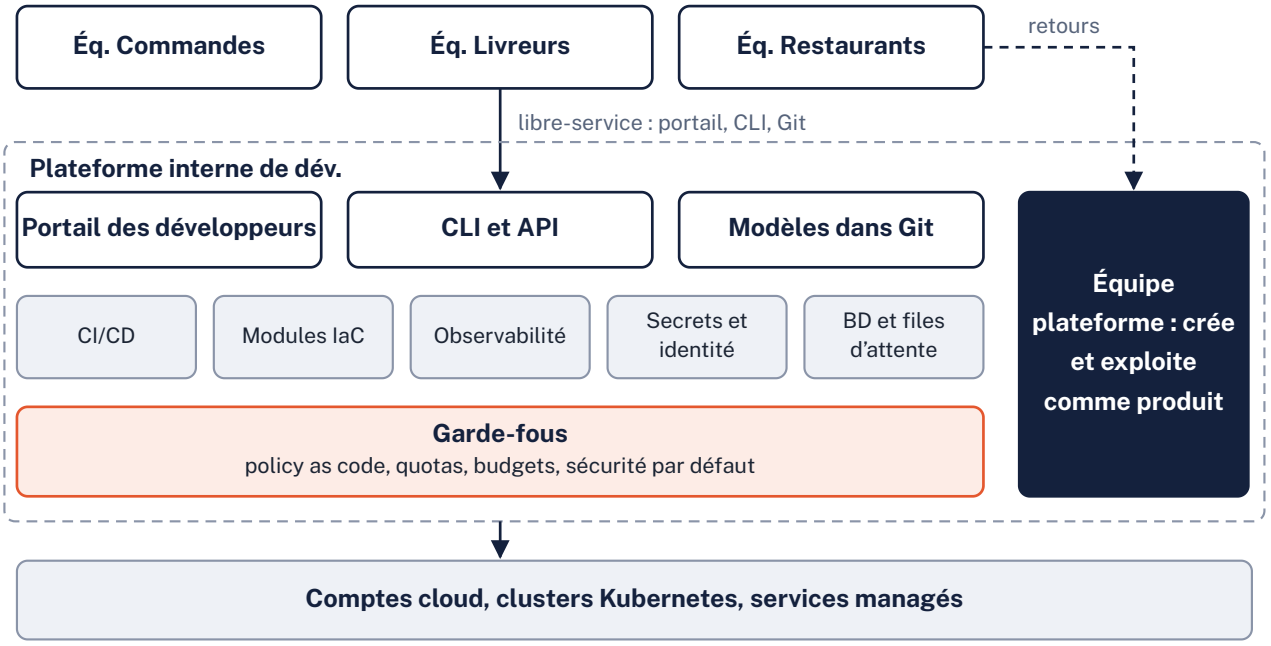
En 2006, le directeur technique d'Amazon, Werner Vogels, a résumé le modèle de fonctionnement de son entreprise par «you build it, you run it» (vous le construisez, vous l'exploitez) : l'équipe qui écrit un service le déploie, le supervise et le corrige aussi. L'idée est devenue centrale pour le DevOps et pour les microservices, et elle fonctionne : les équipes qui exploitent leur propre logiciel subissent les conséquences de leurs choix de conception et les corrigent. Mais l'outillage cloud native décrit dans ce chapitre est vaste. Une équipe produit qui doit aujourd'hui construire, exploiter et sécuriser un service peut avoir besoin de comprendre les images de conteneurs, les objets Kubernetes, l'infrastructure en tant que code, les pipelines, l'observabilité, la gestion des secrets, les politiques réseau, l'analyse de la chaîne d'approvisionnement et le contrôle des coûts, avant d'écrire la moindre ligne de logique métier.

Lorsque chaque équipe résout tout cela seule, trois choses se produisent. Les efforts sont dupliqués : dix équipes écrivent dix pipelines légèrement différents. La qualité varie : certaines équipes chiffrent leurs bases de données et configurent des sauvegardes, d'autres oublient. Et le temps que les équipes produit consacrent à l'infrastructure est du temps qu'elles ne consacrent pas au produit. L'ingénierie de plateforme (platform engineering) est la réponse : une équipe dédiée construit une plateforme interne de développement (IDP) qui regroupe ces capacités pour que les équipes produit puissent les utiliser en libre-service.

L'article d'Evan Bottcher publié en 2018 sur martinfowler.com en donne une description très citée : une plateforme numérique est un socle d'API, d'outils, de services, de connaissances et de support en libre-service, organisés comme un produit interne convaincant. Chaque élément de cette description compte. *En libre-service* signifie pas de tickets pour les besoins courants. *Connaissances et support* signifie de la documentation et des personnes, pas seulement du

logiciel. *Convaincant* signifie que les équipes choisissent de l'utiliser parce que c'est mieux que de faire le travail elles-mêmes. *Produit interne* signifie qu'elle est gérée comme un produit, avec des clients, plutôt que comme un projet ou un centre de coûts.

Ce que contient une plateforme interne de développement



Les équipes produit consomment la plateforme au moyen d'interfaces en libre-service ; l'équipe plateforme la gère comme un produit et tire des enseignements de leurs retours.

Une IDP n'est pas un produit unique. C'est généralement une intégration d'outils existants (un système de CI, un contrôleur GitOps, les services managés d'un fournisseur cloud, une pile d'observabilité, un coffre de secrets) derrière des interfaces cohérentes et des valeurs par défaut raisonnables. Le travail propre à l'équipe plateforme consiste à les choisir, les intégrer, les documenter et les prendre en charge, et à concevoir l'expérience de leur utilisation, pas à écrire un nouvel orchestrateur.

Golden paths et paved roads

Ce qu'une jeune plateforme peut offrir de plus précieux est un golden path (chemin balisé) : une manière prescriptive, prise en charge et documentée d'accomplir de bout en bout une tâche courante, comme créer un nouveau service et le mettre en production. Spotify a popularisé le terme (son blog d'ingénierie a décrit ses golden paths en 2020) ; Netflix parle de *paved road* (route goudronnée). La métaphore est précise. Une route goudronnée est l'itinéraire le plus facile, et elle est entretenue, mais vous pouvez toujours partir hors piste si vous avez une bonne raison ; vous ne bénéficiez simplement plus du soutien de la route.

Un bon golden path a cinq propriétés. Il est **de bout en bout** : il mène une équipe d'un dépôt vide à un service supervisé en production, et pas seulement à travers une étape. Il est **prescriptif** : il fait les choix courants (versions de langage, format de journalisation, stratégie de déploiement, réglages par défaut de la base de données) pour que les équipes n'aient pas à les faire. Il est **facultatif** : les équipes aux besoins inhabituels peuvent le quitter, ce qui oblige l'équipe plateforme à rester honnête sur la question de savoir si le chemin est vraiment meilleur. Il est **pris en charge** : quand il casse, l'équipe plateforme le répare. Et il est **évolutif** : les services créés à partir du modèle de l'an dernier reçoivent des mises à jour, au lieu de rester figés dans leur version de départ.

Ce que fournit généralement un golden path pour un nouveau service

Élément	Ce que reçoit l'équipe produit	Ce que cela remplace
Modèle de service	Un dépôt avec un squelette fonctionnel, des tests, un Dockerfile, du linting et un README, généré en quelques minutes.	Copier un ancien service et supprimer ce qui ne convient pas.
Pipeline	Un pipeline CI/CD prêt à l'emploi, avec analyse, signature et déploiement progressif, comme à la section 10.5.	Chaque équipe écrit et maintient son propre pipeline.
Configuration d'exécution	Des réglages Kubernetes ou de plateforme managée, avec des requêtes de ressources, des sondes et un autoscaling raisonnables.	Des essais et erreurs en production.
Modules d'infrastructure	Des modules d'infrastructure en tant que code relus pour les bases de données, les files d'attente et les buckets, avec chiffrement, sauvegardes et étiquettes intégrés.	Des tickets à une équipe d'exploitation, ou des modules écrits à la main, de qualité variable.
Observabilité	Journalisation structurée, métriques, traces et un tableau de bord de départ avec des alertes sur les objectifs de niveau de service.	Une supervision ajoutée après le premier incident.
Entrée de catalogue et documentation	Le service enregistré avec son propriétaire, ses dépendances et sa documentation.	Des tableurs et un savoir informel sur qui possède quoi.

Backstage et les portails des développeurs

Un portail des développeurs est la porte d'entrée de nombreuses plateformes. Le plus connu est Backstage, construit en interne chez Spotify, passé en open source en mars 2020, accepté par la Cloud Native Computing Foundation (CNCF) comme projet sandbox en septembre 2020 et promu au stade d'incubation en mars 2022. Ses fonctions principales sont un **catalogue logiciel**, dans lequel chaque service, site web, bibliothèque, API et ressource est enregistré avec son propriétaire et ses relations ; des **modèles logiciels**, qui génèrent de nouveaux projets

à partir des modèles des golden paths ; **TechDocs**, qui publie la documentation écrite en Markdown à côté du code ; et un vaste écosystème de plug-ins qui affichent les pipelines, l'état Kubernetes, les coûts et les incidents à côté de chaque entrée du catalogue.

Chaque composant se décrit lui-même dans un petit fichier conservé dans son propre dépôt, de sorte que les informations de propriété vivent avec le code et sont relues comme du code :

```
apiVersion: backstage.io/v1alpha1
kind: Component
metadata:
  name: restaurant-ratings
  description: Stores and serves customer ratings of restaurants
  tags: [python, http]
spec:
  type: service
  lifecycle: production
  owner: team-restaurants
  system: restaurant-discovery
  providesApis: [ratings-api]
  dependsOn: [resource:ratings-db]
```

Une entrée du catalogue Backstage (catalog-info.yaml) pour un service de livraison de repas : son propriétaire, son cycle de vie, le système auquel il appartient, l'API qu'il fournit et une ressource dont il dépend.

Deux mises en garde. Backstage est un framework qu'une organisation assemble et exploite, ce qui exige généralement des ingénieurs qui maîtrisent TypeScript et React, et ses plug-ins sont de qualité variable ; des services Backstage managés et des portails commerciaux existent pour les équipes qui ne veulent pas l'exploiter elles-mêmes. Plus important encore, **un portail n'est pas une plateforme**. Un beau portail placé devant des processus manuels pilotés par les tickets ne fait qu'afficher la file d'attente de façon plus attrayante. Commencez par les capacités et le golden path ; ajoutez un portail quand il y a derrière assez de choses qui valent la peine d'être trouvées.

La plateforme en tant que produit

L'idée la plus importante de l'ingénierie de plateforme est que la plateforme est un produit et que les développeurs qui l'utilisent sont ses clients. Cela suppose un responsable produit ou un responsable de la plateforme qui porte la feuille de route ; de la recherche utilisateur (entretiens, observation d'une équipe qui crée un service, enquêtes) avant de construire ; une documentation et une prise en main traitées comme des fonctionnalités ; des objectifs de niveau de service publiés pour la plateforme elle-même ; des politiques de versionnement et de dépréciation pour que les équipes puissent planifier ; et, surtout, une adoption méritée plutôt qu'imposée. Si les équipes évitent la plateforme, c'est un retour du marché, pas de l'insubordination.

Team Topologies (Matthew Skelton et Manuel Pais, 2019), présenté à la section 11.5, fournit le modèle organisationnel. Une équipe plateforme existe pour alléger la charge des équipes orientées flux, celles qui sont alignées sur un flux de changements métier comme la commande ou l'affectation des livreurs, et interagit avec elles principalement en mode *X-as-a-Service* : une offre claire, en libre-service, qui demande peu de collaboration. Le mode collaboration est utilisé délibérément et brièvement, par exemple pour concevoir une nouvelle capacité avec une ou deux équipes pionnières. Skelton et Pais ont aussi forgé la notion de plateforme viable la plus fine (thinnest viable platform) : la plus petite plateforme qui aide réellement, et qui, pour une petite organisation, peut se résumer à une page de wiki bien écrite listant les services approuvés et à un modèle de pipeline partagé.

Cela signifie aussi que la plateforme doit faire jouer la loi de Conway en sa faveur : les interfaces de la plateforme deviennent les frontières entre l'équipe plateforme et chaque équipe produit ; elles doivent donc être stables, documentées et étroites, comme toute bonne abstraction.

Le libre-service avec garde-fous

Le libre-service supprime la file d'attente ; les garde-fous le rendent sûr. La différence avec les contrôles traditionnels tient à l'endroit où la vérification a lieu. Un **point de validation** (gate) est une personne qui doit approuver une demande, comme un comité consultatif des changements ou un administrateur de bases de données qui examine un ticket. Un **garde-fou** (guardrail) est une limite automatique qui laisse passer tout ce qui respecte la politique et bloque ou signale ce qui ne la respecte pas, au moment même de la demande. Les garde-fous sont généralement écrits sous forme de politique en tant que code (policy as code, présentée à la section 10.4) et appliqués dans les pipelines, les contrôleurs d'admission et les politiques d'organisation du cloud.

Capacités en libre-service et leurs garde-fous

Capacité	Interface en libre-service	Exemples de garde-fous
Nouveau service	Modèle dans le portail ou la CLI	Doit avoir un propriétaire dans le catalogue ; doit utiliser l'image de base approuvée.
Base	Module d'infrastructure ou formulaire du portail	Chiffrement au repos activé ; sauvegardes automatiques activées ; aucun accès réseau public ; taille limitée par le quota de l'équipe.
Déploiement	Une fusion dans main déclenche le pipeline	Uniquement des images signées provenant du registre interne ; requêtes de ressources et sondes obligatoires ; analyse canari obligatoire pour les services de niveau 1.
Dépenses cloud	Comptes d'équipe ou namespaces créés à la demande	Étiquettes de coût obligatoires ; alertes budgétaires à 80 % et 100 % ; les types d'instances coûteux doivent être justifiés.
Accès	Demandes de rôles via le portail	L'accès à la production est limité dans le temps et journalisé ; pas de clés personnelles à longue durée de vie.

Les bons garde-fous partagent trois propriétés : ils donnent un message clair qui explique ce qui a été bloqué et comment corriger le problème ; ils sont versionnés et testés comme du code, avec une période d'avertissements avant qu'une nouvelle règle ne commence à bloquer ; et ils s'accompagnent d'une procédure documentée pour les exceptions justifiées, car un garde-fou sans procédure d'exception redevient un point de validation dès la première fois qu'il se trompe.

SI VOUS DÉBUTEZ

Pas à pas : point de validation ou garde-fou ?

Le point qui fait le plus souvent trébucher ici est la différence entre un point de validation et un garde-fou. Tous deux peuvent faire respecter les mêmes règles, mais un point de validation place une personne devant chaque demande, tandis qu'un garde-fou vérifie automatiquement au moment de la demande. Suivez une même demande de compte dans les deux cas.

- L'équipe Restaurants a besoin d'un compte cloud pour son nouveau service de notation des restaurants. Les règles exigent que les alertes budgétaires soient activées, que chaque ressource porte des étiquettes de coût et que les types de machines coûteux soient justifiés.
- Avec un point de validation, l'équipe ouvre un ticket. Quand celui-ci arrive en tête de la file, peut-être plusieurs jours plus tard, un ingénieur des opérations cloud le vérifie au regard des règles et crée le compte. Les règles sont respectées, mais au prix de l'attente.
- Avec un garde-fou, l'équipe demande le compte via le portail de la plateforme. Des vérifications automatiques, écrites sous forme de politique en tant que code, le comparent aux mêmes règles. La demande est acceptée, et le compte est créé sans ticket ni approbation humaine.
- Plus tard, l'équipe essaie de démarrer un serveur sans étiquettes de coût. La vérification le bloque immédiatement, avec un message qui indique ce qui a été bloqué, pourquoi, et comment corriger le problème. L'équipe ajoute les étiquettes et réessaie.
- Si l'équipe avait un cas que les règles n'avaient pas prévu, elle utiliserait la procédure d'exception documentée. Sans elle, la première règle erronée laisse l'équipe bloquée, et le garde-fou est redevenu un point de validation.
- **La leçon :** un garde-fou n'affaiblit pas les règles ; il déplace la vérification de la file d'attente d'une personne vers le moment de la demande. Il fonctionne lorsqu'il s'explique, qu'il est testé comme du code et qu'il laisse une voie pour les exceptions justifiées.

Réduire la charge cognitive, sans tout cacher

Team Topologies emprunte la notion de charge cognitive aux recherches en sciences de l'éducation de John Sweller et en distingue trois types. La charge intrinsèque est inhérente à la tâche, comme comprendre le fonctionnement des frais de livraison. La charge extrinsèque vient de l'environnement, comme se souvenir des douze étapes pour demander une base de données. La charge pertinente est l'effort d'apprentissage qui facilite le travail futur, comme comprendre

le domaine plus en profondeur. Une plateforme doit éliminer la charge extrinsèque pour que les équipes puissent consacrer leur attention limitée à la charge intrinsèque et à la charge pertinente de leur propre domaine.

Cela ne veut pas dire tout cacher. Une équipe qui exploite son service doit toujours comprendre comment il se comporte en production : sa consommation de ressources, ses modes de défaillance, sa latence et ses coûts. Les abstractions qui cachent trop fuient au pire moment, pendant un incident, quand l'équipe découvre qu'elle ne peut pas voir les journaux ou ne sait pas sur quelle gamme de base de données elle tourne. Les bonnes plateformes sont transparentes : elles fournissent des valeurs par défaut raisonnables et des interfaces simples, tout en laissant les équipes regarder en dessous et modifier les réglages quand elles en ont besoin.

Mesurer si la plateforme fonctionne

La valeur d'une plateforme se manifeste dans les résultats des autres équipes : c'est donc là que ses mesures doivent porter. Les mesures utiles se répartissent en quatre groupes, et un ensemble équilibré en prend au moins une dans chacun.

Mesures du succès d'une plateforme

Groupe	Exemples de mesures	Points de vigilance
Adoption	Part des services sur le golden path ; adoption volontaire des nouvelles capacités ; équipes qui quittent le chemin, et pourquoi.	Une adoption imposée ne dit rien de la valeur.
Livraison	Métriques DORA pour les équipes sur la plateforme et hors de celle-ci ; délai entre la création d'un service et son premier déploiement en production ; durée d'intégration d'un nouvel ingénieur.	Les comparaisons entre équipes dont le travail est très différent.
Expérience développeur	Enquêtes régulières sur la satisfaction, les frictions perçues et le temps perdu à cause de l'outillage ; entretiens.	La lassitude face aux enquêtes ; traitez les résultats comme des signaux, pas comme des notes.
Santé de la plateforme	Les SLO propres à la plateforme (disponibilité des pipelines, délai de provisionnement), la charge de support, le coût par équipe.	Une plateforme fiable mais inutilisée reste un échec.

Le cadre SPACE, publié dans ACM Queue en 2021 par Nicole Forsgren, Margaret-Anne Storey et des collègues de GitHub, de Microsoft Research et de l'Université de Victoria, avertit que la productivité des développeurs ne peut pas se résumer à un seul chiffre. Il nomme cinq dimensions : la satisfaction et le bien-être, la performance, l'activité, la communication et la collaboration, ainsi que l'efficacité et le flux. Il recommande de mesurer plusieurs dimensions à la fois, dont au moins une mesure perçue comme une enquête, et de ne jamais juger les individus sur des comptages d'activité comme le nombre de commits ou de lignes de code.

Les données sur les plateformes sont encourageantes, mais pas inconditionnelles. Le rapport DORA 2024 *Accelerate State of DevOps* a constaté que les utilisateurs de plateformes internes de développement déclaraient une productivité individuelle supérieure d'environ 8 % et une performance d'équipe supérieure de 10 %, mais que le débit de livraison logicielle et la stabilité des changements étaient inférieurs de 8 % et 14 %. Le rapport avance des explications possibles, comme des passages de relais supplémentaires et la couche de plateforme qui ajoute elle-même des étapes, et la leçon pour les architectes est de mesurer les résultats de livraison autant que la satisfaction, et de s'attendre à un creux pendant qu'une plateforme mûrit.

EXEMPLE COMMENTÉ

Exemple commenté : le premier golden path pour un nouveau service de livraison de repas

L'entreprise de livraison de repas compte douze équipes produit. Une enquête et une semaine passée à observer les équipes au travail montrent que créer un nouveau service prend environ trois semaines et quatorze tickets adressés à quatre équipes différentes. Une équipe plateforme de quatre personnes est formée. Les premières équipes produit à avoir besoin d'un nouveau service sont l'équipe Restaurants, qui veut un service de notation des restaurants, et l'équipe Livreurs.

Choisir le chemin	Inventaire des services existants : 70 % sont des services HTTP sans état en Python ou en Go, avec une base de données PostgreSQL et, souvent, un abonnement à une file d'attente. Cette forme devient le golden path v1 ; tout le reste reste hors piste pour l'instant.
Concevoir avec les utilisateurs	Deux ingénieurs des équipes Restaurants et Livreurs rejoignent l'équipe plateforme pendant trois semaines (mode collaboration) pour construire le chemin avec elles, et non pour elles.
Sommaire	Dépôt modèle ; le pipeline standard avec déploiement canari ; des réglages Kubernetes avec requêtes, sondes et autoscaling ; un module de base de données avec chiffrement, sauvegardes et étiquettes de coût ; journalisation, traçage et un tableau de bord des SLO ; une entrée de catalogue ; un guide de démarrage.
Garde-fous	Étiquettes de propriétaire et de coût obligatoires ; images signées uniquement ; base de données sans accès public. Tout est vérifié automatiquement ; avertissements pendant un mois avant le blocage.
Premier résultat	L'équipe Restaurants crée restaurant-ratings depuis le portail et déploie une version « hello world » en production en 5 heures, contre trois semaines auparavant. Tickets nécessaires : aucun.
Mesurer	Délai jusqu'au premier déploiement en production, enquête auprès des deux équipes pilotes et métriques DORA pour les services sur le chemin, examinés chaque mois. Trois autres équipes adoptent le chemin volontairement en un trimestre.
Étapes suivantes	Les équipes demandent la prise en charge des consommateurs d'événements et des environnements de prévisualisation par pull request ; la feuille de route est ordonnée selon la demande. Une demande de prise en charge d'un langage de niche est refusée pour l'instant et documentée comme hors piste.

La plateforme a commencé par un seul chemin pour le cas le plus courant, construit avec ses premiers utilisateurs, et a été jugée sur leurs résultats plutôt que sur le nombre de fonctionnalités livrées.

L'essentiel du travail a consisté à intégrer des outils existants, à définir leurs valeurs par défaut et à les documenter, pas à écrire de nouveaux logiciels.

Risques et modes de défaillance

Les initiatives de plateforme échouent de manières reconnaissables. La plus courante consiste à **construire une plateforme dont personne ne veut** : une équipe passe un an à bâtir une plateforme généraliste ambitieuse, la lance, et découvre que les équipes produit ont déjà des solutions qui fonctionnent et pas de temps pour migrer. Commencer petit, avec de vrais utilisateurs et un seul golden path, en est l'antidote. La deuxième est **l'équipe d'exploitation renommée** : une équipe d'infrastructure est rebaptisée équipe plateforme mais travaille toujours à partir d'une file de tickets, si bien que rien ne change, sauf les intitulés de poste. Si les demandes courantes exigent encore un ticket et une personne, ce n'est pas encore une plateforme.

Les autres risques sont d'imposer l'adoption, ce qui supprime les retours qui maintiennent l'utilité d'une plateforme ; la sur-abstraction, où une couche maison masque si complètement les outils sous-jacents que les équipes ne peuvent ni la déboguer ni l'étendre, et où l'équipe plateforme devient un goulot d'étranglement pour chaque changement ; la négligence de la longue traîne des services créés à partir d'anciens modèles ; et une plateforme devenue point de défaillance unique, dont la panne arrête tous les déploiements. À l'inverse, une petite organisation peut n'avoir besoin d'aucune équipe plateforme : quelques modèles partagés, entretenus par des volontaires, peuvent constituer la plateforme viable la plus fine pendant des années.

Liste de contrôle : lancer une plateforme

- [ ] **Des preuves du problème**
  Mesurez combien de temps prennent aujourd'hui les tâches courantes (nouveau service, nouvelle base de données, premier déploiement) et combien de tickets et d'équipes elles mobilisent.
- [ ] **Un responsable produit**
  Quelqu'un porte la feuille de route, parle régulièrement aux utilisateurs et sait dire non.
- [ ] **D'abord un seul golden path**
  Choisissez la forme de service la plus courante et rendez-la excellente de bout en bout avant d'élargir l'offre.
- [ ] **Le libre-service, pas les tickets**
  Chaque demande courante sur le chemin aboutit sans approbation humaine ; les exceptions disposent d'une procédure documentée.
- [ ] **Des garde-fous en tant que code**
  Des règles de sécurité, de coût et de fiabilité appliquées automatiquement, avec des messages clairs et une période d'avertissement.
- [ ] **Des abstractions transparentes**
  Les équipes peuvent voir les journaux, les métriques, les coûts et les ressources sous-jacentes, et modifier les valeurs par défaut quand c'est justifié.
- [ ] **Des mesures de résultats**
  Adoption, délai jusqu'au premier déploiement, métriques DORA et enquêtes auprès des développeurs, examinés régulièrement et publiés.
- [ ] **Un chemin de mise à niveau**
  Les services créés à partir d'anciens modèles reçoivent les améliorations, et les dépréciations s'accompagnent d'un préavis et d'une aide à la migration.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Quelle est la différence entre un point de validation et un garde-fou ?**

Un point de validation exige qu'une personne approuve chaque demande ; un garde-fou est une vérification automatique qui laisse passer immédiatement les demandes conformes et bloque ou signale les autres.

▼ **Pourquoi un golden path doit-il être facultatif ?**

Pour que les équipes aux besoins inhabituels ne soient pas bloquées, et pour que l'équipe plateforme apprenne, en voyant qui quitte le chemin, s'il est réellement meilleur que les autres options.

▼ **Une entreprise installe Backstage, mais créer une base de données exige toujours un ticket et une semaine d'attente. Qu'a-t-elle construit ?**

Un portail devant un processus manuel, pas une plateforme ; ce sont les capacités en libre-service derrière le portail qui créent de la valeur.

▼ **Pourquoi le cadre SPACE déconseille-t-il de mesurer la productivité des développeurs uniquement au nombre de commits ou de lignes de code ?**

Les comptages d'activité ne saisissent qu'une dimension, sont faciles à manipuler et ne disent rien de la satisfaction, de la collaboration, du flux ni de la valeur du résultat.

▼ **À vous : La direction renomme l'équipe des opérations « Platform », impose à toutes les équipes de l'utiliser et mesure le succès au nombre de tickets clos. Les demandes de nouveau service prennent toujours quatorze tickets. Critiquez cela et dites ce que vous changeriez.**

C'est une équipe pilotée par les tickets et rebaptisée, pas une plateforme. Traitez-la comme un produit : donnez-lui un product owner, étudiez les besoins des équipes produit et construisez un golden path minimal pour la tâche la plus courante, un nouveau service sans état, en libre-service avec des garde-fous automatiques. Rendez-le facultatif et méritez son adoption. Mesurez les résultats des autres, comme le délai avant le premier déploiement, l'adoption, les métriques DORA et l'expérience des développeurs. L'objectif est d'avoir moins de tickets ; compter les clôtures récompense le mauvais comportement.

SOURCES

- Evan Bottcher: What I Talk About When I Talk About Platforms, martinfowler.com: https://martinfowler.com/articles/talk-about-platforms.html
- CNCF TAG App Delivery: Platforms White Paper: https://tag-app-delivery.cncf.io/whitepapers/platforms/
- Backstage documentation: What is Backstage?: https://backstage.io/docs/overview/what-is-backstage
- Forsgren et al.: The SPACE of Developer Productivity, ACM Queue (2021): https://queue.acm.org/detail.cfm?id=3454124

FinOps, architecture des coûts et dépendance au fournisseur

AVANT DE LIRE

Cette section s'appuie sur :

- 2.7 Sécurité, utilisabilité et autres qualités
- 5.6 Hexagonale et Clean Architecture
- 10.1 Modèles cloud

Termes déjà vus qu'elle utilise : Service managé, Multicloud, Requêtes et limites de ressources (requests and limits), Ports et adaptateurs (architecture hexagonale), Serverless

L'IDÉE EN BREF

Louer une voiture pour une journée ne coûte pas cher ; en louer une chaque jour pendant trois ans revient bien plus cher que de l'acheter. Laisser le moteur tourner sur le parking brûle du carburant pour rien. Et si vous prenez en location longue durée une voiture dotée d'une prise de recharge spéciale, changer de marque plus tard implique aussi de nouveaux chargeurs. Le cloud computing soulève ces trois questions à la fois. Le FinOps, c'est la bonne habitude du budget familial appliquée au cloud : voir ce que coûte chaque poste, réduire le gaspillage, acheter judicieusement. La dépendance au fournisseur, c'est la question de la prise spéciale : une certaine dépendance en vaut la peine, à condition de l'avoir choisie délibérément et de savoir comment vous en sortiriez.

CE QU'IL FAUT SAVOIR

- Dans le cloud, chaque décision d'architecture est aussi une décision d'achat : le coût est donc un attribut de qualité qui change à chaque déploiement et s'exprime au mieux par unité de valeur métier, comme le coût par commande.
- Le FinOps, défini par la FinOps Foundation (un projet de la Linux Foundation), partage la responsabilité financière entre l'ingénierie, la finance et le métier à travers trois phases itératives : informer, optimiser et opérer.
- Les facteurs de coût courants sont la sortie de données et les transferts entre zones, les ressources inactives et surdimensionnées, le volume de journalisation, les surcoûts des services managés et la croissance non maîtrisée du stockage ; la plupart sont déterminés par l'architecture.

- Les remises pour engagement ne conviennent qu'au socle certain, la capacité spot convient au travail interruptible et pouvant être relancé, et l'autoscaling à la demande couvre les pics ; redimensionnez avant de vous engager.
- L'étiquetage permet la répartition des coûts ; le showback présente leurs coûts aux équipes, le chargeback les impute à leur budget et exige une répartition exacte et des règles convenues pour les coûts partagés.
- La dépendance au fournisseur est un compromis : Hohpe (2019) en distingue de nombreuses formes et met en balance le coût de migration multiplié par sa probabilité et le coût de l'éviter ; acceptez-la quand elle apporte une valeur unique.
- La portabilité a des coûts réels ; gardez la logique métier derrière des ports aux points de jonction probables, gardez les données exportables et tenez à jour une stratégie de sortie testée, ce que la réglementation financière de l'UE exige désormais pour les services critiques.

Le coût est une qualité architecturale

La section 2.7 (sécurité, utilisabilité et autres qualités) a rangé le coût parmi les attributs de qualité et recommandé de l'exprimer par unité de travail. Dans un centre de données, le coût était surtout fixe : le matériel était acheté à quelques années d'intervalle et les choix d'un architecte faisaient à peine bouger la facture mensuelle. Dans le cloud, chaque décision d'architecture est aussi une décision d'achat, prise en continu par les ingénieurs plutôt qu'une fois par an par le service des achats. Une frontière de service bavarde, un niveau de journalisation debug, un environnement de test oublié ou un appel synchrone entre régions apparaissent chacun sur la facture du mois suivant. Le coût est devenu un attribut de qualité qui change à chaque déploiement.

Deux disciplines aident. Le FinOps est la pratique opérationnelle qui rend les dépenses cloud visibles, attribuées à des responsables et pilotées. L'architecture attentive aux coûts est la pratique de conception qui rend les systèmes peu coûteux à exploiter par unité de valeur dès le départ. Cette section traite des deux, puis se tourne vers leur voisin encombrant : la dépendance au fournisseur, c'est-à-dire le coût du départ.

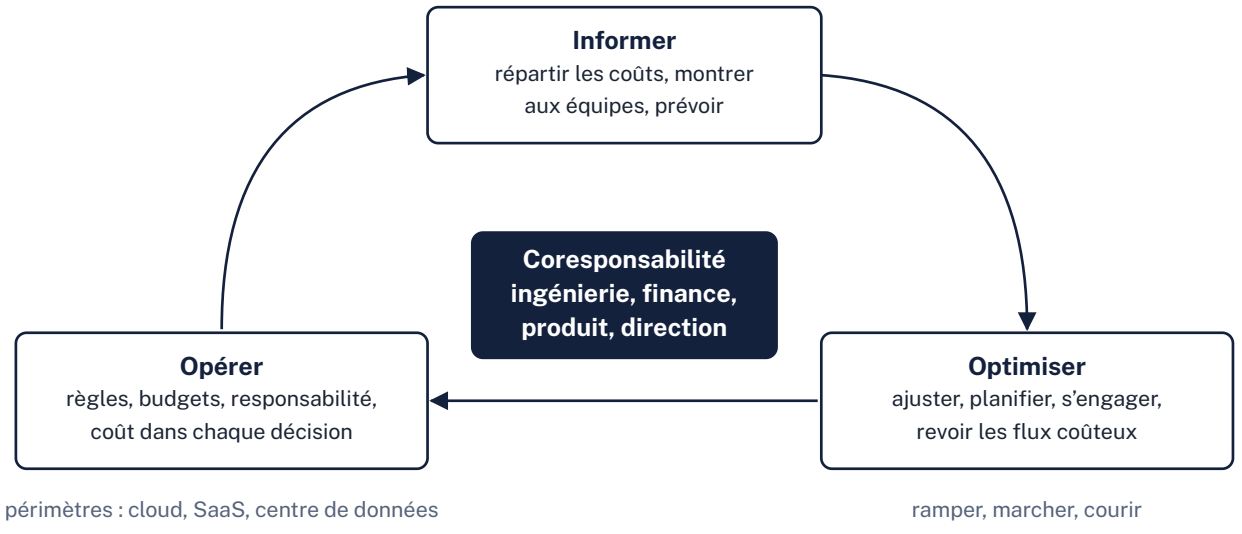
FinOps : phases, principes et qui fait quoi

Le FinOps est défini par la FinOps Foundation, un projet de la Linux Foundation, comme un cadre opérationnel et une pratique culturelle qui maximise la valeur métier de la technologie grâce à des décisions prises en temps utile et fondées sur les données, et à une responsabilité financière partagée entre l'ingénierie, la finance et le métier. Le nom fait délibérément écho au

DevOps : de même que le DevOps a abattu le mur entre la construction et l'exploitation des logiciels, le FinOps abat le mur entre la dépense d'argent et la décision sur la façon de construire les logiciels.

Le FinOps Framework organise le travail en trois phases itératives. **Informer** : rendre les coûts visibles, les répartir entre les équipes et les produits qui les génèrent, établir des prévisions et des comparaisons. **Optimiser** : réduire le gaspillage et améliorer les tarifs, par le redimensionnement, la planification, les engagements et les changements de conception. **Opérer** : instaurer les habitudes qui maintiennent les coûts sous contrôle, comme les budgets, les politiques, la responsabilité et la prise en compte du coût dans chaque décision de conception. Les organisations parcourent le cycle à plusieurs reprises, en progressant, pour chaque capacité, de « ramper » à « marcher » puis à « courir ». Parmi les principes du cadre : les équipes collaborent ; la valeur métier guide les décisions technologiques ; chacun assume la responsabilité de son usage de la technologie ; les données de coût sont accessibles, disponibles en temps utile et exactes ; le FinOps est soutenu de manière centralisée ; et le modèle de coût variable du cloud doit être mis à profit. Les versions récentes étendent le cadre au-delà du cloud public à d'autres dépenses technologiques, comme les licences SaaS, les centres de données et les services d'IA.

Le cycle FinOps



Le FinOps est un cycle répété, pas un projet ponctuel de réduction des coûts, et il repose sur une responsabilité partagée plutôt que sur une équipe centrale qui surveille les dépenses.

Économie unitaire : le coût par commande

Une facture cloud qui passe de 80 000 à 100 000 par mois peut être un désastre ou un triomphe, selon que l'activité a progressé de 5 % ou de 50 %. L'économie unitaire (unit economics) exprime le coût par unité de valeur métier : le coût par commande pour la

plateforme de livraison de repas, par client actif pour un service de streaming, par millier d'appels d'API pour une plateforme destinée aux développeurs. Elle distingue l'efficacité de la croissance, relie les décisions d'ingénierie aux résultats métier et donne aux équipes un objectif sur lequel elles peuvent agir.

La calculer correctement exige trois choses. D'abord, une **unité qui compte pour le métier**, choisie avec la finance et le produit, et non inventée par les ingénieurs. Ensuite, la **répartition des coûts** : les coûts directs (le calcul et la base de données du service des commandes) attribués au moyen d'étiquettes ou de namespaces, et les coûts partagés (le plan de contrôle Kubernetes, la pile d'observabilité, les outils de l'équipe plateforme) répartis selon une règle convenue, comme l'usage proportionnel. Enfin, une distinction entre **coût moyen et coût marginal** : le coût moyen par commande inclut des coûts fixes qui n'augmentent pas avec les commandes, si bien que le coût des mille commandes suivantes est généralement inférieur à ce que suggère la moyenne, ce qui compte pour fixer le prix d'une promotion ou prévoir la croissance.

SI VOUS DÉBUTEZ

Pas à pas : ce que coûtent les mille commandes suivantes

L'idée la plus difficile de cette section est la différence entre coût moyen et coût marginal. La moyenne répartit toute la facture sur l'ensemble des commandes ; le coût marginal est ce qu'ajoute une commande de plus. Comme une partie de la facture est fixe, les deux peuvent beaucoup différer. Suivez une promotion à travers les deux, avec des chiffres donnés à titre d'illustration.

- La facture cloud mensuelle de la plateforme de livraison de repas est de £96 000 pour 1,2 million de commandes : son coût par commande, la mesure d'économie unitaire que recommande cette section, est donc de £0,080. C'est la moyenne.
- Une partie de la facture ne varie pas avec les commandes : la base de données des commandes, dimensionnée à l'avance, les serveurs toujours allumés pour les heures creuses, le plan de contrôle Kubernetes et les outils de l'équipe plateforme. Disons que £36 000 sont fixes de cette manière et que £60 000 augmentent avec les commandes : surtout du calcul en autoscaling, du transfert de données et des journaux.
- Chaque commande supplémentaire ajoute donc environ £60 000 divisés par 1,2 million, soit £0,050 : c'est le coût marginal, bien inférieur à la moyenne.
- L'entreprise prévoit une promotion qui devrait apporter 200 000 commandes supplémentaires par mois. Au coût moyen, les servir semble coûter 200 000 x £0,080 = £16 000. Au coût marginal, cela revient à environ £10 000, tant que les commandes supplémentaires tiennent dans la capacité fixe déjà payée.
- Le coût marginal n'est pas toujours inférieur. Si la promotion poussait la base de données des commandes au-delà de sa taille actuelle, passer à une taille supérieure ajouterait une marche fixe, et les premières commandes au-delà de ce seuil pourraient coûter bien plus que la moyenne.
- **La leçon :** suivez l'efficacité avec le coût moyen par commande, mais évaluez le volume supplémentaire, comme une promotion ou une prévision de croissance, avec le coût marginal. Demandez-vous d'abord quels coûts varient avec les commandes et lesquels ne varient pas.

Où va l'argent : les facteurs de coût courants

La plupart des factures cloud surprenantes proviennent d'une poignée de facteurs, et presque tous sont déterminés par l'architecture plutôt que par la négociation de meilleurs prix.

Facteurs de coût courants et réponses architecturales

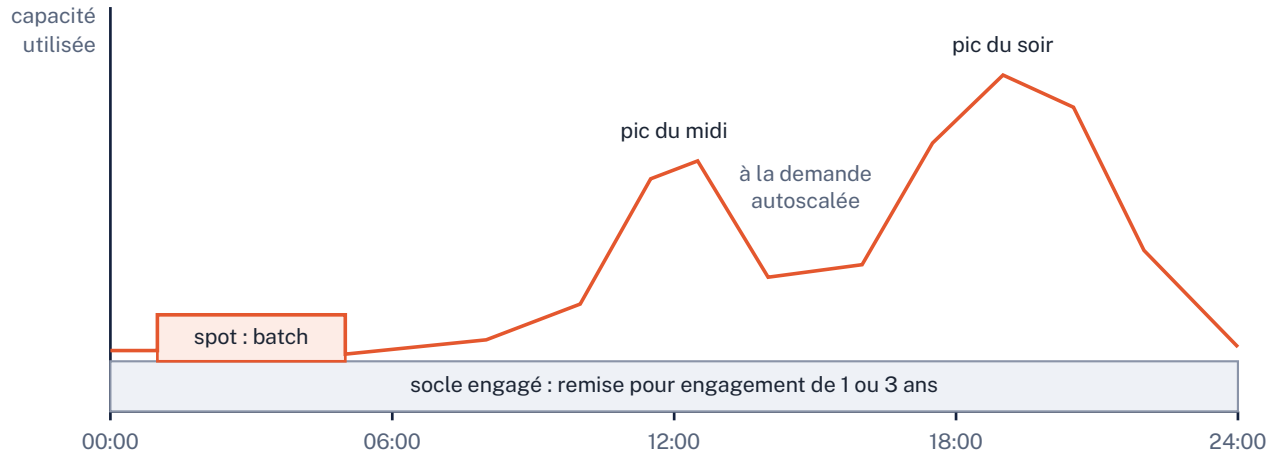
Facteur	Pourquoi il augmente	Réponses architecturales
Sortie de données (egress) et transferts	Les fournisseurs facturent au gigaoctet les données qui sortent vers Internet, et souvent le trafic entre zones et entre régions ainsi que celui qui passe par les passerelles NAT managées. Des services bavards répartis entre plusieurs zones paient à chaque appel.	Servir le contenu public via un CDN ; garder les services bavards dans une même zone (routage tenant compte de la topologie) ; utiliser des points de terminaison privés vers les services du fournisseur ; compresser ; éviter le trafic synchrone entre régions.
Ressources inactives et surdimensionnées	Des environnements de test qui tournent 168 heures par semaine pour 50 heures d'utilisation ; des requêtes de ressources de conteneurs fixées bien au-dessus de l'usage réel ; des volumes, instantanés et équilibreurs de charge orphelins.	Le redimensionnement (rightsizing) à partir de l'usage mesuré ; des plannings et un autoscaling jusqu'à zéro hors production ; des environnements de prévisualisation à durée limitée ; le nettoyage automatisé des ressources non rattachées.
Volume de journalisation et d'observabilité	L'ingestion et la conservation sont facturées au gigaoctet ; les journaux de débogage en production, la journalisation détaillée des requêtes et les labels de métriques à forte cardinalité augmentent avec le trafic.	Niveaux de journalisation et échantillonnage pour les requêtes réussies ; conservation par paliers (chaude pendant des jours, archivée pendant des mois) ; des métriques plutôt que des journaux pour les comptages ; échantillonnage des traces.
Surcoûts des services managés	Un service managé coûte plus cher par unité de capacité que d'exploiter le logiciel vous-même ; la tarification à la requête croît linéairement avec le volume.	Comparer le coût total de possession, en incluant les personnes et les risques ; réexaminer les services facturés à la requête quand le volume est élevé et stable (voir serverless) ; les utiliser là où le surcoût achète une vraie fiabilité ou une vraie rapidité.
Croissance du stockage	Les données jamais supprimées, les versions d'objets, les copies en double et les anciennes sauvegardes s'accumulent sans bruit.	Politiques de cycle de vie vers des paliers moins chers et suppression ; règles de conservation convenues avec le métier et conformes à la loi ; sauvegardes incrémentales et dédupliquées.

Payer moins pour la même capacité

Une fois le gaspillage éliminé, le tarif payé pour la capacité restante peut être abaissé. La capacité à la demande est la plus souple et la plus chère. Une remise pour engagement d'utilisation échange de la souplesse contre un prix : les Reserved Instances et Savings Plans d'AWS, les réservations et plans d'économies d'Azure et les committed use discounts de Google Cloud offrent des réductions substantielles (AWS annonce jusqu'à 72 % pour certains engagements) en échange d'un engagement sur un niveau d'utilisation ou de dépense pendant un ou trois ans. Les engagements ne doivent couvrir que le socle dont on aura certainement besoin pendant toute la durée ; s'engager sur la capacité de pointe transforme une remise en gaspillage.

La capacité spot (AWS Spot Instances, Azure Spot Virtual Machines, Google Cloud Spot VMs) vend de la capacité excédentaire avec de fortes remises, souvent de 60 % à 90 %, mais le fournisseur peut la récupérer avec un court préavis : deux minutes sur AWS, environ 30 secondes sur Azure et Google Cloud. Elle convient au travail sans état, interruptible et pouvant être relancé, comme le traitement par lots, les runners de CI, le rendu et certaines couches web sans état réparties sur de nombreux types d'instances. Elle ne convient pas aux bases de données ni à tout ce qui ne peut pas être arrêté en cours de tâche. L'architecture décide de la part d'une charge de travail qui peut utiliser le spot : les tâches idempotentes et dotées de points de reprise le peuvent ; les tâches longues et avec état ne le peuvent pas.

Adapter les modèles de tarification à la forme de la demande



Engagez-vous sur le plancher, utilisez l'autoscaling à la demande pour les pics, et reportez le travail interruptible sur la capacité spot aux heures creuses.

Étiquetage, showback et chargeback

Personne n'optimise un coût qu'il ne voit pas. La répartition commence par une politique d'étiquetage (tags ou labels) : chaque ressource porte un petit ensemble d'étiquettes obligatoires, comme l'équipe propriétaire, le service, l'environnement et le centre de coûts, imposées par des garde-fous dans le code d'infrastructure et les politiques d'organisation du cloud, car les étiquettes ajoutées à la main finissent vite par manquer. Pour Kubernetes, les coûts sont répartis par namespace et par label avec des outils comme le projet open source OpenCost ou les fonctions de répartition des coûts des fournisseurs eux-mêmes. La spécification FOCUS de la FinOps Foundation définit un format commun pour les données de facturation, afin que les coûts de plusieurs fournisseurs puissent être combinés.

Une fois les coûts répartis, une organisation choisit comment les utiliser. Le showback présente à chaque équipe ses coûts sans toucher à son budget ; il développe la prise de conscience et constitue le point de départ habituel. Le chargeback (refacturation interne) impute les coûts au budget de chaque équipe, ce qui crée des incitations plus fortes mais exige une répartition exacte, des règles convenues pour les coûts partagés et un moyen de traiter les litiges. Dans les deux cas, publiez la part des dépenses qui ne peut pas être répartie : une part non étiquetée supérieure à quelques pour cent signifie que les chiffres ne sont pas encore assez fiables pour fonder des décisions.

Les décisions architecturales qui déterminent les coûts

Les économies les plus importantes viennent de la conception, pas des remises, et elles sont les moins chères à obtenir avant que le système soit construit. Chacune des décisions suivantes a un profil de coût prévisible. **La granularité des services** : chaque saut réseau coûte du calcul, de la sérialisation et éventuellement du transfert de données, si bien que des services à grain fin répartis dans de nombreuses zones coûtent plus cher par requête qu'un monolithe modulaire. **La diffusion synchrone en éventail (fan-out)** : une requête qui appelle dix services maintient onze ensembles de ressources occupés pendant toute sa durée. **Le nombre d'environnements** : des copies complètes de la production pour chaque équipe et chaque branche multiplient la facture ; des environnements partagés avec isolation ou des prévisualisations éphémères coûtent bien moins cher. **La conservation et la réplication des données** : chaque copie et chaque région supplémentaires multiplient le stockage et les transferts. **Le cache et les CDN** : une réponse servie par le cache coûte généralement bien moins cher qu'un nouveau calcul, et une réponse servie par le CDN évite la sortie de données depuis l'origine. **Le modèle de calcul** : la tarification serverless à la requête est moins chère que la capacité permanente en dessous d'un volume de bascule, et plus chère au-dessus.

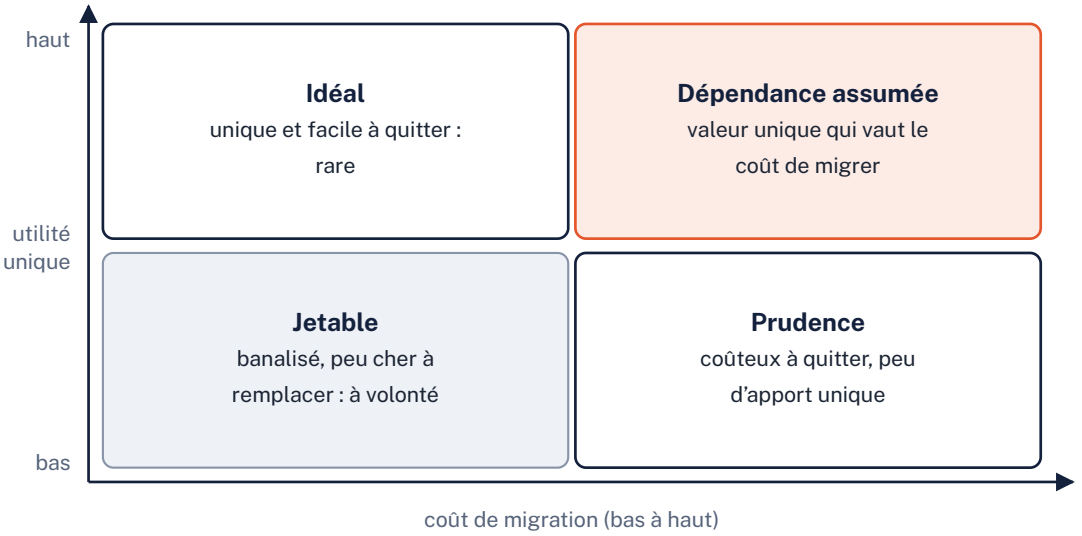
Consignez les conséquences importantes sur les coûts dans les enregistrements de décision, aux côtés des autres compromis de qualité, et envisagez d'ajouter le coût aux fonctions d'aptitude qui protègent l'architecture : par exemple, une vérification mensuelle que le coût par commande n'a pas augmenté de plus de 10 % sans explication.

La dépendance au fournisseur est un compromis, pas un péché

La dépendance au fournisseur (lock-in) est le coût et la difficulté de quitter quelque chose une fois que vous en dépendez. On en parle généralement comme d'une chose à éviter à tout prix ; l'article de Gregor Hohpe publié en 2019 sur martinfowler.com, *Don't get locked up into avoiding lock-in*, soutient que c'est en soi un piège. Il distingue plusieurs formes de dépendance : envers le fournisseur (contrats et conditions commerciales), le produit (les API et fonctionnalités d'un produit particulier, même open source), la version, l'architecture (un style comme le serverless), la plateforme (comptes, identité et modèles de sécurité), les compétences (ce que savent vos collaborateurs), le juridique (conformité et licences) et le mental (les présupposés qui vous empêchent d'envisager d'autres options). Utiliser des logiciels open source réduit la dépendance envers le fournisseur mais laisse subsister la plupart des autres.

L'essentiel est de traiter la dépendance comme un problème d'évaluation d'option. Le coût de migration (switching cost) est ce que coûterait un changement ; multiplié par la probabilité de changer un jour, il donne le coût attendu de la dépendance. Réduire le coût de migration a lui-même un prix : abstraction supplémentaire, conceptions au plus petit dénominateur commun et fonctionnalités auxquelles on renonce. Un changement qui coûterait 50 000 et a 5 % de chances de se produire représente un coût attendu de 2500 : dépenser bien plus que cela pour l'éviter est une mauvaise affaire. La matrice de Hohpe, qui croise l'utilité unique et le coût de migration, en fait une classification rapide.

Utilité unique et coût de migration



D'après Gregor Hohpe (2019) : la dépendance est acceptable quand elle apporte une valeur unique ; méfiez-vous des dépendances coûteuses qui rapportent peu en retour.

Le coût de la portabilité et la stratégie de sortie

La portabilité n'est pas gratuite. Concevoir chaque composant pour qu'il tourne sur plusieurs clouds signifie n'utiliser que les fonctionnalités qu'ils ont tous en commun, construire et tester des couches d'abstraction qui fuient (voir masquage de l'information et abstractions qui fuient), entretenir des compétences sur plusieurs plateformes et renoncer à des services managés qui auraient économisé des efforts. Beaucoup de stratégies multicloud finissent avec les coûts de tous leurs fournisseurs et les avantages d'aucun. La portabilité est la moins chère là où des standards ouverts existent déjà : images de conteneurs OCI, manifestes Kubernetes, bases de données compatibles PostgreSQL, API de stockage objet compatibles S3, protocoles standard comme HTTP, AMQP et le protocole Kafka, et OpenTelemetry pour l'observabilité. Même les outils utilisés pour rester portable, comme Terraform ou OpenTofu, sont des dépendances à part entière.

Les praticiens ne s'accordent pas sur jusqu'où aller. Les partisans de la portabilité invoquent les hausses de prix, les pannes, les sanctions, la réglementation et les cas où quitter le cloud a été payant (voir le rapatriement depuis le cloud à la section 10.1). Les partisans de l'engagement envers un seul fournisseur invoquent la rapidité et la fiabilité obtenues en utilisant pleinement les services managés. Une position intermédiaire solide consiste à accepter délibérément la dépendance là où elle apporte une valeur claire, à garder la logique métier indépendante des API du fournisseur derrière des ports et adaptateurs aux quelques points de jonction les plus susceptibles de changer, à conserver les données dans des formats exportables, et à mettre par écrit une stratégie de sortie.

La réglementation exige de plus en plus cette dernière étape. Le règlement européen sur la résilience opérationnelle numérique (Digital Operational Resilience Act, règlement (UE) 2022/2554, applicable depuis le 17 janvier 2025 ; à ne pas confondre avec les métriques de livraison DORA) impose aux entités financières de disposer de stratégies de sortie pour les services TIC qui soutiennent des fonctions critiques ou importantes. Les règles du règlement européen sur les données (Data Act) relatives au changement de fournisseur cloud s'appliquent depuis le 12 septembre 2025 : les clients peuvent changer de fournisseur avec un préavis de deux mois au maximum, et les fournisseurs ne peuvent facturer que leurs coûts directs de changement, sortie de données comprise, jusqu'au 12 janvier 2027, et plus rien ensuite. En 2024, les plus grands fournisseurs ont aussi annoncé la gratuité du transfert de données pour les clients qui les quittent, sous conditions. Une stratégie de sortie doit désigner le remplaçant de chaque service critique, estimer le temps et le coût de la migration, montrer que les données peuvent être exportées dans un format utilisable, et être répétée au moins en partie, par exemple en restaurant une sauvegarde dans un autre environnement.

EXEMPLE COMMENTÉ

Exemple commenté : coût par commande et trois changements architecturaux

La plateforme de livraison de repas traite 1,2 million de commandes par mois. Sa facture cloud mensuelle est de 96 000 : calcul 42 000, bases de données managées 18 000, observabilité 14 000, transfert de données 12 000, autres services managés 6000 et stockage 4000. Tous les chiffres sont donnés à titre d'illustration, en livres sterling.

Situation de départ	96 000 / 1 200 000 commandes = 0,080 par commande. Le showback révèle que les requêtes de calcul représentent environ le double de l'usage mesuré, que la moitié du volume de journaux est de la sortie de débogage, et que les services des commandes et de tarification échangent environ 40 appels par commande entre zones de disponibilité.
Changement 1 : redimensionner, puis s'engager	Des requêtes fixées à partir de l'usage mesuré au p95 réduisent le nombre de nœuds de 25 % : le calcul passe de 42 000 à 31 500. Les 60 % stables de ce montant (18 900) sont couverts par un engagement d'un an avec environ 30 % de remise, soit 5670 d'économie. Le calcul tombe à 25 830, soit une économie de 16 170.
Changement 2 : journaliser moins, conserver moins longtemps	Journalisation de débogage désactivée en production, requêtes réussies échantillonnées à 10 %, conservation à chaud réduite de 90 à 14 jours avec un palier d'archivage. L'ingestion baisse d'environ 60 % : l'observabilité passe de 14 000 à 6500, soit une économie de 7500.
Changement 3 : cesser de payer la distance	Logique de tarification déplacée dans le processus du service des commandes pour le chemin critique (ce qui supprime la plupart des 40 appels), routage tenant compte des zones pour les appels restants, et images des restaurants servies depuis un CDN plutôt que depuis l'origine. Transfert de données de 12 000 à 6000 ; le CDN ajoute 1000. Économie nette : 5000.
Résultat	96 000 moins 28 670 = 67 330 par mois, soit 0,056 par commande : 30 % de moins par commande, sans changement des fonctionnalités ni des objectifs de fiabilité.
Ce qui n'a pas changé	Les coûts des bases de données et du stockage n'ont pas été touchés ce trimestre, car les économies possibles y étaient faibles au regard du risque ; ils restent dans le backlog.

Le changement structurel le plus important était architectural (supprimer une frontière bavarde et introduire un CDN), le changement de journalisation était une décision de conception sur l'observabilité, et la remise tarifaire n'a été appliquée qu'après le redimensionnement, si bien que l'entreprise ne s'est pas engagée sur du gaspillage.

Suivre le coût par commande plutôt que la facture totale montre si la croissance future préserve l'amélioration : si les commandes doublent et que le coût par commande reste proche de 0,056, l'efficacité s'est maintenue.

Questions fréquentes

Faut-il facturer à chaque équipe ses coûts cloud ?
Commencez par le showback, pour que les équipes voient leurs coûts et puissent agir, et ne passez au chargeback que lorsque la répartition est exacte et que les coûts partagés sont répartis selon des règles acceptées par tous ; facturer des chiffres inexacts crée des disputes, pas des économies.

Le multicloud est-il la meilleure protection contre la dépendance au fournisseur ?
Rarement à lui seul. Faire tourner le même système sur plusieurs clouds coûte cher et vous limite aux fonctionnalités communes. Utiliser des fournisseurs différents pour des charges de travail différentes et indépendantes, avec une stratégie de sortie testée pour les charges critiques, apporte généralement l'essentiel de la protection pour une fraction du coût.

Quand l'optimisation des coûts est-elle prématurée ?
Quand le produit n'a pas trouvé son marché et que le temps d'ingénierie est la ressource la plus rare. Même alors, l'étiquetage dès le premier jour et un coût unitaire approximatif sont peu coûteux et facilitent beaucoup l'optimisation ultérieure.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Pourquoi le coût par commande est-il plus utile que la facture mensuelle totale ?**

Il distingue l'efficacité de la croissance : une facture en hausse avec un coût par commande en baisse est saine, alors qu'une facture stable avec des commandes en baisse ne l'est pas.

▼ **Pourquoi le redimensionnement doit-il précéder l'achat de remises pour engagement ?**

Un engagement fige un niveau de dépense pour un ou trois ans ; s'engager avant d'avoir éliminé le surdimensionnement transforme le gaspillage en obligation à long terme.

▼ **Quel type de charge de travail convient à la capacité spot, et pourquoi ?**

Le travail sans état, interruptible et pouvant être relancé, comme les traitements par lots et les runners de CI, parce que le fournisseur peut récupérer la capacité spot avec un préavis de quelques secondes ou minutes.

▼ **Selon Hohpe, quand la dépendance au fournisseur est-elle acceptable ?**

Quand elle apporte une utilité unique qui vaut le coût de migration, ou quand un changement est si improbable que le coût attendu est inférieur au coût d'éviter la dépendance.

▼ **À vous : La facture est de £60 000 par mois pour 1 500 000 commandes : calcul £30 000, journalisation £9000. Le redimensionnement (rightsizing) réduit le calcul de 30 % ; l'abandon des journaux de débogage divise la journalisation par deux. Trouvez le coût par commande avant et après. Lequel doit précéder toute remise pour engagement d'utilisation, et pourquoi ?**

Avant : £60 000 / 1 500 000 = 4,0 pence. Les économies sont de £9000 et £4500 : la facture tombe donc à £46 500, soit 3,1 pence. Commencez par le redimensionnement, car un engagement sur une capacité surdimensionnée fige le gaspillage pendant toute sa durée. Ne vous engagez ensuite que sur la base mesurée. Communiquez le coût par commande aux équipes par showback, car un total en hausse peut simplement signifier plus d'activité, alors qu'un coût unitaire en hausse signale un vrai problème.

SOURCES

- FinOps Foundation: FinOps Framework: https://www.finops.org/framework/
- Gregor Hohpe: Don't get locked up into avoiding lock-in, martinfowler.com: https://martinfowler.com/articles/oss-lockin.html
- AWS Well-Architected Framework: Cost Optimization Pillar: https://docs.aws.amazon.com/wellarchitected/latest/cost-optimization-pillar/welcome.html
- Regulation (EU) 2023/2854 (Data Act), EUR-Lex: https://eur-lex.europa.eu/eli/reg/2023/2854/oj

Lectures complémentaires

- *The NIST Definition of Cloud Computing (SP 800-145)*, Peter Mell and Timothy Grance (2011), National Institute of Standards and Technology. La définition courte et largement citée des caractéristiques essentielles, des modèles de services et des modèles de déploiement.
  https://csrc.nist.gov/pubs/sp/800/145/final
- *The Twelve-Factor App*, Adam Wiggins (2011), 12factor.net (Heroku; community-maintained since 2024). Douze pratiques concises pour des applications déployables et scalables ; elles restent le vocabulaire commun de la conception d'applications cloud native.
  https://12factor.net/
- *Kubernetes: Up and Running (3rd edition)*, Brendan Burns, Joe Beda, Kelsey Hightower and Lachlan Evenson (2022), O'Reilly Media. Une introduction claire aux objets et à l'exploitation de Kubernetes, par des personnes qui ont contribué à le créer.
- *Infrastructure as Code (3rd edition)*, Kief Morris (2025), O'Reilly Media. L'ouvrage de référence sur la conception, le test et la livraison du code d'infrastructure, indépendamment de tout outil particulier.
- *Continuous Delivery*, Jez Humble and David Farley (2010), Addison-Wesley. Le texte fondateur sur les pipelines de déploiement, le principe « build once, deploy many » et la banalisation des mises en production.
- *Accelerate: The Science of Lean Software and DevOps*, Nicole Forsgren, Jez Humble and Gene Kim (2018), IT Revolution. La recherche à l'origine des métriques DORA, qui relie les pratiques de livraison et une architecture faiblement couplée à la performance des organisations.
- *Site Reliability Engineering: How Google Runs Production Systems*, Betsy Beyer, Chris Jones, Jennifer Petoff and Niall Richard Murphy (eds) (2016), O'Reilly Media (free online). Comment un grand opérateur exploite ses systèmes grâce à l'automatisation, aux budgets d'erreur et à l'ingénierie des mises en production.
  https://sre.google/books/
- *Don't get locked up into avoiding lock-in*, Gregor Hohpe (2019), martinfowler.com. Une analyse équilibrée des différentes formes de dépendance au fournisseur et du coût réel de leur évitement.
  https://martinfowler.com/articles/oss-lockin.html

Glossaire

Actif-actif (multisite)
Organisation dans laquelle deux régions ou plus servent du trafic réel en même temps, de sorte que la perte d'une région ne nécessite aucun failover, seulement une redistribution du trafic.

Analyse canari automatisée
Comparaison statistique des métriques d'un canari avec celles d'une référence exécutant l'ancienne version, utilisée par le pipeline pour promouvoir ou annuler la version sans jugement humain.

Application douze facteurs (twelve-factor app)
Méthodologie de douze pratiques pour construire des applications web déployables et capables de passer à l'échelle, rédigée par Adam Wiggins chez Heroku en 2011 et publiée en open source en 2024.

Architecture cellulaire (cell-based architecture)
Exploitation de nombreuses copies complètes et indépendantes (cellules) d'un système, chacune servant un sous-ensemble fixe de clients, derrière un routeur léger, de sorte qu'une défaillance n'affecte qu'une seule cellule.

Boucle de réconciliation
Le cycle sans fin d'un contrôleur qui observe l'état réel, le compare à l'état souhaité et agit pour supprimer l'écart.

Build once, deploy many (construire une fois, déployer partout)
Règle selon laquelle une modification est construite une seule fois en un artefact immuable, et ce même artefact est promu dans chaque environnement, seule la configuration changeant.

Build, release, run (assembler, publier, exécuter)
Séparation stricte entre la transformation du code en artefact (build), sa combinaison avec la configuration d'un environnement (release) et son exécution (run).

Cadre SPACE
Cadre de 2021 pour la productivité des développeurs, qui couvre la satisfaction et le bien-être, la performance, l'activité, la communication et la collaboration, ainsi que l'efficacité et le flux.

Canary release (déploiement canari)
Envoi d'abord d'une petite part du trafic réel vers une nouvelle version, comparaison de son comportement avec celui de l'ancienne version, et élargissement de cette part uniquement si elle est saine.

Capacité spot
Capacité cloud excédentaire vendue avec une forte remise, que le fournisseur peut reprendre à bref délai ; ne convient qu'aux travaux qui tolèrent une interruption.

Charge cognitive
Quantité d'effort mental qu'une personne ou une équipe doit fournir pour comprendre un système et travailler dans son cadre ; Team Topologies s'en sert pour dimensionner les responsabilités des équipes.

Chargeback (refacturation interne)
Imputation au budget de chaque équipe ou produit des coûts qui lui sont attribués, afin que les dépenses apparaissent dans ses propres comptes.

Cloud hybride
Combinaison de cloud privé et de cloud public (ou de systèmes sur site) reliés de sorte que les données et les applications puissent fonctionner de l'un à l'autre.

Cloud native
Approche consistant à construire et exploiter des logiciels sous forme de systèmes faiblement couplés, automatisés, observables et résilients, déployés de manière programmatique et reproductible, dans des clouds publics, privés ou hybrides. Elle décrit comment un système est construit, et non où il s'exécute.

Cloud privé
Infrastructure cloud dédiée à une seule organisation, offrant un provisionnement en libre-service piloté par API ; elle peut être sur site ou hébergée par un tiers.

Conteneurs en tant que service (CaaS)
Service cloud qui exécute des images de conteneur pour vous, soit sous forme de Kubernetes managé, soit sous forme de plateforme de conteneurs serverless où vous ne voyez jamais les machines.

Coût de migration (switching cost)
L'effort, l'argent et le risque totaux nécessaires pour passer d'une technologie ou d'un fournisseur à un autre.

Dépendance au fournisseur (lock-in)
Le coût et la difficulté de quitter un fournisseur, un produit, une plateforme ou une technologie une fois qu'un système en dépend.

Déploiement bleu-vert (blue-green)
Exploitation de deux environnements de production complets et bascule de tout le trafic de l'ancien (bleu) vers le nouveau (vert) en une seule fois, en gardant l'ancien prêt pour un retour instantané.

Déploiement progressif (rolling deployment)
Remplacer les instances d'un service quelques-unes à la fois, de sorte que l'ancienne et la nouvelle version tournent côte à côte jusqu'à ce que toutes les instances exécutent la nouvelle version.

Deployment (Kubernetes)
Objet Kubernetes qui déclare combien de pods identiques d'une application sans état doivent s'exécuter et qui déploie progressivement les nouvelles versions au moyen de ReplicaSets.

Dépôt d'artefacts
Stockage pour des produits construits et versionnés, comme des images de conteneur et des paquets, à partir duquel chaque environnement déploie le même fichier.

Dérive de configuration
Écarts qui s'accumulent entre la configuration prévue dans le code et l'infrastructure réelle, généralement à cause de modifications manuelles ou de mises à jour partielles.

Développement basé sur le tronc (trunk-based development)
Façon de travailler dans laquelle les développeurs intègrent de petits changements dans une branche principale partagée au moins une fois par jour, au lieu de conserver des branches de fonctionnalité de longue durée.

Économie unitaire (unit economics)
Exprimer le coût (et le chiffre d'affaires) par unité de valeur métier, comme le coût par commande ou par client actif, afin de pouvoir juger l'efficacité indépendamment de la croissance.

État (infrastructure en tant que code)
L'enregistrement que tient un outil d'infrastructure des ressources réelles correspondant à chaque définition du code, utilisé pour déterminer ce qu'il faut créer, modifier ou supprimer.

État souhaité
Description déclarative de ce à quoi un système doit ressembler (par exemple, trois copies de la version 2.4), que l'automatisation compare à l'état réel et s'efforce d'atteindre.

FinOps
Pratique dans laquelle les équipes d'ingénierie, de finance et métier partagent la responsabilité des dépenses cloud et technologiques, en s'appuyant sur des données de coûts à jour pour prendre des décisions qui maximisent la valeur métier.

Fonction en périphérie (edge function)
Petit morceau de code exécuté par un CDN ou un fournisseur edge à des emplacements proches des utilisateurs, pour traiter ou modifier les requêtes avant qu'elles n'atteignent l'origine (ou à sa place).

Garde-fou (guardrail)
Limite ou vérification automatique qui garantit la sûreté de l'usage en libre-service (par exemple, une politique qui bloque les buckets de stockage publics) sans exiger qu'une personne approuve chaque demande.

Gateway API
API de Kubernetes pour acheminer le trafic externe et interne vers les services, disponible en version stable depuis 2023 et successeur recommandé d'Ingress.

Gestion de configuration
Outils comme Ansible, Chef et Puppet qui amènent des serveurs existants et de longue durée dans une configuration décrite, en y installant des paquets et en modifiant des fichiers.

GitOps
Exploitation des systèmes en conservant leur état souhaité déclaratif dans Git et en laissant des agents le récupérer et réconcilier en continu le système en cours d'exécution pour qu'il y corresponde.

Golden path (paved road, chemin balisé)
Manière préconisée, prise en charge et documentée d'accomplir une tâche courante, comme créer un nouveau service, qui est la plus simple mais pas la seule autorisée.

Horizontal Pod Autoscaler (HPA)
Contrôleur Kubernetes qui modifie le nombre de répliques de pods pour maintenir une métrique, comme l'utilisation moyenne du CPU rapportée aux requêtes de ressources, près d'une cible.

Infrastructure en tant que code (IaC)
Définition des serveurs, réseaux, bases de données et autres éléments d'infrastructure dans des fichiers placés sous gestion de versions, que des outils appliquent automatiquement, au lieu de les configurer à la main.

Infrastructure immuable
Pratique consistant à ne jamais modifier sur place un serveur ou un conteneur en cours d'exécution ; chaque modification se fait en construisant une nouvelle image et en remplaçant les anciennes instances.

Ingénierie de plateforme (platform engineering)
La discipline qui consiste à construire et à exploiter des plateformes internes permettant aux équipes produit de provisionner, de déployer et d'exploiter des logiciels en libre-service.

Jetabilité (disposability)
Propriété des processus qui démarrent rapidement, s'arrêtent proprement et peuvent être tués à tout moment sans dommage, de sorte qu'on peut les remplacer et les multiplier librement.

Kubernetes
L'orchestrateur de conteneurs open source le plus utilisé, créé chez Google, annoncé en 2014 et publié en version 1.0 en 2015 ; il maintient l'état réel d'un cluster conforme à un état souhaité déclaré.

Livraison progressive (progressive delivery)
Mettre un changement en production pour des publics de plus en plus larges, avec des vérifications automatisées à chaque étape, afin qu'un mauvais changement atteigne le moins d'utilisateurs possible avant d'être arrêté.

Modèle de responsabilité partagée
La répartition des obligations de sécurité et d'exploitation entre un fournisseur de cloud, qui sécurise le cloud lui-même, et le client, qui sécurise ses données, ses identités et sa configuration dans le cloud.

Multicloud
Utiliser les services de deux fournisseurs de cloud public ou plus, pour des services qui leur sont propres, à la suite d'acquisitions, pour peser dans les négociations ou pour des plans de sortie exigés par la réglementation.

Open Container Initiative (OCI)
Projet de la Linux Foundation, fondé en 2015, qui maintient des normes ouvertes pour les formats d'images de conteneurs, les environnements d'exécution (runtimes) et les registres, afin que les outils de différents fournisseurs soient interopérables.

OpenTofu
Fork open source de Terraform créé en 2023 après le passage de Terraform sous Business Source Licence par HashiCorp ; hébergé par la Linux Foundation et projet de la CNCF.

Orchestration de conteneurs
Placement automatique des conteneurs sur des machines et maintien de leur fonctionnement : redémarrage après défaillance, mise à l'échelle, déploiement progressif de nouvelles versions et connexion aux réseaux et au stockage.

Pilot light (veilleuse)
Stratégie de reprise après sinistre qui maintient les données répliquées et l'infrastructure essentielle provisionnée dans une seconde région, mais ne démarre et ne met à l'échelle les serveurs d'application qu'en cas de sinistre.

Pipeline de déploiement
Suite automatisée d'étapes (build, tests, empaquetage, déploiement, vérification) par laquelle passe chaque modification entre un commit et la production.

Plan de contrôle (control plane)
Composants qui prennent les décisions pour un cluster ou un système (dans Kubernetes : le serveur d'API, etcd, l'ordonnanceur et les contrôleurs), par opposition aux nœuds qui exécutent les charges de travail.

Plateforme interne de développement (IDP)
Ensemble d'outils, de services, de modèles et de documentation en libre-service que l'équipe plateforme d'une organisation fournit afin que les équipes produit puissent construire et exploiter des logiciels sans passer par des tickets.

Plateforme viable la plus fine (thinnest viable platform)
Le plus petit ensemble de services partagés et de documentation qui accélère réellement les équipes produit ; la mise en garde de Team Topologies contre le fait de construire plus de plateforme que nécessaire.

Pod
La plus petite unité déployable dans Kubernetes : un ou plusieurs conteneurs planifiés ensemble sur un même nœud, qui partagent une adresse réseau et des volumes de stockage.

Politique en tant que code (policy as code)
Écrire les règles portant sur l'infrastructure et les déploiements (pas de stockage public, chiffrement obligatoire, régions approuvées) sous forme de code que des outils vérifient automatiquement avant ou pendant les changements.

Portail des développeurs
Interface web, comme Backstage, qui rassemble en un seul endroit un catalogue de logiciels, des modèles, de la documentation et des liens vers les services de la plateforme.

Rapatriement depuis le cloud (cloud repatriation)
Déplacement de charges de travail hors d'un cloud public vers du matériel détenu en propre ou hébergé en colocation, généralement parce qu'une charge stable rend la propriété moins chère.

Redimensionnement (rightsizing)
Ajuster la taille des ressources provisionnées (types de machines, requêtes des conteneurs, niveaux de base de données) à ce qu'une charge de travail utilise réellement.

Remise pour engagement d'utilisation
Prix réduit en échange d'un engagement sur un niveau d'utilisation ou de dépense pendant un ou trois ans, comme les instances réservées, les savings plans ou les committed use discounts.

Reprise après sinistre (DR)
Plans, copies et procédures qui rétablissent un service et ses données après une défaillance de grande ampleur, comme la perte d'un centre de données ou d'une région entière.

Requêtes et limites de ressources (requests and limits)
Réglages de CPU et de mémoire propres à chaque conteneur dans Kubernetes : les requêtes (requests) réservent de la capacité et guident la planification ; les limites (limits) plafonnent la consommation, en bridant le CPU et en arrêtant les conteneurs qui dépassent leur mémoire.

Roll forward (correction vers l'avant)
Se remettre d'une mauvaise version en déployant un nouveau correctif plutôt qu'en revenant à la version précédente.

Service (Kubernetes)
Objet Kubernetes qui donne à un ensemble changeant de pods un nom et une adresse virtuelle stables, et qui répartit le trafic entre eux.

Service externe (backing service)
Tout service qu'une application utilise par le réseau, comme une base de données, une file, un cache ou une API d'e-mail, traité comme une ressource attachée que la configuration permet de remplacer.

Showback
Communiquer à chaque équipe ou produit les coûts qui lui sont attribués, sans les imputer réellement à son budget.

Shuffle sharding
Affecter chaque client à une petite combinaison de ressources choisie au hasard, de sorte que deux clients partagent rarement toutes leurs ressources et qu'un client problématique n'en affecte que peu d'autres.

Sonde (liveness, readiness, startup)
Vérification périodique d'un conteneur par Kubernetes : la sonde liveness décide s'il faut le redémarrer, la sonde readiness s'il faut lui envoyer du trafic, la sonde startup s'il a fini de démarrer.

Sortie de données (egress)
Données qui quittent le réseau d'un fournisseur cloud, ou qui passent d'une de ses zones ou régions à une autre, généralement facturées au gigaoctet.

Stabilité statique
Propriété selon laquelle un système continue de fonctionner pendant une défaillance sans avoir à effectuer de changements, comme lancer de nouveaux serveurs ou mettre à jour la configuration, au moment de la défaillance.

Stratégie de sortie
Plan documenté, et idéalement répété, pour quitter un fournisseur ou un produit, couvrant l'export des données, les services de remplacement, les contrats, les délais et les coûts.

Trafic fantôme (shadow traffic, dark launch)
Copier des requêtes réelles vers une nouvelle version dont les réponses sont comparées ou écartées, mais jamais renvoyées aux utilisateurs, afin de la tester sous charge réelle sans impact pour les utilisateurs.

Warm standby (secours tiède)
Stratégie de reprise après sinistre qui maintient en fonctionnement, dans une seconde région, une copie réduite mais pleinement opérationnelle du système, prête à recevoir le trafic et à monter en charge.