BE SOFTWARE ARCHITECTURE WISE · CHAPITRE 11

Évaluation, prise de décision et gouvernance

Ce chapitre porte sur l'évaluation, la gouvernance et l'évolution des architectures dans le temps. Il couvre l'évaluation formelle avec l'ATAM, les revues légères comme le risk-storming et les pre-mortems, les fonctions d'aptitude et l'architecture évolutive, la gouvernance dans les organisations agiles, Team Topologies et la conception sociotechnique, la modernisation des systèmes hérités avec le strangler fig, le leadership et l'influence, ainsi que la justification économique et éthique du travail d'architecture. Il fait la synthèse de tous les chapitres précédents, utilise la plateforme de livraison de repas comme exemple fil rouge et s'adresse aux développeurs expérimentés, aux responsables techniques et aux architectes qui prennent ou orientent des décisions importantes.

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

EN UN COUP D'ŒIL

Vous apprendrez à

- Expliquer quand et pourquoi évaluer une architecture, et dérouler les neuf étapes de l'ATAM pour produire des risques, des non-risques, des points de sensibilité, des points de compromis et des thèmes de risque.
- Animer des revues légères comme le risk-storming et les pre-mortems, et transformer leurs résultats en un registre des risques avec des responsables.
- Concevoir des fonctions d'aptitude qui transforment les scénarios d'attributs de qualité en vérifications automatisées de la structure, des performances, de la sécurité, de la résilience et des données.
- Rédiger des principes d'architecture et choisir entre garde-fous, points de validation et processus de consultation pour gouverner différents types de décisions.
- Appliquer les types d'équipes, les modes d'interaction et la charge cognitive de Team Topologies pour aligner les équipes sur l'architecture visée.
- Planifier une modernisation incrémentale d'un système hérité avec le strangler fig, le branch by abstraction et les exécutions en parallèle.
- Mener et influencer les décisions architecturales sans autorité formelle, y compris dans la communication avec les dirigeants.
- Calculer la valeur actuelle nette et le délai de récupération pour une analyse de rentabilité, et examiner une architecture sous l'angle des préoccupations éthiques et de durabilité.

Avant de commencer

Les chapitres 1 à 4, en particulier les décisions et les compromis (section 1.6), la dette technique (section 1.7), les scénarios d'attributs de qualité et les arbres d'utilité (sections 2.3 et 2.8), la loi de Conway (section 3.7) et les enregistrements de décision (section 4.5). Les chapitres 5 à 10 aident, mais ne sont pas indispensables.

Durée

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

Termes clés

ATAM · Arbre d'utilité · Point de sensibilité · Point de compromis · Risk-storming · Pre-mortem · Fonction d'aptitude (fitness function) · Architecture évolutive · Processus de consultation (advice process) · Radar technologique · Team Topologies · Strangler fig (figuier étrangleur) · Valeur actuelle nette · Software Carbon Intensity

Juger, gouverner et faire évoluer l'architecture

Les chapitres précédents ont montré comment décrire ce qu'un système doit accomplir, comment le structurer et comment le rendre rapide, fiable, sûr et économique. Ce chapitre porte sur le travail qui entoure ces décisions pendant toute la vie d'un système : vérifier qu'elles sont solides avant d'engager l'argent, les maintenir vraies alors que des centaines de

changements arrivent chaque mois, décider qui peut prendre quelles décisions dans une organisation de nombreuses équipes, remodeler les équipes et les systèmes hérités, convaincre, et justifier le coût du travail d'architecture en termes économiques et éthiques. C'est là que le cours se rassemble ; il renvoie donc souvent aux chapitres précédents : aux scénarios d'attributs de qualité et aux arbres d'utilité du chapitre 2, aux principes et aux métriques du chapitre 3, aux enregistrements de décision du chapitre 4, à la loi de Conway de la section 3.7 et à la dette technique de la section 1.7.

Le niveau est avancé, mais chaque section commence par une courte explication pour les débutants, et l'exemple fil rouge est la plateforme de livraison de repas utilisée depuis le chapitre 2 : des clients, des restaurants et des livreurs, des services de commandes, d'affectation des livreurs et de notifications, un courtier d'événements et un prestataire de paiement externe. Vous verrez son architecture évaluée, soumise au risk-storming, protégée par des fonctions d'aptitude, gouvernée par des principes et un radar technologique, réorganisée autour des équipes et en partie modernisée.

Évaluer et protéger les décisions : sections 11.1 à 11.3

La section 11.1 traite de l'évaluation de l'architecture formelle et de l'ATAM du SEI : pourquoi et quand évaluer, ses neuf étapes réparties sur deux réunions, qui y participe, et ses résultats caractéristiques : les risques, les non-risques avec leurs hypothèses, les points de sensibilité, les points de compromis et les thèmes de risque. L'arbre d'utilité de la section 2.8 revient dans son cadre d'origine, et la CBAM montre comment mettre en balance le bénéfice de chaque correction et son coût. Une ATAM condensée de la plateforme de livraison de repas montre les tableaux que produit réellement une évaluation.

La section 11.2 explique pourquoi les comités de revue mensuels échouent si souvent et ce qui fonctionne à la place : le risk-storming de Simon Brown, le pre-mortem de Gary Klein, les katas d'architecture de Ted Neward, TARA d'Eoin Woods pour les systèmes existants, les revues de conception écrites et les listes de contrôle. Elle se termine par un format pas à pas pour une revue de 90 minutes et par un guide pour choisir entre les techniques.

La section 11.3 transforme une évaluation ponctuelle en protection continue. Une fonction d'aptitude (fitness function) est une vérification objective d'une caractéristique architecturale, idéalement automatisée : une règle de dépendance, un test de charge avec des seuils de percentiles, une analyse de vulnérabilités, une expérience de chaos ou une vérification de compatibilité de schéma. Associées à des changements petits, fréquents et sûrs, les fonctions d'aptitude rendent possible une architecture évolutive. Trois scénarios du chapitre 2 deviennent des vérifications opérationnelles, avec le code.

Gouverner, organiser et moderniser : sections 11.4 à 11.6

La section 11.4 se demande comment une organisation de nombreuses équipes autonomes garde une architecture cohérente. Elle distingue principes, normes, garde-fous et points de validation ; montre comment rédiger des principes avec un nom, un énoncé, une justification et des implications ; et décrit la prise de décision décentralisée grâce au processus de consultation (advice process) d'Andrew Harmel-Law, aux forums consultatifs d'architecture et aux radars technologiques, la conformité étant automatisée par des fonctions d'aptitude et la politique en tant que code (policy as code).

La section 11.5 prend au sérieux la loi de Conway comme outil de conception. Elle présente Team Topologies de Matthew Skelton et Manuel Pais : quatre types d'équipes, trois modes d'interaction, la charge cognitive de l'équipe, les API d'équipe et les plans de fracture, et montre comment réorganiser les équipes de la plateforme de livraison de repas autour de flux de changement. La section 11.6 traite des systèmes qui existent déjà : pourquoi les réécritures big bang échouent si souvent, l'éventail des stratégies de migration et les techniques incrémentales qui fonctionnent, dont le strangler fig, le branch by abstraction, les exécutions en parallèle, la capture des données modifiées et les couches anticorruption, mises en pratique en extrayant un module de commandes d'un monolithe à la manière d'un figuier étrangleur.

Diriger, justifier et agir de manière responsable : sections 11.7 et 11.8

La section 11.7 couvre la dimension humaine du rôle : diriger sans autorité formelle, bâtir la confiance, faciliter les décisions par consentement ou par consensus, gérer les conflits et la politique interne, communiquer avec les dirigeants, faire du mentorat, les parcours d'ingénierie staff-plus et les pièges de décision qui guettent les personnes expérimentées, avec une décision contestée sur un bus d'événements menée pas à pas.

La section 11.8 clôt le chapitre avec les deux argumentaires qu'un architecte doit le plus savoir bien construire. L'analyse de rentabilité : coûts, bénéfices, risques et options, avec la valeur actuelle nette et le délai de récupération calculés dans un exemple commenté, et la dette technique exprimée en argent et en temps. Et la justification éthique : codes de conduite professionnels, préjudices prévisibles comme la perte de vie privée, les biais, l'inaccessibilité et la conception manipulatrice, et la durabilité mesurée avec la spécification Software Carbon Intensity.

L'idée à retenir

L'architecture n'est pas terminée quand le diagramme est dessiné. Ses décisions doivent être confrontées aux besoins du métier, protégées pendant que le système évolue, prises par les bonnes personnes avec les bons conseils, portées par des équipes dont la structure leur

correspond, et justifiées dans des termes que les commanditaires et la société peuvent accepter. Les architectes les plus efficaces rendent ce travail léger et continu plutôt que lourd et ponctuel : de courtes revues des risques au lieu d'audits annuels, des garde-fous automatisés au lieu de files d'approbation, des conseils au lieu d'autorisations, une migration incrémentale au lieu de réécritures. La discipline qui relie le tout est celle par laquelle ce cours a commencé : nommer la qualité dont vous avez besoin, la rendre mesurable, décider délibérément, écrire pourquoi, et continuer à vérifier.

SECTION 11.1 SUR 11.8 · AVANCÉ · 28 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Évaluation de l'architecture et ATAM

AVANT DE LIRE

Cette section s'appuie sur :

- 1.6 Décisions et compromis
- 2.3 Scénarios d'attributs de qualité
- 2.8 Éliciter et hiérarchiser

Termes déjà vus qu'elle utilise : Scénario d'attribut de qualité, Arbre d'utilité, Attribut de qualité, Partie prenante

L'IDÉE EN BREF

Avant qu'une ville construise un nouveau pont, des ingénieurs indépendants vérifient la conception : ils demandent ce qui se passe en cas de crue, sous les camions les plus lourds, pendant une tempête, et ils signalent les endroits où un choix qui fait économiser de l'argent aggrave un autre danger. Ils ne redessinent pas le pont et ne lui donnent pas une note sur dix ; ils indiquent aux propriétaires où se trouvent les vrais risques, pour que ceux-ci puissent décider quoi faire tant que les changements coûtent encore peu. L'évaluation de l'architecture fait la même chose pour le logiciel. Sa méthode la plus connue, l'ATAM, réunit les propriétaires, les bâtisseurs et les utilisateurs du système pour confronter la conception aux situations qui comptent le plus.

CE QU'IL FAUT SAVOIR

- L'évaluation de l'architecture vérifie si les décisions significatives d'un système lui permettront d'atteindre ses objectifs métier et ses exigences de qualité ; elle a le plus de valeur avant des engagements majeurs ou irréversibles.
- L'ATAM (Kazman, Klein et Clements, SEI, 2000) se déroule en neuf étapes réparties entre la préparation, deux réunions d'évaluation et le suivi, avec une équipe d'évaluation indépendante, les décideurs du projet et un groupe plus large de parties prenantes.
- Ses résultats sont les risques, les non-risques (avec leurs hypothèses), les points de sensibilité, les points de compromis et les thèmes de risque reliés aux facteurs métier qu'ils menacent ; elle n'attribue pas de note de réussite.
- L'arbre d'utilité construit par les décideurs à l'étape 5 est comparé aux scénarios que les parties prenantes proposent et soumettent au vote à l'étape 7, et les écarts entre les deux révèlent des angles morts.

- La CBAM prolonge l'ATAM en pondérant l'utilité que chaque stratégie architecturale ajoute aux scénarios priorisés et en la divisant par son coût pour comparer le rapport valeur-coût, tandis que des contraintes peuvent toujours l'emporter sur ce ratio.
- Une ATAM complète coûte des dizaines de jours-personnes ; la méthode doit donc être adaptée aux enjeux, et ses constats doivent devenir des enregistrements de décision, des éléments de backlog et des fonctions d'aptitude dotés de responsables.

Pourquoi évaluer une architecture

Toute architecture est un pari. Quand une équipe choisit un courtier d'événements, une base de données par service ou une seule région cloud, elle prédit que ces décisions apporteront les performances, la disponibilité, la sécurité et le coût dont le métier a besoin. La section 1.3 a montré pourquoi de tels paris comptent : le coût de changement d'une décision structurelle augmente avec tout ce qui est construit par-dessus, et certaines des défaillances les plus coûteuses du logiciel tenaient à l'agencement plutôt qu'à des lignes de code isolées. L'évaluation de l'architecture est l'acte délibéré de vérifier le pari avant que l'argent soit dépensé, ou pendant qu'il l'est.

Une évaluation pose une question étroite et pratique : compte tenu de ce que ce système doit accomplir, lesquelles de ses décisions architecturales mettent ces objectifs en danger ? Ce n'est ni une revue de code, ni un concours de beauté entre styles, ni un audit de la qualité de la documentation. Elle porte sur les décisions et leurs conséquences pour les attributs de qualité ; elle peut donc être menée bien avant qu'il existe du code, sur une conception au tableau blanc ou un ensemble d'enregistrements de décision, et elle peut être répétée sur un système qui fonctionne depuis des années.

L'évaluation a aussi des bénéfices au-delà de ses constats. Des parties prenantes qui se rencontrent rarement (responsables produit, exploitation, sécurité, finance, développeurs) doivent s'accorder sur ce qui compte le plus, ce qui met souvent au jour des hypothèses contradictoires. L'architecte doit expliquer la conception assez clairement pour que des personnes extérieures puissent l'examiner, ce qui améliore la documentation. Et l'organisation obtient une trace des risques connus qu'elle a choisi d'accepter, ce qui est très différent de risques que personne n'a remarqués.

Quand évaluer

Le meilleur moment pour évaluer est celui où suffisamment de choses ont été décidées pour que l'évaluation ait de quoi examiner, mais avant que ces décisions deviennent coûteuses à défaire. En pratique, il existe plusieurs moments naturels, et les organisations matures utilisent pour chacun une évaluation plus ou moins lourde.

Les moments de l'évaluation

Déclencheur	Ce qui est vérifié	Poids habituel
Une architecture candidate existe, avant la construction à grande échelle	Si les principaux styles et tactiques peuvent satisfaire les scénarios de qualité priorisés	ATAM complète ou réduite pour les systèmes importants ou à longue durée de vie ; sinon une revue légère
Avant un engagement majeur	Un contrat fournisseur, une migration vers le cloud, une étape de financement ou une décision de changement de plateforme coûteuse à défaire	Formelle, souvent avec des évaluateurs externes, car la décision est une porte à sens unique
Changement important des facteurs métier	Nouveaux marchés, fusion, objectif de croissance d'un facteur dix, nouvelle réglementation	Refaire l'arbre d'utilité et analyser les scénarios qui ont changé
Changements accumulés et dérive	Si le système tel que construit correspond encore à l'architecture voulue et à ses raisons	Revue légère périodique plus fonctions d'aptitude automatisées
Acquisition ou due diligence	Si un système existant peut soutenir les projets de l'acquéreur	Évaluation limitée dans le temps d'un système existant, souvent combinée à une analyse du code
Après un incident grave	Si l'incident a révélé une faiblesse architecturale plutôt qu'une faute isolée	Analyse ciblée des scénarios concernés

Évaluer trop tôt gaspille des efforts sur des décisions qui ne sont pas encore réelles ; évaluer trop tard transforme les constats en regrets. Une règle utile du modèle piloté par les risques de Fairbanks (section 1.8) s'applique ici aussi : investissez dans l'évaluation en proportion du risque. Un outil interne utilisé par deux équipes n'a pas besoin d'un atelier de deux jours ; une plateforme nationale de paiements, si.

> **SI VOUS DÉBUTEZ**
>
> #### Si vous débutez : ce dont une évaluation a besoin en entrée
>
> Une évaluation ne vaut que ce qu'on lui fournit. Au minimum, il lui faut les objectifs métier, une courte liste de scénarios d'attributs de qualité priorisés (section 2.3) et une description de l'architecture : un diagramme de contexte et un diagramme de conteneurs (section 4.3), les principaux flux d'exécution et les décisions significatives. S'ils n'existent pas, les produire est la première partie de l'évaluation, et souvent la plus précieuse.

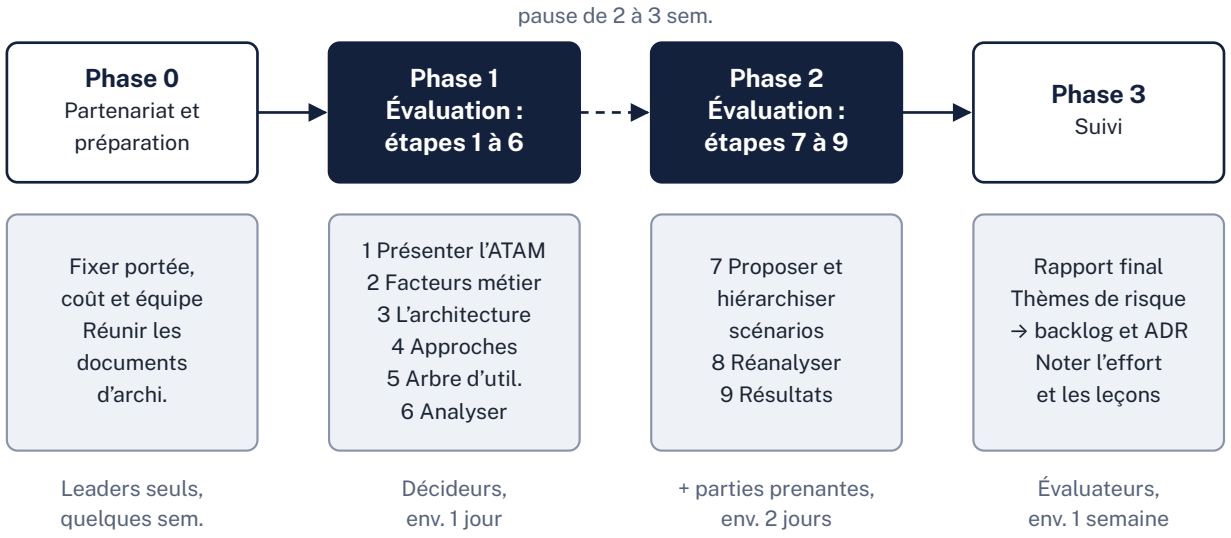
L’ATAM : origines, objectif et participants

L’Architecture Tradeoff Analysis Method (ATAM) est la méthode d’évaluation la plus connue. Rick Kazman, Mark Klein et Paul Clements, du Software Engineering Institute (SEI) de l’université Carnegie Mellon, l’ont décrite dans le rapport technique *ATAM: Method for Architecture Evaluation* (CMU/SEI-2000-TR-004, août 2000). Elle est issue de la Software Architecture Analysis Method (SAAM), plus ancienne, que Kazman et ses collègues ont publiée en 1994 et qui portait surtout sur la modifiabilité. L’ATAM est aussi décrite dans *Software Architecture in Practice* de Bass, Clements et Kazman, aujourd’hui dans sa quatrième édition (2021).

Son objectif est souvent mal compris. L’ATAM ne donne pas de note à une architecture ni de note de réussite, et elle ne prouve pas qu’un système satisfera ses exigences. Elle identifie les endroits où les décisions architecturales mettent en danger les objectifs métier et ceux où elles imposent des compromis entre qualités, afin que les personnes responsables puissent décider quoi faire. Le nom de la méthode met l’accent sur les compromis, car la plupart des risques intéressants se trouvent là où une qualité est obtenue aux dépens d’une autre.

Trois groupes y participent. L’**équipe d’évaluation**, en général trois à cinq personnes extérieures au projet, conduit la méthode : un chef d’équipe, un responsable de l’évaluation qui anime, des secrétaires qui consignent les scénarios et les débats, et des questionneurs experts de certaines qualités. Les **décideurs du projet** sont l’architecte, le chef de projet ou le responsable produit, et le commanditaire ou le client qui peut agir sur les constats. Les **parties prenantes de l’architecture** sont toutes les autres personnes concernées : développeurs, testeurs, exploitation, sécurité, intégrateurs, support et représentants des utilisateurs. L’indépendance compte : des évaluateurs qui ont conçu le système, ou qui rendent compte à l’architecte, trouvent moins de risques.

Les phases de l'ATAM



Les deux réunions d'évaluation sont séparées par une pause pendant laquelle l'équipe assimile les premiers constats et le groupe élargi des parties prenantes est constitué.

Les neuf étapes

1

1. Présenter l'ATAM

Le responsable de l'évaluation explique la méthode, les rôles des participants et les résultats, pour que personne ne prenne la séance pour un audit ou pour une revue de conception de son travail personnel.

2

2. Présenter les facteurs métier

Un représentant de haut niveau du métier explique, en termes métier, les objectifs du système, ses contraintes, ses parties prenantes et les qualités qui comptent le plus. Ces facteurs métier (business drivers)

servent d'étalon pour tout ce qui suit.

3

3. Présenter l'architecture

L'architecte décrit l'architecture à un niveau approprié : contexte, structures principales, comportement à l'exécution, déploiement, et façon dont elle répond aux facteurs. Une limite de temps habituelle d'environ une heure oblige à se concentrer.

4

4. Identifier les approches architecturales

L'architecte nomme les

approches architecturales

utilisées (styles, patrons et tactiques comme un courtier, la réplication ou la mise en cache) et l'équipe d'évaluation les répertorie, sans encore les analyser.

5 **5. Générer l'arbre d'utilité des attributs de qualité**
Les décideurs construisent ou affinent un arbre d'utilité et cotent chaque scénario feuille selon son importance métier et sa difficulté technique.

6 **6. Analyser les approches architecturales**
Pour les scénarios les plus prioritaires, les évaluateurs demandent quelles approches traitent chacun d'eux et les examinent avec des questions propres à chaque qualité, en consignant les risques, les non-risques, les points de sensibilité et les points de compromis.

7 **7. Proposer et hiérarchiser des scénarios**
En phase 2, le groupe élargi des parties prenantes propose ses propres scénarios et vote sur eux. Le résultat est comparé à l'arbre d'utilité ; les scénarios importants absents de l'arbre constituent en eux-mêmes un constat.

8 **8. Analyser de nouveau les approches architecturales**
Les nouveaux scénarios les mieux classés sont analysés de la même façon qu'à l'étape 6, cette fois avec les connaissances des parties prenantes présentes dans la salle.

9 **9. Présenter les résultats**
L'équipe d'évaluation présente les approches, les scénarios, l'arbre d'utilité, les risques, les non-risques, les points de sensibilité, les points de compromis et les thèmes de risque, et relie chaque thème aux facteurs métier qu'il menace.

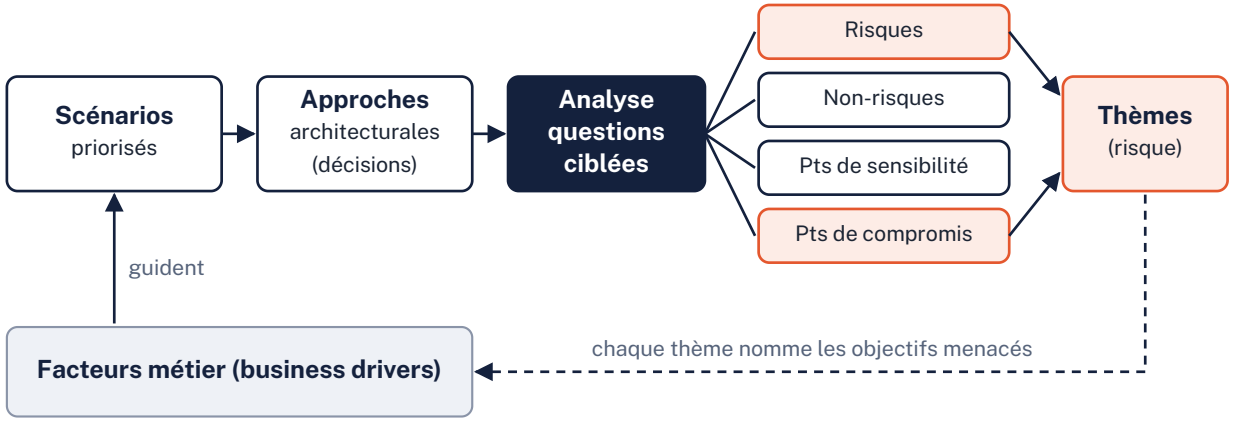
Le vocabulaire de l'analyse

Le cœur de la méthode, ce sont les étapes 6 et 8. Pour chaque scénario important, les évaluateurs demandent à l'architecte de dérouler la façon dont l'architecture y répond, et ils creusent avec des questions tirées de chaque qualité : combien d'appels synchrones se trouvent sur ce chemin, qu'arrive-t-il aux requêtes en cours quand ce nœud tombe en panne, où ce secret d'authentification est-il stocké, qu'est-ce qui limite la file d'attente ? Chaque réponse est classée à l'aide de cinq notions devenues un vocabulaire standard bien au-delà de l'ATAM elle-même.

Les résultats de l'analyse

Sortie	Signification	Exemple tiré d'une plateforme de livraison de repas
Risque	Une décision, ou une décision manquante, qui peut empêcher d'atteindre un scénario important ou un objectif métier	L'autorisation de carte se trouve dans le chemin synchrone de la commande ; un prestataire de paiement lent ralentit donc chaque commande
Non-risque	Une décision jugée solide pour le scénario, consignée avec les hypothèses dont elle dépend	L'outbox transactionnelle et les consommateurs idempotents satisfont le scénario « aucune commande perdue », à condition que le courtier réplique chaque partition trois fois
Point de sensibilité	Une propriété de composants ou de connexions qui est critique pour une réponse de qualité donnée	La latence des commandes est sensible au nombre de sauts synchrones et au temps de réponse p99 du prestataire
Point de compromis	Une propriété qui est un point de sensibilité pour plus d'une qualité, si bien qu'améliorer l'une dégrade l'autre	L'intervalle de mise à jour de la position des livreurs : plus court, il améliore la fraîcheur de la carte, mais il vide les batteries des téléphones et augmente le coût des données mobiles et de l'ingestion
Thème de risque	Un groupe de risques ayant une cause sous-jacente commune, relié aux facteurs métier qu'il menace	Des services tiers se trouvent sur des chemins critiques synchrones, ce qui menace l'objectif de doubler le volume de commandes sans perdre de conversions

Comment l'analyse produit ses résultats



L'analyse relie les scénarios priorisés aux décisions qui les traitent ; les risques qui en résultent sont regroupés en thèmes qui renvoient aux objectifs métier qu'ils menacent.

Les non-risques méritent plus d'attention qu'ils n'en reçoivent d'habitude. Consigner qu'une décision est solide *en raison* d'une hypothèse précise (trois réplicas, un temps de réponse contractuel d'un fournisseur, un trafic sous un certain niveau) transforme une vague assurance en quelque chose que l'on pourra vérifier plus tard. Quand l'hypothèse change, le non-risque devient un risque potentiel. Les points de sensibilité et de compromis sont tout aussi précieux après l'évaluation : ils indiquent aux futurs mainteneurs quels paramètres modifier avec précaution, et ce sont des emplacements naturels pour les vérifications automatisées décrites à la section 11.3.

L'arbre d'utilité revisité

La section 2.8 a présenté l'arbre d'utilité comme un outil pour hiérarchiser les exigences de qualité. Il est né dans l'ATAM, et au sein de la méthode il joue deux rôles distincts. À l'étape 5, les décideurs construisent l'arbre de haut en bas : l'utilité, puis les attributs de qualité, puis les raffinements, puis des scénarios concrets cotés (importance métier, difficulté technique). Les feuilles (H,H) deviennent l'ordre du jour de l'étape 6, car elles comptent le plus et sont les plus difficiles à atteindre : c'est donc là que les risques ont le plus de chances de se cacher.

À l'étape 7, le groupe élargi des parties prenantes propose des scénarios de bas en haut et vote sur eux, à peu près comme dans un Quality Attribute Workshop. Comparer les deux visions est l'un des moments les plus révélateurs d'une évaluation. Si les scénarios préférés des parties prenantes figurent déjà dans l'arbre, les décideurs comprennent la raison d'être de leur système. Si un scénario qui reçoit beaucoup de voix est absent, par exemple un scénario d'exploitation sur des tablettes de restaurant qui passent hors ligne, les décideurs ont un angle mort, et l'architecture en a peut-être un aussi. Ces scénarios sont ajoutés à l'arbre et analysés à l'étape 8.

EXEMPLE COMMENTÉ

Exemple commenté : une ATAM condensée de la plateforme de livraison de repas

La plateforme des chapitres 2 et 4 met en relation des clients, 4000 restaurants et 9000 livreurs. Les facteurs métier sont de doubler le volume de commandes en 18 mois (de 50 à 100 commandes par seconde au pic du vendredi) sans perdre de conversions, et de signer avec des chaînes nationales de restaurants qui exigent une disponibilité contractuelle. L'architecture : des microservices par contexte délimité (commandes, restaurants, affectation, notifications), chacun avec sa propre base de données PostgreSQL ; une API gateway ; du REST synchrone pour la prise de commande, y compris l'autorisation de carte auprès d'un prestataire de paiement externe ; un courtier Kafka avec une outbox transactionnelle pour les événements de commande ; les positions des livreurs envoyées toutes les 4 secondes et conservées dans Redis ; une seule région cloud répartie sur trois zones de disponibilité, avec des sauvegardes nocturnes copiées dans une deuxième région. L'arbre d'utilité de la phase 1 a repris les six scénarios de la section 2.8.

Latence des commandes (H,H)	p95 inférieur à 1,5 s et p99 inférieur à 3 s à 100 commandes/s. Approches : gateway, quatre sauts synchrones, autorisation de carte dans la requête. Latence mesurée du prestataire : p95 de 900 ms, p99 de 2,5 s
Analyse	Le prestataire consomme à lui seul l'essentiel du budget p99, ce qui laisse 500 ms pour quatre sauts et les écritures en base de données. La loi de Little utilise le temps moyen, et non le p99 (section 2.4), mais le risque est clair : si un ralentissement du prestataire porte la moyenne à 2,5 s, 100 commandes/s représentent 250 requêtes en cours, et le pool de 200 workers du service des commandes sature. **Risque R1** ; **point de sensibilité S1** : nombre de sauts synchrones et p99 du prestataire
Disponibilité du paiement (H,H)	Le prestataire de paiement dépasse son délai d'expiration au pic : commandes acceptées et mises en file, aucune perdue, client informé en moins de 10 s. La conception actuelle fait échouer la commande quand l'autorisation expire. **Risque R2**
Option proposée	Autoriser de façon asynchrone : accepter la commande comme en attente, l'autoriser depuis une file. Améliore la latence et la disponibilité, mais certaines commandes acceptées échouent ensuite au paiement, et les restaurants ne doivent pas préparer de commandes impayées. **Point de compromis T1** : l'endroit du parcours de commande où le paiement est confirmé
Aucune commande perdue (partie de H,H)	Outbox dans la même transaction locale que la commande, Kafka avec trois réplicas, consommateurs idempotents. **Non-risque N1**, à condition d'un facteur de réplication de 3 et d'accusés de réception du producteur par tous les réplicas synchronisés

Panne de région (H,M)	RTO 15 min, RPO 1 min. Les sauvegardes nocturnes donnent un RPO pouvant aller jusqu'à 24 heures et une restauration qui se mesure en heures. **Risque R3**
Fraîcheur du suivi (M,M)	Position sur la carte jamais vieille de plus de 5 s. 9000 livreurs toutes les 4 s, cela fait 2250 écritures/s, ce que Redis absorbe sans peine. **Point de compromis T2** : l'intervalle de mise à jour (fraîcheur contre batterie, données et coût)
Prise de contrôle de compte (H,M)	La limitation de débit par adresse IP au niveau de la gateway ne détecte pas le bourrage d'identifiants (credential stuffing) réparti sur des milliers d'adresses. **Risque R4**
Proposition de scénarios en phase 2	Les parties prenantes des restaurants et de l'exploitation ont voté massivement pour un scénario absent de l'arbre : une tablette de restaurant hors ligne pendant 10 minutes au pic doit recevoir toutes les commandes à la reconnexion. Les notifications push sont envoyées une seule fois, sans accusé de réception. **Risque R5**, et un angle mort de l'arbre
Thèmes de risque	**Thème A** (R1, R2, R4) : des dépendances tierces et en périphérie se trouvent sur des chemins critiques synchrones sans isolation ; menace l'objectif de doubler le volume sans perdre de conversions. **Thème B** (R3, R5) : les garanties de reprise et de livraison sont en retard sur la disponibilité promise aux chaînes de restaurants ; menace les contrats avec les chaînes
La suite	Des enregistrements de décision pour l'autorisation asynchrone avec un état en attente (en acceptant T1, les restaurants n'étant notifiés qu'après l'autorisation), une région de secours tiède (warm standby), la livraison des commandes aux tablettes avec accusé de réception, et la détection du bourrage d'identifiants au niveau du fournisseur d'identité. Les hypothèses de N1 sont devenues une vérification automatisée de la configuration

Les constats portent sur des décisions, pas sur des personnes ni sur la qualité du code, et chacun est rattaché à un scénario et à un facteur métier.

Deux des cinq risques venaient de scénarios que l'architecte n'avait pas priorisés ou pas envisagés : c'est la raison d'être de l'étape des parties prenantes.

Les points de compromis ne sont pas des problèmes à éliminer : T1 a été accepté délibérément et consigné, avec ses conséquences pour les restaurants et les clients.

La CBAM : mettre l'argent sur la table

L'ATAM vous dit où sont les risques ; elle ne vous dit pas quelles corrections valent la peine d'être payées. La Cost Benefit Analysis Method (CBAM), mise au point au SEI par Rick Kazman, Jai Asundi et Mark Klein (publiée pour la première fois en 2001), s'appuie sur les scénarios d'une ATAM pour construire cet argumentaire économique.

Dans les grandes lignes, les parties prenantes prennent les scénarios priorisés et attribuent une utilité (sur une échelle de 0 à 100) à plusieurs niveaux de réponse pour chacun : pire cas, situation actuelle, niveau souhaité et meilleur cas. Les architectes proposent des stratégies architecturales et estiment la réponse que chacune obtiendrait. L'utilité gagnée par chaque stratégie se lit sur la courbe utilité-réponse du scénario et est pondérée par la priorité du scénario ; additionnée sur l'ensemble des scénarios, elle constitue le bénéfice de la stratégie. Diviser le bénéfice par le coût donne un rapport valeur-coût (VFC), et les stratégies sont retenues par ordre de VFC dans la limite du budget disponible, puis confrontées à l'intuition et aux contraintes.

Une CBAM miniature pour le risque de panne de région

Stratégie	Réponse attendue	Gain d'utilité x poids	Coût annuel	Rapport valeur-coût
Actuelle : sauvegardes nocturnes	RTO d'environ 6 h, RPO jusqu'à 24 h	utilité 10 (référence)	déjà payé	n/a
A : région de secours tiède, réplication asynchrone de la base de données	RTO 10 min, RPO inférieur à 1 min	(90 - 10) x 0.3 = 24	£180k	24 / 180 = 0.13
B : pilot light (veilleuse : données répliquées, calcul démarré à la demande)	RTO 40 min, RPO inférieur à 1 min	(60 - 10) x 0.3 = 15	£70k	15 / 70 = 0.21

La stratégie B apporte plus de bénéfice par livre sterling, mais elle ne respecte pas le temps de reprise de 15 minutes qu'exigent les contrats avec les restaurants. Les chiffres éclairent la décision ; ils ne la prennent pas. La vraie valeur de la CBAM est d'obliger à faire des estimations explicites et comparables du bénéfice et du coût, et de montrer quand c'est une contrainte, et non un ratio, qui doit trancher. Sa faiblesse est que les utilités et les coûts sont des jugements ; les équipes devraient les faire varier et voir si le classement change, exactement comme la section 1.6 le recommande pour les matrices de décision pondérées. La section 11.8 montre comment transformer de telles estimations en analyse de rentabilité.

Coûts et aspects pratiques

Une ATAM complète coûte cher en temps de travail. À titre d'illustration approximative : une équipe d'évaluation de quatre personnes qui consacre chacune deux jours à la préparation (8 jours-personnes), une phase 1 d'une journée avec trois décideurs (7), une phase 2 de deux jours avec dix parties prenantes en plus (34) et une semaine de suivi par deux personnes (10)

totalisent environ 60 jours-personnes, avant que quiconque ait corrigé quoi que ce soit. Pour un système dont la défaillance coûterait des millions, ou qui fonctionnera pendant dix ans, c'est une assurance bon marché. Pour une petite équipe produit, c'est disproportionné.

Plusieurs pratiques font que les évaluations en valent la peine. Adaptez la méthode : Bass, Clements et Kazman décrivent une évaluation d'architecture allégée, une ATAM réduite pour les équipes internes qui peut se faire en une journée ou moins, et certaines organisations ne mènent que les étapes 2 à 6 avec une poignée de personnes. Préparez-vous bien : l'essentiel du coût d'une mauvaise évaluation est dépensé à découvrir, en séance, que personne ne sait décrire l'architecture. Gardez des évaluateurs indépendants mais informés. Convenez à l'avance de qui est responsable des constats et de la façon dont ils seront suivis, idéalement sous forme d'enregistrements de décision et d'éléments de backlog dotés de responsables. Et recommencez : les constats sont un instantané, et les risques changent à mesure que le système et le métier évoluent.

Les praticiens ne s'accordent pas sur la place de la méthode aujourd'hui. Ses détracteurs y voient un vestige lourd des grands projets conçus en amont, mal adapté aux équipes qui déploient de nombreuses fois par jour. Ses défenseurs répondent que ses notions (scénarios, points de sensibilité et de compromis, thèmes de risque) sont exactement ce qui manque aux équipes agiles quand elles sautent l'évaluation, et que la méthode peut être allégée. Les deux camps ont en partie raison : le cérémonial complet convient aux systèmes à forts enjeux et à longue durée de vie et aux engagements majeurs, tandis que le vocabulaire et l'habitude d'un questionnement guidé par les scénarios ont leur place partout. La section 11.2 traite de l'extrémité la plus légère.

ATTENTION

Erreurs courantes dans les évaluations d'architecture

- Traiter l'évaluation comme un examen avec réussite ou échec, si bien que l'architecte se défend au lieu d'explorer, et que les risques sont cachés.
- Évaluer sans facteurs métier, ce qui produit une liste d'inquiétudes techniques impossibles à classer.
- Laisser l'analyse dériver vers le style de code, les préférences d'outils ou les goûts personnels au lieu de questions guidées par les scénarios.
- Consigner des non-risques sans leurs hypothèses, si bien que personne ne remarque quand une hypothèse cesse d'être vraie.
- Produire une longue liste de risques sans thèmes de risque, ce qui empêche les managers de voir les quelques causes sous-jacentes.
- Tenir l'évaluation après la signature du contrat ou l'achat de la plateforme, quand les constats ne peuvent plus qu'être regrettés.
- Ne pas assurer le suivi : les constats qui ne deviennent pas des enregistrements de décision, des éléments de backlog et des responsables sont oubliés en un trimestre.

Questions fréquentes

L'ATAM peut-elle s'appliquer à un système qui existe déjà ?
Oui. Les étapes sont les mêmes ; l'architecture est reconstituée à partir de la documentation, du code et des personnes qui exploitent le système, et les mesures du système en production remplacent les estimations. Évaluer un système existant est courant avant une extension majeure, une migration ou une acquisition.

Une ATAM sans problème signifie-t-elle que le système satisfera ses exigences ?
Non. L'ATAM examine un échantillon de scénarios au niveau des décisions de conception. Elle réduit le risque de mauvaises surprises mais ne peut pas remplacer la mesure, les tests et l'expérience d'exploitation. Ses constats devraient être transformés en tests et en fonctions d'aptitude chaque fois que c'est possible.

Qui doit présenter l'architecture ?
L'architecte ou l'ingénieur principal qui la connaît le mieux. Le rôle de l'équipe d'évaluation est de poser des questions, pas de présenter, et celui de l'architecte est d'expliquer et d'explorer, pas de se défendre.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Quelle est la différence entre un point de sensibilité et un point de compromis ?**

Un point de sensibilité est une propriété critique pour une réponse de qualité donnée. Un point de compromis est une propriété qui est un point de sensibilité pour plusieurs qualités à la fois, si bien que la modifier aide l'une et nuit à l'autre.

▼ **Pourquoi l'ATAM consigne-t-elle les non-risques, et que faut-il consigner avec eux ?**

Pour montrer quelles décisions ont été examinées et jugées solides. Les hypothèses dont elles dépendent doivent aussi être consignées, afin que le non-risque puisse être réexaminé si une hypothèse change.

▼ **Que signifie le fait que les parties prenantes votent massivement, en phase 2, pour un scénario absent de l'arbre d'utilité ?**

Les décideurs ont un angle mort sur ce qui compte, et l'architecture ne traite peut-être pas ce scénario. Il est ajouté à l'arbre et analysé à l'étape 8.

▼ **Dans la CBAM, comment calcule-t-on le rapport valeur-coût d'une stratégie ?**

Son bénéfice, l'utilité pondérée par la priorité qu'elle ajoute sur les scénarios qu'elle touche, est divisé par son coût.

▼ **À vous : Classez ces résultats de l'ATAM : le débit des commandes dépend directement du nombre de partitions du courtier ; le chiffrement d'une liaison améliore la sécurité mais ajoute de la latence ; un seul prestataire de paiement, sans solution de repli, sur le chemin de commande ; la défaillance d'une zone est gérée par un failover testé.**

Le nombre de partitions est un point de sensibilité : une propriété influence fortement une qualité. Le chiffrement est un point de compromis, car la même décision aide la sécurité et nuit aux performances. Le prestataire de paiement unique est un risque, qui menace le facteur métier de disponibilité des commandes. Le basculement testé est un non-risque, sous des hypothèses énoncées, comme une capacité suffisante dans les autres zones. Regroupez les risques par thèmes, puis faites de chaque constat un enregistrement de décision ou un élément de backlog doté d'un responsable.

SOURCES

- SEI: ATAM: Method for Architecture Evaluation (CMU/SEI-2000-TR-004): https://www.sei.cmu.edu/library/atam-method-for-architecture-evaluation/
- SEI: Making Architecture Design Decisions: An Economic Approach (the CBAM): https://www.sei.cmu.edu/library/making-architecture-design-decisions-an-economic-approach

SECTION 11.2 SUR 11.8 · AVANCÉ · 27 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Revues légères et risk-storming

AVANT DE LIRE

Cette section s'appuie sur :

- 4.3 Le modèle C4
- 4.8 Communiquer
- 11.1 Évaluation avec l'ATAM

Termes déjà vus qu'elle utilise : Risque (dans l'évaluation d'architecture), Modèle C4, Diagramme de contexte système, Diagramme de conteneurs, Enregistrement de décision d'architecture (ADR)

L'IDÉE EN BREF

Avant une longue randonnée, un groupe avisé étale la carte sur la table et chacun, en silence et de son côté, marque les endroits qui l'inquiètent : le passage de la rivière, l'éboulis raide, le tronçon sans réseau téléphonique. Puis ils comparent. Les endroits marqués par beaucoup de personnes reçoivent un plan ; un endroit marqué par une seule personne reçoit une question, car cette personne sait peut-être quelque chose que les autres ignorent. Les revues d'architecture légères fonctionnent ainsi. Au lieu d'un comité lointain qui approuve un plan achevé, les personnes qui connaissent le système examinent ensemble ses diagrammes, tôt et souvent, et s'accordent sur les risques à traiter en premier.

CE QU'IL FAUT SAVOIR

- Les comités de revue qui se réunissent à des étapes fixes ont tendance à voir les conceptions trop tard, manquent de contexte, suscitent des présentations défensives et ajoutent un délai d'attente ; une revue indépendante fonctionne mieux lorsqu'elle est fréquente, précoce, collaborative et centrée sur les risques.
- Le risk-storming (Simon Brown) comporte quatre étapes : dessiner des diagrammes, identifier les risques individuellement et en silence, les rassembler sur les diagrammes, puis les examiner et les résumer, en notant la probabilité multipliée par l'impact de 1 à 9.
- Un pre-mortem (Gary Klein, Harvard Business Review, 2007) demande aux participants d'imaginer que le projet a déjà échoué et d'écrire pourquoi, ce qui fait émerger des doutes sur les plans, les compétences et les calendriers que les revues de diagrammes manquent.

- Les katas d'architecture (Ted Neward) développent les compétences de conception et de revue, tandis que TARA (Eoin Woods) examine les systèmes existants en combinant le jugement d'experts et l'analyse automatisée du code et des données d'exécution.
- Les revues écrites de conception et de RFC fonctionnent mieux avec une décision énoncée, une date limite pour les commentaires, des relecteurs obligatoires et des commentaires étiquetés comme questions, suggestions ou objections bloquantes.
- Une bonne revue produit un registre des risques avec des responsables, des dates, les risques acceptés et les non-risques avec leurs hypothèses, publié en quelques jours et suivi dans le backlog habituel.

Pourquoi les comités de revue lourds échouent souvent

De nombreuses organisations gouvernent l'architecture au moyen d'un comité de revue d'architecture : un groupe d'architectes seniors qui se réunit chaque mois, reçoit une présentation ou un long document, et approuve ou rejette des conceptions. L'intention est bonne : un regard indépendant et expérimenté sur les décisions importantes. Les résultats sont souvent médiocres, pour des raisons structurelles plutôt que personnelles.

Le calendrier est le premier problème. Un comité qui se réunit à des étapes fixes voit une conception soit trop tôt (rien n'est encore décidé), soit, bien plus souvent, trop tard, quand l'équipe a déjà construit un prototype, s'est engagée auprès d'un fournisseur et a promis une date. Un rejet coûte alors si cher que le comité approuve avec des conditions que personne ne suit. Le contexte est le deuxième : des relecteurs qui passent une heure sur un système qu'ils n'ont jamais vu posent des questions génériques et manquent les risques précis que les développeurs, les équipes d'exploitation et les ingénieurs sécurité de l'équipe pourraient citer en quelques minutes. Les incitations sont le troisième : quand une revue est un examen, les équipes présentent la conception qu'elles veulent faire approuver, pas les doutes qu'elles ont, et le comité apprend peu. L'attente est le quatrième : une réunion mensuelle à l'ordre du jour chargé ajoute des semaines d'attente à chaque décision importante, ce qui pousse les équipes à éviter le comité en qualifiant leurs décisions de mineures.

Rien de tout cela ne signifie qu'une revue indépendante est inutile. Les secteurs réglementés (banque, santé, aviation) exigent souvent une validation documentée, et certaines décisions ont réellement besoin de l'autorité d'un groupe de haut niveau. La leçon est que la revue doit être *fréquente, précoce, collaborative et centrée sur les risques*, l'approbation formelle étant réservée aux rares décisions qui l'exigent vraiment. La section 11.4 revient sur la façon dont la gouvernance peut fonctionner sans points de validation lents.

Ce qu’apporte une bonne revue légère

Une revue légère est une séance courte et structurée, en général d’une heure ou deux, pendant laquelle les personnes qui connaissent le mieux une conception et quelques personnes qui la découvrent cherchent ensemble les risques. Elle vise quatre choses : découvrir les risques tant qu’ils coûtent peu à traiter, construire une compréhension partagée de la conception dans l’équipe, offrir une remise en question indépendante des hypothèses de l’architecte, et laisser une trace des risques acceptés, atténués ou écartés. Son résultat est une liste de risques priorisée avec des responsables, pas un verdict.

Plusieurs techniques correspondent à cette description. Elles diffèrent par ce qu’elles examinent (un diagramme, un plan, un document ou un système en fonctionnement) et par ce qu’elles font le mieux, et elles se combinent bien.

Les techniques de revue légère comparées

Technique	Examine	Idéal pour	Durée habituelle
Risk-storming (Simon Brown)	Diagrammes d’architecture	Trouver rapidement les risques techniques avec toute l’équipe	45 à 90 minutes
Pre-mortem (Gary Klein)	Un plan ou une décision	Faire émerger les doutes sur l’ensemble de l’entreprise, y compris les personnes et le calendrier	30 à 60 minutes
Kata d’architecture (Ted Neward)	Un énoncé court, inventé ou réel	Développer les compétences de conception et explorer des alternatives	1 à 3 heures
TARA (Eoin Woods)	Un système existant : personnes, documents, code et données d’exécution	Évaluer la dérive et les risques d’un système en fonctionnement	Quelques jours de travail d’experts
Revue de conception ou de RFC	Une proposition écrite	Des commentaires asynchrones et ouverts à tous avant la construction	Une période de commentaires de quelques jours, plus une courte réunion
Revue par liste de contrôle	Toute conception	S’assurer que les angles morts habituels ont été examinés	Intégrée à l’une des techniques ci-dessus

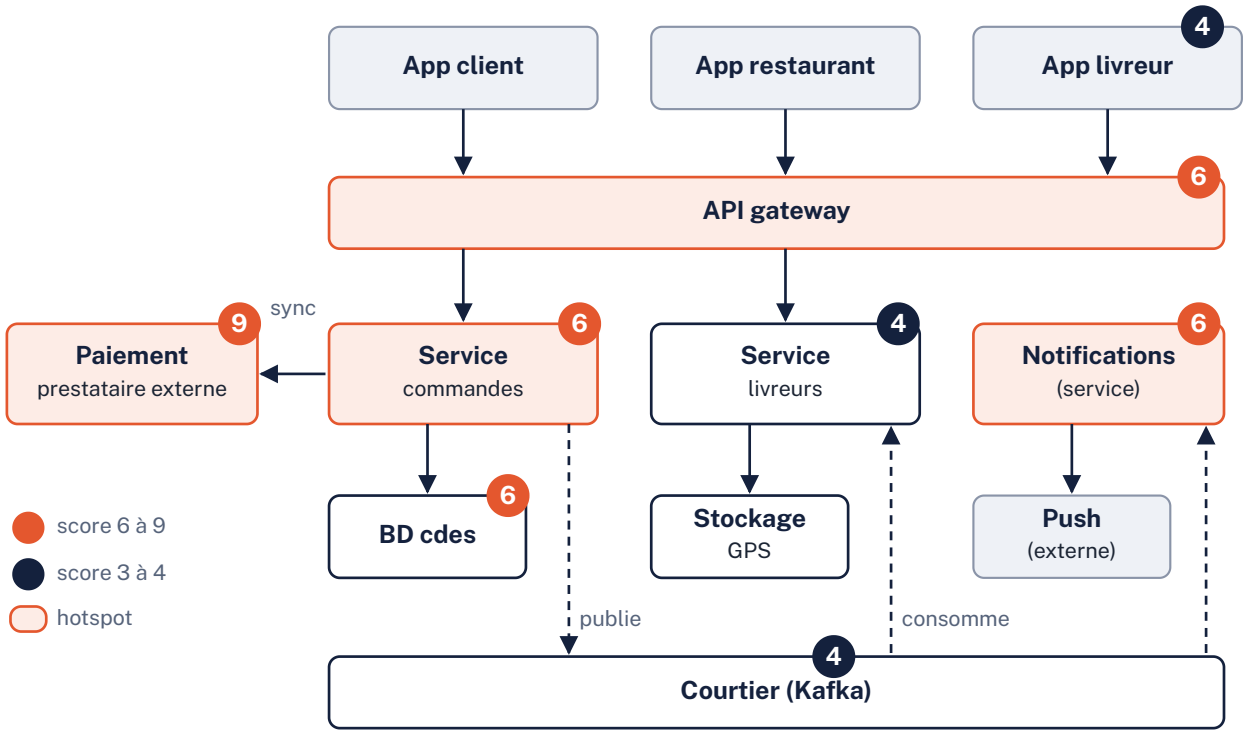
Risk-storming

Le risk-storming est une technique visuelle et collaborative créée par Simon Brown, l'auteur du modèle C4 (section 4.3). Il repose sur une observation simple : des personnes différentes voient des risques différents, et le meilleur moyen de les recueillir est de placer tout le monde devant les mêmes diagrammes et de demander à chacun de les examiner seul avant d'en discuter. Il comporte quatre étapes.

1. Dessiner l'architecture. Affichez des diagrammes de ce que vous prévoyez de construire ou de modifier, idéalement à plus d'un niveau : un diagramme de contexte système et un diagramme de conteneurs suffisent en général, avec un diagramme de composants pour le conteneur le plus risqué. **2. Identifier les risques individuellement.** Chaque participant écrit en silence un risque par note adhésive, avec une courte description et une priorité, pendant une dizaine de minutes. Le silence compte : il empêche la première personne qui parle, ou la plus haut placée, d'ancrer tous les autres, ce qui est la principale façon dont les discussions de groupe perdent de l'information. **3. Rassembler les risques sur les diagrammes.** Chacun place ses notes sur la partie du diagramme que le risque concerne. Des grappes apparaissent immédiatement, et le diagramme devient une carte de chaleur. **4. Examiner et résumer.** Le groupe discute des grappes, puis accorde une attention particulière à deux types de notes : les risques soulevés par une seule personne (souvent quelqu'un qui possède un savoir que les autres n'ont pas, comme l'ingénieur du support) et les risques auxquels différentes personnes ont donné des priorités très différentes (signe d'un malentendu). Les risques retenus entrent dans un registre des risques avec des responsables ; les éléments dont on a convenu qu'ils ne sont pas des risques sont consignés aussi.

La priorité se note en général comme la probabilité multipliée par l'impact, chacun étant coté faible (1), moyen (2) ou élevé (3), ce qui donne des scores de 1 à 9. Les recommandations de Brown les répartissent en élevé (6 ou 9), moyen (3 ou 4) et faible (1 ou 2), représentés par convention avec des notes roses ou rouges, jaunes ou orangées, et vertes. Une liste de rappel aide chacun à penser au-delà de sa propre spécialité : scalabilité, disponibilité, performances, sécurité, cohérence des données, dépendances envers des tiers, coût, maturité des technologies, compétences de l'équipe, et livraison et exploitation.

Une carte de chaleur de risk-storming sur un diagramme de conteneurs



Après le rassemblement, les scores se concentrent sur le chemin des commandes et du paiement, la gateway partagée et les notifications aux restaurants ; ces conteneurs sont les points chauds à discuter en premier.

EXEMPLE COMMENTÉ

Exemple commenté : un risk-storming du diagramme de conteneurs de la plateforme de livraison de repas

Six personnes (l'architecte, deux développeurs, un ingénieur d'exploitation, un ingénieur sécurité et un ingénieur du support qui traite les réclamations des restaurants) ont passé 75 minutes sur le diagramme de conteneurs ci-dessus, avec le diagramme de contexte système à côté. Chacun a écrit des risques en silence pendant dix minutes ; 31 notes ont été affichées, qui se sont regroupées en 8 risques distincts.

Prestataire de paiement sur le chemin synchrone de la commande (9)	Soulevé par quatre personnes. Les ralentissements du prestataire au pic bloquent la prise de commande. Action : enregistrement de décision sur l'autorisation asynchrone avec un état de commande en attente ; responsable : l'architecte ; échéance dans deux semaines
Notifications aux restaurants sans accusé de réception (6)	Soulevé uniquement par l'ingénieur du support, qui voit chaque vendredi des réclamations pour commandes manquées. Personne d'autre n'était au courant. Action : livraison avec accusé de réception, renvoi et alerte quand une tablette cesse d'accuser réception ; responsable : l'équipe des notifications
Configuration partagée de la gateway (6)	Un seul changement de route défectueux peut faire tomber les trois apps à la fois. Action : configuration de la gateway par app et déploiement canary des changements de gateway
Basculement de la base de données des commandes jamais testé (6)	Le réplica existe, mais sa promotion n'a jamais été répétée. Action : exercice de basculement trimestriel, mesuré par rapport au scénario de reprise
Compétences Kafka (notées 9 par une personne, 2 par les autres)	Le seul ingénieur qui avait déjà exploité Kafka a expliqué le rééquilibrage des partitions pendant les déploiements, le retard des consommateurs et les erreurs de rétention. Après discussion, le groupe s'est accordé sur 4 : réel mais gérable. Action : service Kafka managé et un runbook
Trafic de position des livreurs passant par la gateway (4)	Environ 2250 mises à jour par seconde partagent la gateway avec le trafic des commandes. Action : mesurer lors du prochain test de charge ; endpoint d'ingestion séparé si la latence des commandes en pâtit
Retard du consommateur de l'affectation au pic (4)	Action : alerte sur le retard des consommateurs ; ajouter une fonction d'aptitude au test de charge
Le stockage des positions perd ses données au redémarrage (2)	Convenu que ce n'est pas un risque : les positions se rafraîchissent toutes les 4 secondes. Consigné comme tel, avec l'hypothèse

Le constat le plus précieux (les notifications aux restaurants) est venu de la personne la moins susceptible de prendre la parole en premier dans une discussion ouverte : c'est tout l'intérêt de l'identification individuelle et silencieuse.

Le désaccord sur Kafka a été résolu par la connaissance, pas en faisant la moyenne des scores.

Chaque risque retenu a quitté la salle avec un responsable et une action ; le registre a été publié le jour même.

Les pre-mortems

Le risk-storming examine une structure. Un pre-mortem examine un plan. Gary Klein l'a décrit dans la *Harvard Business Review* en septembre 2007. Une autopsie (post-mortem) explique pourquoi un patient est mort, ce qui aide tout le monde sauf le patient ; un pre-mortem a lieu avant le début du projet, quand les réponses peuvent encore en changer le cours.

La procédure est courte. L'animateur présente au groupe le plan ou la décision, puis annonce que l'on se trouve quelques mois plus tard et que le projet a échoué, gravement. Chaque personne passe quelques minutes à écrire, seule, toutes les raisons de l'échec qui lui viennent à l'esprit. L'animateur fait ensuite le tour de la salle en recueillant une raison à la fois jusqu'à épuisement des listes, et le responsable du plan les examine pour trouver des moyens de le renforcer. Klein cite des travaux de Deborah Mitchell, Jay Russo et Nancy Pennington (1989) montrant qu'imaginer qu'un événement s'est déjà produit (le recul prospectif) augmente d'environ 30 % la capacité des personnes à trouver les raisons de résultats futurs. Ce cadrage change aussi la dynamique sociale : la question n'est plus de savoir si le plan pourrait échouer, ce qui paraît déloyal, mais pourquoi il a échoué, ce qui invite à la créativité.

En architecture, les pre-mortems sont particulièrement utiles avant les gros engagements : adopter un style piloté par les événements, migrer vers un nouveau cloud, remplacer un système hérité (section 11.6) ou introduire une plateforme. Parmi les résultats typiques qu'une revue fondée sur les diagrammes manquerait : l'équipe qui manque de compétences dans la nouvelle technologie, la migration qui s'enlise à mi-chemin avec deux systèmes à exploiter, un fournisseur clé qui change ses tarifs, et le métier qui perd patience avant que les bénéfices apparaissent.

SI VOUS DÉBUTEZ

Pas à pas : imaginer d'abord l'échec

L'idée qui demande le plus d'attention ici est la raison pour laquelle un pre-mortem fonctionne : imaginer qu'un plan a déjà échoué fait sortir des doutes qu'une simple discussion du type « qu'est-ce qui pourrait mal tourner ? » laisse souvent non exprimés. Suivez un plan issu de la séance de risk-storming ci-dessus.

- Supposons que l'équipe décide de sortir l'autorisation de carte du chemin synchrone de la commande, en acceptant les commandes dans un état en attente. Avant le début des travaux, l'équipe et des personnes des équipes voisines se réunissent une demi-heure.
- L'animateur explique le plan, puis annonce que l'on est six mois plus tard et que le changement a gravement échoué. Chaque personne écrit, seule et en silence, toutes les raisons qui lui viennent à l'esprit.
- C'est le cadrage qui fait le travail. Si l'on demande si le plan pourrait échouer, quelqu'un qui a des doutes paraît déloyal en le disant. Si l'on demande pourquoi il a échoué, chaque doute devient une réponse utile.
- L'animateur recueille une raison à la fois jusqu'à épuisement des listes : personne dans l'équipe des commandes n'avait jamais construit ce type de conception ; le changement s'est enlisé à mi-chemin, laissant fonctionner à la fois l'ancien et le nouveau parcours de paiement ; le prestataire de paiement a changé ses tarifs ; le métier a perdu patience avant que les bénéfices apparaissent.
- Peu de ces raisons apparaîtraient sous forme de notes adhésives sur le diagramme de conteneurs : elles concernent le calendrier, les fournisseurs et la patience plus que les boîtes et les flèches. Le responsable du plan le renforce ensuite, par exemple en organisant d'abord une formation et en fixant une date pour éteindre l'ancien parcours.
- **La leçon :** le risk-storming examine une structure ; un pre-mortem examine un plan. Imaginer que l'échec s'est déjà produit rend les raisons plus faciles à trouver et moins risquées à dire ; organisez-en donc un avant un gros engagement, tant que les réponses peuvent encore changer le plan.

Les katas d'architecture

Les katas d'architecture ont été créés par Ted Neward, en s'inspirant des katas de code de Dave Thomas, comme exercices d'entraînement pour les architectes. Un kata est un court énoncé décrivant un système fictif (quelques paragraphes d'exigences, d'utilisateurs et de contraintes, volontairement incomplets). De petites équipes de trois à cinq personnes y travaillent pendant un temps fixé, en interrogeant un modérateur qui joue le client pour découvrir les exigences

manquantes, puis esquissent une architecture et la présentent aux autres équipes, qui la contestent durement mais loyalement avant que la salle ne vote. Neal Ford tient à jour une collection publique de katas, et O'Reilly a organisé des compétitions de katas d'architecture.

Les katas ne sont pas des revues d'une conception réelle, mais ils contribuent à la culture de la revue de deux façons. Ils développent la capacité à expliquer et à défendre une conception face aux questions, et à poser de bonnes questions sur celle de quelqu'un d'autre, ce dont les revues légères ont justement besoin. Et un kata interne sur un vrai problème à venir, avec deux ou trois équipes qui conçoivent chacune de leur côté, est un moyen peu coûteux d'explorer l'espace des options avant de s'engager : quand trois équipes produisent trois conceptions différentes, les différences montrent où se trouvent les vrais compromis.

TARA et revues fondées sur des listes de contrôle

Les revues de systèmes qui existent déjà font face à un autre problème : l'architecture telle que construite peut différer de l'architecture telle que décrite, et les éléments de preuve sont dispersés entre les personnes, les documents, le code et les données de production. Eoin Woods, coauteur de *Software Systems Architecture*, a décrit en 2011 la Tiny Architectural Review Approach (TARA) comme une alternative industrielle, volontairement légère, aux méthodes très centrées sur les scénarios. Un petit nombre de relecteurs recueillent les exigences clés et le contexte du système, établissent sa structure réelle, évaluent sa mise en œuvre à l'aide d'une analyse automatisée (dépendances entre modules, taille, métriques de code, couverture de tests) ainsi que d'informations d'exécution comme les journaux, s'appuient sur leur jugement d'experts pour relier ce qu'ils trouvent aux exigences, et présentent leurs constats et recommandations sous une forme prédéfinie. Sa force est la rapidité et les preuves ; sa faiblesse, que Woods a reconnue, est sa dépendance à l'expertise des relecteurs.

Les listes de contrôle complètent toutes les techniques. Une liste de contrôle ne trouve pas de risques par elle-même, mais elle évite qu'une revue oublie toute une catégorie (restauration des sauvegardes, secrets, coût à grande échelle) simplement parce que personne dans la salle n'y a pensé. Gardez des listes de contrôle courtes, propres à l'historique d'incidents de votre organisation, et formulées sous forme de questions qui exigent des éléments de preuve plutôt qu'un oui ou un non.

Les revues Well-Architected

Les grands fournisseurs de cloud publient des cadres qui peuvent servir de listes de contrôle toutes faites pour les revues. AWS et Google Cloud décrivent chacun six piliers et Azure cinq, couvrant l'exploitation, la sécurité, la fiabilité, les performances et le coût, les deux premiers y ajoutant la durabilité. Une revue fait passer une charge de travail par une série de questions structurées. L'AWS Well-Architected Tool n'entraîne aucun coût supplémentaire, enregistre vos

réponses et signale les problèmes à haut risque, et des lentilles personnalisées (custom lenses) peuvent y ajouter vos propres règles ; Azure propose une Well-Architected Review en ligne. Pour la charge de travail des commandes de la plateforme de livraison de repas, une séance d'une demi-journée offre une large couverture et une liste de risques à corriger ou à accepter. Les limites sont réelles : les questions expriment la vision des bonnes pratiques d'un seul fournisseur, et elles ne mettent pas en balance vos propres scénarios de qualité comme le fait la section 11.1. Vérifiez la version actuelle, car les piliers et les questions évoluent.

Documents de conception et revues de RFC

La section 4.8 a présenté les documents de conception et les RFC comme un moyen de proposer un changement par écrit avant de le construire. Bien les relire est une compétence à part entière. Une revue écrite présente de réels avantages sur une réunion : les relecteurs commentent quand ils ont le temps de réfléchir, les collègues à distance ou plus discrets contribuent à égalité, et les commentaires constituent une trace. Ses échecs typiques sont des documents que personne ne lit, des fils de commentaires qui ne se referment jamais et des revues qui deviennent un point de validation officieux.

La bonne pratique consiste à indiquer en tête la décision recherchée et la date limite des commentaires, à nommer les relecteurs dont l'avis est requis (sécurité, exploitation, responsables des services concernés) et à inviter toute autre personne ; à demander aux relecteurs d'étiqueter chaque commentaire comme question, suggestion ou objection bloquante, pour que l'auteur sache lesquels doivent être résolus ; à relire au regard des scénarios de qualité et de la liste de contrôle ci-dessous, et non des goûts personnels ; à conclure par une courte réunion uniquement pour les objections non résolues ; et à consigner les décisions importantes sous forme d'ADR. Dans les organisations qui utilisent le processus de consultation (section 11.4), la RFC est le moyen de solliciter des conseils, et la décision reste à son auteur.

Animer une revue d'architecture de 90 minutes, pas à pas

1. **Avant : préparer (quelques jours à l'avance)**
   L'architecte fait circuler un résumé d'une page, les diagrammes de contexte et de conteneurs, les trois à cinq principaux scénarios de qualité chiffrés et les questions ouvertes. Invitez cinq à huit personnes : l'équipe, l'exploitation, la sécurité, une personne de chaque équipe voisine et une personne extérieure expérimentée. Choisissez un animateur qui ne soit pas l'architecte.

2. **0 à 10 minutes : objectifs et règles**
   L'animateur énonce le but (trouver des risques, non approuver ou rejeter), la durée impartie et le résultat attendu. Un représentant du métier ou le responsable produit rappelle les facteurs métier en deux minutes.

3. **10 à 25 minutes : dérouler l'architecture à travers des scénarios**
   L'architecte déroule deux ou trois des scénarios les plus importants à travers les diagrammes, par exemple une commande au pic et une panne du prestataire de paiement. Questions de clarification uniquement.

4. **25 à 35 minutes : identification silencieuse des risques**
   Chacun écrit des risques sur des notes, un par note, avec une probabilité et un impact. Pas de discussion.

5. **35 à 50 minutes : rassembler et regrouper**
   Les notes vont sur les diagrammes. L'animateur fusionne les doublons à voix haute et met en évidence les grappes, les notes isolées et les désaccords de priorité.

6. **50 à 75 minutes : discuter des principaux risques**
   Prenez d'abord les grappes aux scores les plus élevés. Pour chacune, convenez de ce qu'est vraiment le risque, de son score et de sa nature : un risque, un non-risque avec une hypothèse, ou une question qui demande un spike.

7. **75 à 85 minutes : actions et responsables**
   Chaque risque élevé ou moyen retenu reçoit une atténuation, un responsable et une date. Les décisions à consigner sous forme d'ADR sont listées.

8. **85 à 90 minutes : clôture**
   Relisez les risques, les risques acceptés et les non-risques avec leurs hypothèses. Convenez du moment où la conception sera réexaminée.

9. **Après : publier sous deux jours**
   Publiez le registre des risques, des photos des diagrammes annotés et la liste des actions là où l'équipe conserve sa documentation d'architecture, et suivez les actions dans le backlog habituel.

GUIDE DE DÉCISION

Guide de décision : quel type de revue ?

Commencez en haut ; les techniques peuvent se combiner.

Q1 **Un engagement coûteux et difficile à défaire est-il imminent, comme un contrat pluriannuel, l'achat d'une plateforme ou une étape de financement, avec de nombreuses parties prenantes ?**

- **Oui** Utilisez une ATAM ou une ATAM réduite avec des évaluateurs indépendants (section 11.1), et ajoutez un pre-mortem sur le plan.
- **Non** Continuez.

Q2 **Existe-t-il une proposition écrite pour un changement important ?**

- **Oui** Menez une revue écrite de conception ou de RFC avec une date limite pour les commentaires, puis une courte réunion pour les objections bloquantes.
- **Non** Continuez.

Q3 **Existe-t-il des diagrammes de l'architecture proposée ou existante ?**

- **Oui** Faites un risk-storming sur ces diagrammes, en suivant le format de 90 minutes.
- **Non** Dessinez d'abord un diagramme de contexte et un diagramme de conteneurs ; les dessiner suffit souvent à révéler des risques.

Q4 **La principale inquiétude porte-t-elle sur la réussite de l'ensemble du projet (compétences, calendrier, migration, adoption) plutôt que sur la structure ?**

- **Oui** Ajoutez un pre-mortem.
- **Non** Continuez.

Q5 **Le système fonctionne-t-il déjà, avec une dérive soupçonnée ou une dette croissante ?**

- **Oui** Utilisez une revue de type TARA qui combine des entretiens et une analyse automatisée du code et de l'exécution, et transformez les constats en fonctions d'aptitude.
- **Non** Appuyez-vous sur les fonctions d'aptitude entre les revues, et refaites une revue à la prochaine décision importante.

Liste de contrôle : questions pour toute revue d'architecture légère

Demandez des preuves, pas des assurances. Adaptez la liste à l'historique d'incidents de votre organisation.

- [ ] **Quels sont les trois à cinq scénarios de qualité qui comptent le plus, et quels sont leurs chiffres ?**
  Sans facteurs mesurables, la revue ne peut pas classer les risques (section 2.3).
- [ ] **Qu'y a-t-il sur le chemin synchrone du parcours le plus important ?**
  Comptez les sauts, listez les délais d'expiration et les nouvelles tentatives, et identifiez chaque tiers présent sur le chemin.
- [ ] **Que se passe-t-il quand chaque dépendance est lente ou en panne ?**
  Lent est généralement pire qu'en panne ; posez la question pour les deux cas, ainsi que sur la façon dont les utilisateurs sont informés.
- [ ] **Qui est propriétaire de chaque donnée importante, et une restauration a-t-elle été testée ?**
  Écrivain unique, attentes de cohérence, conservation, et preuve d'une restauration récente réussie (chapitre 7).
- [ ] **Où sont les frontières de confiance, les identités et les secrets ?**
  Cherchez un modèle de menaces à jour et des secrets dans le code ou la configuration (chapitre 8).
- [ ] **Qu'est-ce qui casse en premier à deux fois le pic attendu ?**
  Demandez le modèle de capacité ou les résultats des tests de charge, pas l'intuition.
- [ ] **Comment le système est-il observé, déployé et ramené en arrière ?**
  Tableaux de bord, alertes liées aux SLO, stratégie de déploiement et durée d'un retour arrière.
- [ ] **Qu'est-ce qui changera probablement dans l'année à venir, et qu'est-ce que ce changement toucherait ?**
  Vérifiez que les changements probables restent à l'intérieur d'un seul module ou service.
- [ ] **Combien coûtera l'exploitation au volume attendu ?**
  Un coût par commande, par utilisateur ou par transaction, avec ses principaux facteurs.
- [ ] **Les décisions importantes sont-elles consignées avec leurs alternatives et leurs conséquences ?**
  Des ADR ou un document de conception, y compris les compromis acceptés.

- [ ] **La structure des équipes correspond-elle à la conception ?**
  Chaque partie a une équipe propriétaire qui a les compétences pour l'exploiter (section 3.7 et section 11.5).

ATTENTION

Comment les revues légères tournent mal

- Une discussion ouverte avant l'identification silencieuse, si bien que la première personne qui parle ancre le groupe.
- Faire la moyenne des désaccords au lieu de demander pourquoi une personne a noté un risque si différemment.
- Examiner la notation du diagramme au lieu des risques de l'architecture.
- Ni responsables ni dates, si bien que le registre des risques devient une liste de bonnes intentions.
- L'architecte comme animateur, ce qui rend difficile de recevoir des critiques tout en menant la séance.
- Ne mener des revues que lorsque quelque chose va déjà mal ; des revues régulières et routinières coûtent moins cher et suscitent moins de réflexes défensifs.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi le risk-storming demande-t-il aux participants d'identifier les risques en silence avant toute discussion ?

Pour empêcher la première personne qui parle, ou la plus haut placée, d'ancrer le groupe, afin que les connaissances propres à chacun arrivent jusqu'au diagramme.

▼ Dans le risk-storming, quels sont les deux types de notes qui méritent une attention particulière lors de l'étape d'examen ?

Les risques soulevés par une seule personne, qui sait peut-être quelque chose que les autres ignorent, et les risques auxquels on a donné des priorités très différentes, signe d'un malentendu.

▼ Qu'examine un pre-mortem qu'une séance de risk-storming sur un diagramme de conteneurs n'examine généralement pas ?

Le plan et son contexte dans leur ensemble : compétences, calendrier, migration, fournisseurs et patience de l'organisation, plutôt que la structure technique.

▼ **Citez deux raisons structurelles pour lesquelles les comités mensuels de revue d'architecture échouent souvent.**

Deux parmi les suivantes : ils voient les conceptions trop tard, les relecteurs manquent de contexte, les équipes présentent sur la défensive, et l'attente ajoute des semaines de délai qui incitent à les contourner.

▼ **À vous : Un risk-storming évalue (probabilité, impact) : panne du prestataire de paiement (3, 3) ; mauvaise configuration du courtier (2, 3) ; index de recherche périmé (3, 1) ; incendie de centre de données (1, 3). Attribuez un score et classez-les. Pourquoi les notes ont-elles d'abord été écrites en silence ?**

Prestataire de paiement 9, élevé. Courtier 6, élevé. Index de recherche 3 et incendie 3, tous deux moyens, même si l'incendie mérite une ligne énonçant ses hypothèses, car son impact est élevé. Traitez d'abord les deux premiers, avec des responsables et des dates dans le backlog normal. L'écriture silencieuse empêche la première voix ou la plus ancienne de diriger le groupe. Lorsque quatre personnes soulèvent indépendamment le même risque, c'est une preuve utile, et les désaccords révèlent des hypothèses différentes.

SOURCES

- Simon Brown: Risk-storming: https://riskstorming.com/
- Gary Klein: Performing a Project Premortem, Harvard Business Review (2007): https://hbr.org/2007/09/performing-a-project-premortem
- Neal Ford: Architectural Katas: https://nealford.com/katas/about.html
- Experiences with ALMA: Architecture-Level Modifiability Analysis: https://research.vu.nl/en/publications/experiences-with-alma-architecture-level-modifiability-analysis/
- The pillars of the AWS Well-Architected Framework: https://docs.aws.amazon.com/wellarchitected/latest/framework/the-pillars-of-the-framework.html
- What is AWS Well-Architected Tool?: https://docs.aws.amazon.com/wellarchitected/latest/userguide/intro.html
- Workloads (AWS Well-Architected Tool): https://docs.aws.amazon.com/wellarchitected/latest/userguide/workloads.html
- AWS Well-Architected Tool pricing: https://aws.amazon.com/well-architected-tool/pricing/
- Azure Well-Architected Framework pillars (Microsoft): https://learn.microsoft.com/en-us/azure/well-architected/pillars
- Google Cloud Well-Architected Framework: https://docs.cloud.google.com/architecture/framework

SECTION 11.3 SUR 11.8 · AVANCÉ · 27 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Fonctions d'aptitude et architecture évolutive

AVANT DE LIRE

Cette section s'appuie sur :

- 2.3 Scénarios d'attributs de qualité
- 3.8 Simplicité et mesure
- 10.5 CI/CD et déploiement

Termes déjà vus qu'elle utilise : Scénario d'attribut de qualité, Mesure de la réponse, Enregistrement de décision d'architecture (ADR), Arbre d'utilité, Dernier moment responsable

L'IDÉE EN BREF

Le tableau de bord d'une voiture comporte des voyants d'alerte pour ce qui ne doit jamais mal tourner sans qu'on le remarque : pression d'huile, température du moteur, freins. Vous n'attendez pas la révision annuelle pour découvrir que le moteur a surchauffé en mars. Une fonction d'aptitude (fitness function) est un voyant d'alerte pour une architecture logicielle. L'équipe décide quelles qualités comptent vraiment, comme la rapidité avec laquelle une commande est confirmée ou les parties du code qui ont le droit de communiquer entre elles, écrit une vérification automatique pour chacune et exécute ces vérifications à chaque modification du logiciel. Quand une vérification échoue, la modification est arrêtée ou quelqu'un est alerté, tant que le problème est encore petit.

CE QU'IL FAUT SAVOIR

- Une fonction d'aptitude architecturale est tout mécanisme qui fournit une évaluation objective d'une caractéristique d'architecture, idéalement automatisé et rattaché à un scénario d'attribut de qualité dont la mesure de la réponse devient le seuil.
- Les fonctions d'aptitude se classent selon qu'elles sont atomiques ou holistiques, déclenchées, continues ou temporelles, statiques ou dynamiques, automatisées ou manuelles, et intentionnelles ou émergentes ; un ensemble sain les combine.
- Les exemples typiques sont les règles de dépendance pour la structure, les tests de charge et les budgets de latence pour les performances, les analyses de vulnérabilités pour la sécurité, les expériences de chaos pour la résilience et les vérifications de compatibilité des schémas pour les données.

- L'architecture évolutive (Ford, Parsons, Kua et Sadalage) permet un changement guidé et incrémental selon de multiples dimensions ; les fonctions d'aptitude fournissent le guidage, et les petites unités déployables et les pipelines fournissent les incréments.
- Chaque fonction d'aptitude a besoin d'un responsable, d'une justification consignée et d'un seuil, et un échec impose une décision : corriger le code, ou changer la règle et consigner pourquoi.
- Commencez par quelques vérifications fiables pour les caractéristiques les plus prioritaires, appliquez aux systèmes existants un effet cliquet à partir d'une base de référence, et tenez les vérifications holistiques lentes à l'écart de chaque build.

De l'évaluation ponctuelle à la vérification continue

Une évaluation comme l'ATAM (section 11.1) est un instantané. Dès le lendemain, les développeurs commencent à modifier le système, et chaque modification peut défaire discrètement une décision sur laquelle l'évaluation s'appuyait : un import commode qui franchit une frontière de module, un appel synchrone supplémentaire sur le chemin de la commande, une bibliothèque qui présente une vulnérabilité connue. La section 1.7 a décrit le résultat : l'érosion, quand l'architecture est violée, et la dérive, quand elle est ignorée. Les revues périodiques en détectent une partie, avec des mois de retard. L'idée de cette section est de vérifier les décisions architecturales les plus importantes automatiquement et en continu, de la même façon que les tests unitaires vérifient le comportement.

Le terme vient de l'ouvrage de Neal Ford, Rebecca Parsons et Patrick Kua, *Building Evolutionary Architectures* (O'Reilly, 2017), dont la deuxième édition, avec Pramod Sadalage, est parue en novembre 2022. Ils l'ont emprunté au calcul évolutionnaire, où une fonction d'aptitude note la proximité d'une solution candidate avec l'objectif, afin que l'algorithme puisse conserver les meilleures variantes. En architecture, l'objectif est l'ensemble des caractéristiques que le système doit conserver pendant que tout le reste change.

Ce qu'est une fonction d'aptitude

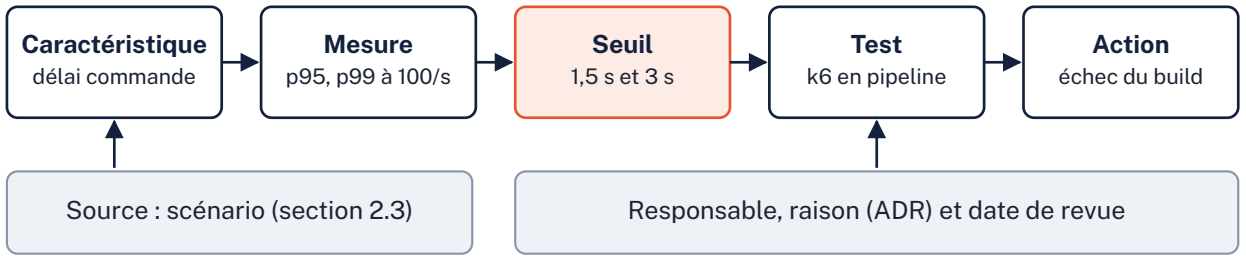
La deuxième édition définit une fonction d'aptitude (fitness function) architecturale comme tout mécanisme qui fournit une évaluation objective de l'intégrité d'une ou de plusieurs caractéristiques d'architecture. Trois termes de cette définition font tout le travail. *Tout mécanisme* : un test unitaire, une règle d'analyse statique, un test de charge, une sonde de supervision, une expérience de chaos ou même une revue manuelle planifiée conviennent tous. *Objective* : le résultat ne doit pas dépendre de la personne qui l'exécute ; « le code semble bien structuré » n'est pas une fonction d'aptitude, « aucun paquet de la couche domaine n'importe le

framework web » en est une. *Caractéristique d'architecture* : la vérification protège une qualité que l'architecture a pour raison d'être de fournir (performances, disponibilité, modifiabilité, sécurité, compatibilité des données), et non une règle métier, qui relève des tests ordinaires.

En pratique, une fonction d'aptitude utile possède quelques propriétés de plus. Elle est rattachée à une exigence, idéalement un scénario d'attribut de qualité dont la mesure de la réponse devient son seuil (section 2.3). Elle a un responsable qui décide de ce qui se passe quand elle échoue. Elle échoue bruyamment, en cassant un build ou en alertant la personne d'astreinte, plutôt qu'en alimentant un tableau de bord que personne ne lit. Et son seuil, ainsi que la raison de ce seuil, sont consignés, généralement dans un ADR, afin qu'une équipe ultérieure puisse distinguer une règle qui protège quelque chose d'important d'un chiffre arbitraire.

La section 3.8 a établi la distinction importante avec les métriques : une métrique décrit, une fonction d'aptitude décide. Une métrique de couplage sur un tableau de bord vous dit que la situation se dégrade ; une fonction d'aptitude encode une décision que l'équipe a déjà prise, si bien que l'enfreindre impose un choix conscient : corriger le code, ou changer la règle et consigner pourquoi.

Anatomie d'une fonction d'aptitude



Une fonction d'aptitude transforme la mesure de la réponse d'un scénario en un seuil, une vérification automatisée et une action, avec un responsable et une raison consignée.

Catégories de fonctions d'aptitude

Ford et ses coauteurs classent les fonctions d'aptitude selon plusieurs dimensions indépendantes. Cette classification est moins utile comme taxinomie que comme aide à la conception : se demander « celle-ci doit-elle être atomique ou holistique, déclenchée ou continue ? » révèle souvent une meilleure vérification que la première qui venait à l'esprit.

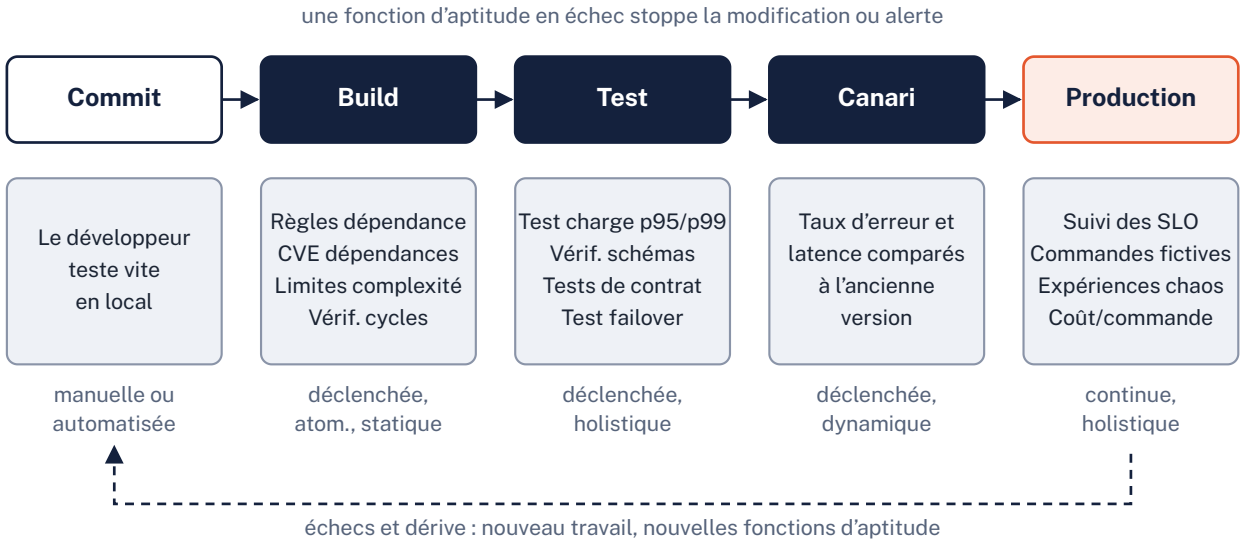
Dimensions des fonctions d'aptitude

Dimension	Options	Signification	Exemple
Portée	Atomique ou holistique	L'atomique vérifie une caractéristique dans un seul contexte ; l'holistique vérifie plusieurs caractéristiques en interaction dans un contexte partagé	Atomique : aucun cycle entre paquets. Holistique : latence des commandes mesurée avec chiffrement, journalisation d'audit et cache tous activés
Moment d'exécution	Déclenchée, continue ou temporelle	La déclenchée s'exécute sur un événement comme un commit, un build ou un déploiement ; la continue s'exécute en permanence sur le système en production ; la temporelle se déclenche une fois un délai écoulé, pour rappeler de réexaminer quelque chose	Déclenchée : analyse des dépendances à chaque build. Continue : alerte sur le taux de consommation du SLO en production. Temporelle : faire échouer le build quand une version figée d'un framework a plus de 12 mois
Type de résultat	Statique ou dynamique	La statique compare à un seuil fixe ; la dynamique ajuste le seuil au contexte, par exemple à l'échelle	Statique : p95 inférieur à 1,5 s. Dynamique : le p95 peut monter à 2 s au-delà de 150 commandes/s, comme convenu pour les pics extrêmes
Mode d'exécution	Automatisée ou manuelle	L'automatisée est préférable ; certaines caractéristiques, comme les exigences légales ou d'utilisabilité, demandent encore des personnes, idéalement selon un calendrier	Manuelle : revue trimestrielle de l'accessibilité au regard des WCAG 2.2 AA
Planification	Intentionnelle ou émergente	Les intentionnelles sont conçues en amont à partir d'exigences connues ; les émergentes sont ajoutées à mesure que l'équipe apprend ce qui compte	Émergente : une vérification du retard des consommateurs (consumer lag) ajoutée après un incident

Les vérifications holistiques méritent l'attention, car les qualités interagissent : les contrôles de sécurité ajoutent de la latence, le cache complique la fraîcheur des données, et un test atomique de chacun pris isolément peut réussir alors que la combinaison échoue. Les vérifications continues comptent, car certaines caractéristiques n'existent qu'en production : la latence réelle vécue par les utilisateurs, les modes de défaillance réels, le coût réel. Un

portefeuille sain combine des vérifications rapides, atomiques et déclenchées dans le build avec un plus petit nombre de vérifications holistiques plus lentes avant la mise en production et des vérifications continues en production.

Où s'exécutent les fonctions d'aptitude



Les vérifications structurelles peu coûteuses s'exécutent à chaque build, les vérifications holistiques avant la mise en production et les vérifications continues en production, et les échecs reviennent sous forme de nouveau travail.

SI VOUS DÉBUTEZ

Pas à pas : quand chaque petite vérification réussit

Le point qui fait le plus souvent trébucher ici est que des vérifications qui réussissent chacune de leur côté peuvent manquer une défaillance qui n'apparaît que lorsque tout s'exécute ensemble ; c'est pourquoi une équipe a besoin de fonctions d'aptitude à la fois atomiques et holistiques. Suivez le scénario de latence des commandes à travers les deux.

- Le scénario de la plateforme de livraison de repas dit qu'au pic du vendredi, à 100 commandes par seconde, 95 % des confirmations de commande doivent revenir en moins de 1,5 seconde.
- Supposons qu'en quelques mois l'équipe active le chiffrement et la journalisation d'audit sur le chemin de la commande. Chaque changement a sa propre vérification atomique, qui confirme qu'il fonctionne et que, mesuré seul, il n'ajoute qu'un peu de temps. Toutes les vérifications réussissent.
- Mais le temps s'additionne. Avec les deux activés sous la charge de pointe, leurs coûts se combinent avec tout le reste du chemin et, mettons, 6 % des confirmations prennent désormais plus de 1,5 seconde, soit plus que les 5 % autorisés par le scénario. Aucune vérification n'était fausse ; aucune ne regardait l'ensemble.
- Une vérification holistique le détecte : un test de charge qui envoie 100 commandes par seconde à un environnement semblable à la production, avec chiffrement, journalisation d'audit et cache tous activés, et qui échoue si le 95e percentile dépasse 1,5 seconde.
- Les vérifications holistiques ont des coûts. Un test de charge de dix minutes est trop lent pour chaque build : il s'exécute donc chaque nuit et avant chaque mise en production. Et un échec indique que l'ensemble est trop lent, pas quelle partie en est la cause ; l'exemple commenté plus loin dans cette section montre une façon de contourner ce problème.
- **La leçon :** des vérifications atomiques rapides à chaque build détectent la plupart des erreurs à peu de frais, mais ne voient pas comment les qualités interagissent. Ajoutez des vérifications holistiques moins nombreuses et plus lentes avant la mise en production, et des vérifications continues en production, là où l'expérience des utilisateurs réels peut être mesurée.

Les fonctions d'aptitude selon les qualités

La plupart des équipes commencent par la structure, parce que les vérifications structurelles sont peu coûteuses, rapides et déterministes, puis ajoutent des vérifications de performances, de sécurité, de résilience et de données à mesure que le pipeline mûrit. Le tableau donne des exemples typiques ; les outils changent, les modèles restent.

Exemples par qualité

Qualité	Fonction d'aptitude (fitness function)	Type	Outils typiques
Structure	Règles de couches et de dépendances, absence de cycles, modules qui ne s'utilisent entre eux que par des API publiques (section 3.4)	Atomique, déclenchée, statique	ArchUnit, import-linter, dependency-cruiser, vérification Spring Modulith
Performances	Test de charge de bout en bout avec seuils de percentiles ; budgets de latence par service dans le pipeline propre à chaque service	Holistique, déclenchée	k6, Gatling, JMeter, Locust ; seuils qui font échouer le job
Sécurité	Faire échouer le build sur les dépendances présentant des vulnérabilités connues de gravité élevée ou critique ; aucun secret dans le code ; images de conteneur provenant uniquement du registre approuvé	Atomique, déclenchée	pip-audit, npm audit, OWASP Dependency-Check, Trivy, détecteurs de secrets ; politique en tant que code (section 11.4)
Résilience	Expériences de chaos qui arrêtent des instances ou ralentissent des dépendances et vérifient que l'état stable se maintient (section 9.8)	Holistique, déclenchée ou continue	Toxiproxy, Chaos Toolkit, services d'injection de pannes des fournisseurs de cloud
Données	Vérifications de compatibilité des schémas sur les événements et les API ; migrations répétées sur des données de la taille de la production (section 7.7)	Atomique, déclenchée	Vérifications de compatibilité du registre de schémas, buf breaking, tests de contrat pilotés par le consommateur
Exploitabilité et coût	Chaque service expose des contrôles de santé et des métriques standard ; le coût par commande reste sous une cible	Atomique ou continue	Vérifications du pipeline sur les modèles de service ; tableaux de bord de coûts avec alertes

Architecture évolutive et changement incrémental

Les fonctions d'aptitude sont l'une des deux moitiés de ce que Ford, Parsons, Kua et Sadalage appellent l'architecture évolutive : une architecture qui permet un changement guidé et incrémental selon de multiples dimensions. Chaque partie de cette formule est une exigence de conception.

Le *changement incrémental* a un versant développement et un versant exploitation. Les développeurs doivent pouvoir effectuer de petits changements en toute sécurité, ce qui dépend de la modularité et d'un couplage approprié (chapitre 3), ainsi que du fait que chaque partie déployable indépendamment, le quantum d'architecture de la section 5.1, reste assez petite pour être modifiée sans coordination avec les autres. L'exploitation doit pouvoir mettre en production de petits changements fréquemment et en toute sécurité, ce qui dépend d'un pipeline de déploiement, d'un déploiement progressif et d'un retour arrière rapide (section 10.5). *Guidé* : c'est là qu'interviennent les fonctions d'aptitude ; elles définissent ce que signifie « apte » et empêchent les incréments de le dégrader. *Selon de multiples dimensions* rappelle que l'architecture n'est pas seulement la structure du code : les données, la sécurité, l'exploitation, la structure des équipes et le coût évoluent tous et ont tous besoin d'être protégés.

Cette approche change la façon dont les architectes pensent l'avenir. Au lieu de prédire chaque exigence et de prévoir à l'avance la flexibilité correspondante, ce qui, comme l'a montré la section 3.8, est coûteux et souvent erroné, l'équipe garde le système facile à modifier et ne protège que les caractéristiques qui comptent, en reportant les autres décisions au dernier moment responsable (section 1.6). Les critiques soulignent de vraies limites. Toutes les qualités ne peuvent pas être vérifiées automatiquement ou à peu de frais ; des seuils mal choisis créent un bruit que les équipes apprennent à ignorer ; et certaines décisions, comme une base de données ou un fournisseur de cloud, restent des portes à sens unique, aussi bon que soit le pipeline. L'architecture évolutive est une discipline qui consiste à garder réversibles les parties réversibles et protégées les parties importantes, et non la promesse que tout pourra être changé plus tard.

EXEMPLE COMMENTÉ

Exemple commenté : trois scénarios du chapitre 2 transformés en fonctions d'aptitude

La section 2.8 a construit un arbre d'utilité pour la plateforme de livraison de repas. Après que l'ATAM de la section 11.1 a conduit à une autorisation de paiement asynchrone, l'équipe transforme trois de ses feuilles en fonctions d'aptitude : l'une déclenchée et holistique, l'une sous forme d'expérience de chaos, l'une structurelle.

Scénario 1 : latence des commandes (H,H)	Au pic du vendredi, à 100 commandes/s, 95 % des confirmations de commande reviennent en moins de 1,5 s et 99 % en moins de 3 s
Fonction d'aptitude 1	Holistique, déclenchée : un test de charge k6 sur un environnement semblable à la production, chaque nuit et avant chaque mise en production, qui échoue si le p95 ou le p99 dépasse le seuil ou si plus de 0,1 % des requêtes échouent. Complétée par une supervision continue en production des mêmes percentiles sur le trafic réel
Budgets de latence	Les 1,5 s sont répartis en budgets : réseau mobile et TLS 400 ms, passerelle 50 ms, commandes 250 ms, tarification 100 ms, écritures en base de données 150 ms, ce qui laisse 550 ms de marge maintenant que l'autorisation de carte est sortie du chemin. Le pipeline de chaque service exécute un test de charge atomique du composant par rapport à son propre budget, de sorte qu'une régression est détectée dans le service qui l'a causée
Scénario 2 : disponibilité du paiement (H,H)	Lorsque le prestataire de paiement dépasse son délai d'expiration pendant le pic, les commandes sont acceptées et mises en file d'attente, aucune commande confirmée n'est perdue, et les clients sont informés en moins de 10 s
Fonction d'aptitude 2	Expérience de chaos holistique, chaque semaine en staging et chaque trimestre sous forme de game day en production avec un faible rayon d'impact : on fait se bloquer le prestataire ; les commandes doivent être acceptées comme en attente en moins de 10 s et toutes doivent être finalisées après le rétablissement. Responsable : équipe Commandes
Scénario 3 : nouvel éditeur de caisse (M,L)	Un nouvel éditeur de point de vente (caisse) pour restaurants est intégré en 5 jours-personnes par l'ajout d'un seul adaptateur
Fonction d'aptitude 3	Atomique, déclenchée, statique : un contrat import-linter interdit aux modules de domaine et d'API des restaurants ainsi qu'au paquet des commandes d'importer le SDK d'un éditeur, et un second contrat empêche les adaptateurs de s'importer entre eux, si bien que le changement reste local. Complétée par une vérification manuelle : l'effort de chaque intégration réelle d'éditeur est consigné et comparé à l'objectif de 5 jours
Consigné avec chacune	Scénario source, seuil, responsable, ADR qui l'explique et date de revue

Chaque mesure de la réponse est devenue un seuil sans réinterprétation : c'est le bénéfice d'avoir bien rédigé les scénarios dès le départ.

Le scénario de modifiabilité ne peut pas être entièrement automatisé : la règle structurelle protège les conditions d'un changement en 5 jours, et la mesure manuelle vérifie le résultat.

Les budgets de latence transforment un chiffre holistique unique en vérifications atomiques qui signalent une régression à la bonne équipe.

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

export const options = {
  scenarios: {
    friday_peak: { executor: 'constant-arrival-rate', rate: 100, timeUnit: '1s',
                   duration: '10m', preAllocatedVUs: 300 },   // 100 orders per second
  },
  thresholds: {   // the scenario's response measure, as pass/fail rules
    'http_req_duration{name:place_order}': ['p(95)<1500', 'p(99)<3000'],
    'http_req_failed{name:place_order}': ['rate<0.001'],
  },
};

const order = { restaurantId: 'r-42', items: [{ sku: 'pad-thai', qty: 1 }] };

export default function () {
  const res = http.post(`${__ENV.BASE_URL}/orders`, JSON.stringify(order), {
    headers: { 'Content-Type': 'application/json' }, tags: { name: 'place_order' } });
  check(res, { 'order accepted': (r) ⇒ r.status ═══ 201 || r.status ═══ 202 });
}
```

Fonction d'aptitude 1 : un test de charge k6 dont les seuils sont la mesure de la réponse du scénario. k6 se termine avec une erreur quand un seuil n'est pas respecté, ce qui fait échouer le job du pipeline.

```
# Fitness function: checkout keeps accepting orders when payments hang (illustrative)
# Runs weekly in staging; toxiproxy, api and ledger are test fixtures.
def test_orders_survive_payment_provider_outage(toxiproxy, api, ledger):
    toxiproxy.add_latency("payment-provider", latency_ms=30_000)   # provider hangs
    placed = [api.place_order(sample_order()) for _ in range(200)]

    assert all(r.status in (201, 202) for r in placed)           # still accepted
    assert all(r.json()["state"] == "PENDING_PAYMENT" for r in placed)
    assert max(r.elapsed_seconds for r in placed) < 10           # told within 10 s

    toxiproxy.remove_toxics("payment-provider")                   # provider recovers
    ledger.wait_until_settled(timeout_seconds=300)
    assert ledger.lost_orders(placed) == []                       # none lost
```

Fonction d'aptitude 2 : une expérience de chaos écrite sous forme de test (à titre d'illustration ; les fixtures encapsulent un proxy d'injection de pannes, l'API et le grand livre des paiements).

```
# .importlinter: only the POS adapters may use vendor SDKs
[importlinter]
root_package = fooddelivery
include_external_packages = True

[importlinter:contract:pos-vendors-isolated]
name = Vendor POS SDKs are used only inside the POS adapters
type = forbidden
source_modules =
    fooddelivery.restaurant.domain
    fooddelivery.restaurant.api
    fooddelivery.ordering
forbidden_modules =
    tillpro_sdk
    kitchenlink

[importlinter:contract:adapters-independent]
name = POS adapters do not import each other
type = independence
modules =
    fooddelivery.restaurant.pos_adapters.tillpro
    fooddelivery.restaurant.pos_adapters.kitchenlink
```

Fonction d'aptitude 3 : des contrats import-linter qui maintiennent le code propre à chaque éditeur à l'intérieur des adaptateurs, exécutés avec lint-imports à chaque build.

La compatibilité des données est protégée de la même manière. Avant qu'une modification de l'événement `OrderPlaced` puisse être fusionnée, le pipeline demande au registre de schémas si le nouveau schéma est compatible avec la dernière version enregistrée selon la règle de compatibilité du sujet, et échoue s'il ne l'est pas :

```
# Data fitness function: block the merge if the new event schema would break consumers
curl -s -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \
  --data @build/order-placed-v7.request.json \
  "$SCHEMA_REGISTRY_URL/compatibility/subjects/order-placed-value/versions/latest" \
  | jq -e '.is_compatible == true'
```

Une fonction d'aptitude de données qui utilise l'endpoint de compatibilité d'un registre de schémas ; jq -e renvoie un code de sortie non nul, ce qui fait échouer le job, quand le schéma est incompatible.

Concevoir des fonctions d'aptitude durables

Commencez petit, et par l'important. Choisissez les trois à cinq caractéristiques les mieux classées dans l'arbre d'utilité, écrivez une fonction d'aptitude pour chacune et assurez-vous qu'elles sont rapides et fiables avant d'en ajouter d'autres. Une douzaine de vérifications dignes de confiance valent mieux qu'une centaine qui échouent au hasard.

Appliquez un effet cliquet aux systèmes existants. Quand une base de code enfreint déjà une règle, consignez les violations actuelles comme base de référence et ne faites échouer le build que sur les nouvelles ; resserrez la base de référence à mesure que l'équipe rembourse la dette. Il en va de même pour les performances : faites échouer le build en cas de régression supérieure à une marge convenue par rapport à la dernière version, tout en progressant vers l'objectif du scénario.

Rendez les seuils explicites et attribuez-leur un responsable. Conservez les seuils dans une configuration sous contrôle de version, à côté d'une référence à l'ADR qui les justifie, avec un responsable nommé. Quand une fonction d'aptitude échoue, le responsable choisit entre corriger le code et changer la règle, et la décision est consignée. Traitez une fonction d'aptitude fréquemment contournée comme un constat : soit la règle est mauvaise, soit l'architecture subit une pression réelle.

Gardez le pipeline rapide. Les vérifications atomiques ont leur place dans chaque build ; les vérifications holistiques lourdes, comme les tests de charge de dix minutes, s'exécutent chaque nuit, avant la mise en production ou sur la version candidate. Les vérifications continues en production demandent le même soin que n'importe quelle alerte : liées à l'impact sur les utilisateurs, accompagnées d'un runbook, et silencieuses quand tout va bien.

ATTENTION

Anti-patrons des fonctions d'aptitude

- Vérifications instables : un test de charge exécuté sur une infrastructure partagée et bruyante échoue au hasard, si bien que les équipes prennent l'habitude de le relancer jusqu'à ce qu'il réussisse et cessent d'y croire.
- Manipuler la mesure : quand un chiffre devient une cible, on optimise le chiffre (loi de Goodhart, section 3.8) ; préférez des vérifications liées à des scénarios réels plutôt que des métriques abstraites.
- Des seuils sans raison : personne ne sait pourquoi la limite est de 15, si bien que personne n'ose la modifier ou que tout le monde l'ignore.
- Des vérifications qui ne font que rendre compte : un tableau de bord que personne ne regarde n'est pas une fonction d'aptitude.
- Automatiser le facile et ignorer l'important : des dizaines de règles de style, mais aucune vérification de la latence du chemin de la commande ni de la solution de repli du paiement.
- Les fonctions d'aptitude comme substitut à la réflexion : elles protègent des décisions déjà prises ; elles ne vous disent pas quelles décisions prendre.

Questions fréquentes

Les fonctions d'aptitude ne sont-elles que des tests ?
Beaucoup sont mises en œuvre sous forme de tests, mais leur objet est différent. Les tests ordinaires vérifient que les fonctionnalités se comportent correctement ; les fonctions d'aptitude vérifient des caractéristiques d'architecture comme la structure, la latence, la résilience ou la compatibilité. Certaines, comme les sondes de supervision en production et les revues manuelles, ne sont pas du tout des tests.

Qui écrit les fonctions d'aptitude : les architectes ou les développeurs ?
Les deux. Les architectes identifient généralement les caractéristiques et les seuils, et écrivent souvent les premières vérifications, ce qui est un bon moyen de garder les mains dans le code (section 1.4). Les équipes en sont propriétaires et les maintiennent comme n'importe quel autre code.

Quel est le lien entre les fonctions d'aptitude et les SLO ?
Un SLO assorti d'une alerte sur son budget d'erreur est une fonction d'aptitude continue et holistique pour la disponibilité ou la latence en production (section 9.5). Les fonctions d'aptitude du pipeline cherchent à détecter les mêmes régressions avant qu'elles n'atteignent les utilisateurs.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Quelle est la différence entre une métrique et une fonction d'aptitude ?**

Une métrique décrit une propriété ; une fonction d'aptitude encode une décision avec un seuil et une action, si bien que l'enfreindre fait échouer un build ou déclenche une alerte et impose un choix conscient.

▼ **Donnez un exemple de fonction d'aptitude atomique et un exemple de fonction d'aptitude holistique.**

Atomique : une règle qui interdit les imports de la couche web dans le domaine. Holistique : un test de charge qui mesure la latence des commandes avec la sécurité, la journalisation et le cache tous actifs.

▼ **Qu'apporte un budget de latence à une fonction d'aptitude de latence de bout en bout ?**

Il répartit l'objectif de bout en bout en limites par service, que chaque service vérifie dans son propre pipeline, de sorte qu'une régression est détectée dans le service qui l'a causée.

▼ **Quelles sont les trois parties de la définition de l'architecture évolutive ?**

Elle permet un changement guidé (par les fonctions d'aptitude), incrémental (petits changements et mises en production sûrs) et selon de multiples dimensions (code, données, sécurité, exploitation, équipes, etc.).

▼ **À vous : Transformez chacun en fonction d'aptitude, avec déclencheur et seuil : « 95 % des confirmations de commande en moins de 1,5 s à 100 commandes/s » et « Commandes ne doit jamais importer les éléments internes de Paiements ». Où chacune s'exécuterait-elle, et que se passe-t-il quand l'une échoue ?**

Latence : un test de charge holistique et dynamique à 100 commandes par seconde, en échec si le p95 dépasse 1,5 s, exécuté chaque nuit ou avant la mise en production car il est lent. Règle de dépendance : une vérification atomique et statique avec ArchUnit ou import-linter, en échec sur tout import interdit, exécutée à chaque build. Un échec impose une décision : corriger le code, ou changer la règle et consigner pourquoi. Chacune a besoin d'un responsable et d'un lien vers son scénario.

SOURCES

- Building Evolutionary Architectures, 2nd edition (O'Reilly): https://oreilly.com/library/view/building-evolutionary-architectures/9781492097532
- Evolutionary Architecture: book site with fitness function katas: https://evolutionaryarchitecture.com/
- Grafana k6 documentation: Thresholds: https://grafana.com/docs/k6/latest/using-k6/thresholds/
- import-linter documentation: Forbidden contracts: https://import-linter.readthedocs.io/en/stable/contract_types/forbidden/

SECTION 11.4 SUR 11.8 · AVANCÉ · 28 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Gouvernance de l'architecture dans les organisations agiles

AVANT DE LIRE

Cette section s'appuie sur :

- 1.6 Décisions et compromis
- 4.5 Enregistrements de décision
- 11.3 Fonctions d'aptitude

Termes déjà vus qu'elle utilise : Enregistrement de décision d'architecture (ADR), Fonction d'aptitude (fitness function), Architecture d'entreprise

L'IDÉE EN BREF

Une ville ne peut pas demander au maire d'approuver chaque nouvelle cuisine. Elle fixe plutôt quelques principes (garder les rues praticables à pied), des règles de construction que des inspecteurs vérifient d'office, et un petit nombre de décisions qui exigent vraiment un comité, comme un nouveau pont. Les habitants peuvent faire la plupart des choses eux-mêmes, mais pour tout ce qui touche leurs voisins, on attend d'eux qu'ils consultent d'abord ces voisins. La gouvernance de l'architecture dans les organisations agiles fonctionne de la même façon : les équipes décident elles-mêmes de la plupart des choses, des vérifications automatiques font respecter les règles qu'une machine peut vérifier, et les décisions plus importantes sont prises après consultation des personnes qu'elles concernent.

CE QU'IL FAUT SAVOIR

- La gouvernance de l'architecture veille à ce que les décisions significatives soient bien prises, en cohérence avec les objectifs de l'organisation, et réellement suivies ; ses finalités comprennent l'alignement, le risque et la conformité, la limitation de la diversité inutile, la protection des qualités partagées et la diffusion des enseignements.
- Les principes donnent une direction durable, les normes rendent obligatoires des choix précis, les garde-fous maintiennent automatiquement le travail dans des limites, et les points de validation arrêtent le travail en attente d'approbation ; privilégiez les garde-fous là où les règles sont objectives et gardez des points de validation précoces et peu nombreux.
- Les principes à la manière de TOGAF ont un nom, un énoncé, une justification et des implications ; un bon principe pourrait raisonnablement être tranché dans l'autre sens et règle de vraies décisions.

- Dans le processus de consultation (Andrew Harmel-Law, 2021), n'importe qui peut prendre une décision architecturale après avoir sollicité l'avis des personnes concernées et d'experts, et la consigne dans un ADR ; un forum consultatif donne de la visibilité mais ne décide pas.
- Un radar technologique place les technologies dans des anneaux comme Adopt, Trial, Assess et Hold, avec des raisons écrites ; un radar interne consigne l'expérience propre de l'organisation et oriente les nouveaux choix.
- La politique en tant que code (par exemple Open Policy Agent avec Rego) et les fonctions d'aptitude automatisent la conformité et produisent des preuves d'audit en sous-produit.

À quoi sert la gouvernance de l'architecture

La gouvernance de l'architecture est l'ensemble des pratiques par lesquelles une organisation s'assure que ses décisions techniques significatives sont bien prises, cohérentes avec ses objectifs et réellement suivies. Elle répond à quatre questions : qui peut prendre quelles décisions, comment les prendre, quelle direction suivre, et comment savoir si le résultat est conforme.

Ses finalités sont pratiques. La gouvernance aligne la technologie sur la stratégie, afin que cinquante équipes ne choisissent pas cinquante façons de stocker les données clients. Elle gère le risque et la conformité : la protection des données, la sécurité, l'accessibilité et la réglementation sectorielle exigent toutes des preuves que les règles sont respectées. Elle limite la diversité inutile, car chaque base de données, langage ou produit de messagerie supplémentaire exige des compétences, des correctifs et une astreinte. Elle protège les qualités dont aucune équipe n'est seule propriétaire, comme la latence de bout en bout, la sécurité et le coût. Et elle diffuse les enseignements, afin qu'une leçon payée par une équipe ne soit pas payée à nouveau par la suivante.

Les organisations agiles ressentent ici une tension. Le principe du Manifeste agile selon lequel les meilleures architectures émergent d'équipes auto-organisées (section 1.8) tire vers l'autonomie ; l'échelle, la réglementation et l'infrastructure partagée tirent vers l'alignement. Les descriptions de la culture d'ingénierie de Spotify par Henrik Kniberg ont popularisé l'expression *autonomie alignée* (aligned autonomy) pour désigner l'objectif : des équipes libres de décider du comment, dans une compréhension partagée du quoi et du pourquoi. La gouvernance est la mécanique qui rend cela possible. Mal faite, elle devient le lent processus d'approbation de la section 11.2 ; bien faite, les équipes la remarquent à peine, parce que la bonne façon de faire est aussi la plus facile.

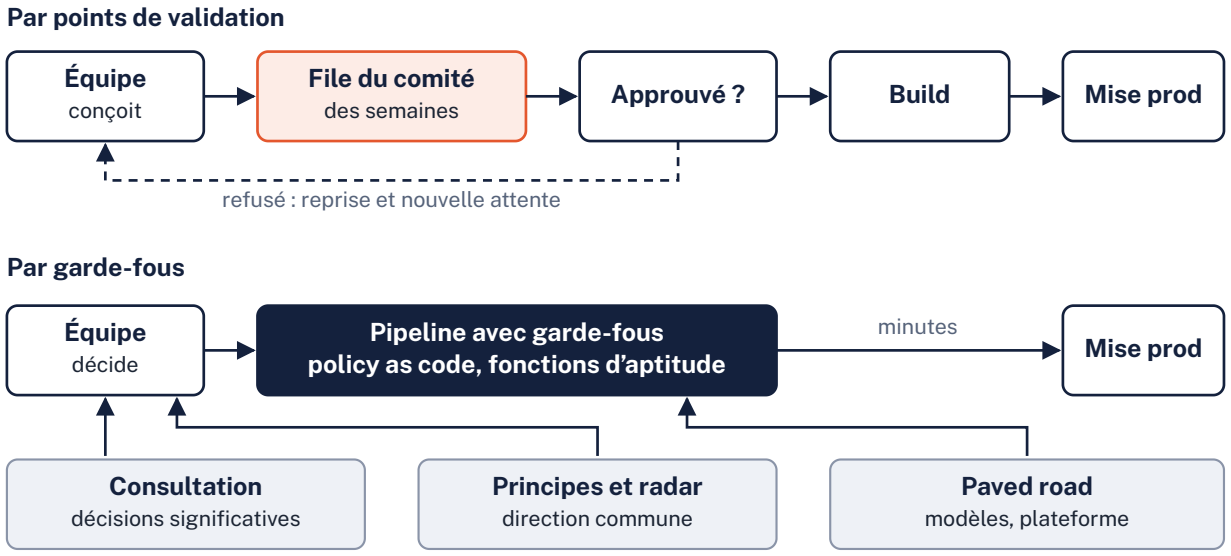
Principes, normes, garde-fous et points de validation

La gouvernance utilise quatre types d'instruments, qui diffèrent par leur degré de contrainte et par le moment où ils agissent. Les principes d'architecture expriment une direction durable et laissent place au jugement. Les normes sont des choix précis et obligatoires (tous les services exposent des traces OpenTelemetry ; toutes les API publiques sont décrites en OpenAPI). Les garde-fous (guardrails) maintiennent les équipes dans des limites sûres pendant qu'elles travaillent, généralement de façon automatique et continue. Les points de validation (gates) arrêtent le travail jusqu'à ce que quelqu'un l'approuve.

Les instruments de la gouvernance

Instrument	Exemple	Force	Risque
Principe	Chaque donnée métier a exactement un service propriétaire	Guide de nombreuses décisions, y compris imprévues	Trop vague pour agir, ou ignoré en pratique
Norme	Tous les services émettent des traces OpenTelemetry et des journaux JSON structurés	Cohérence, outillage partagé, intégration facile des nouveaux arrivants	Devient obsolète ; bloque de meilleures options ; les exceptions s'accumulent
Garde-fou (guardrail)	Le pipeline rejette les images de conteneur provenant de registres non approuvés	Continu, rapide, objectif, sans file d'attente	Ne fonctionne que pour les règles qu'une machine peut vérifier ; peut être contourné si personne n'en est responsable
Point de validation	Validation par la sécurité avant le traitement d'une nouvelle catégorie de données à caractère personnel	Un jugement humain responsable pour les décisions irréversibles à fort enjeu	Retards, retour tardif, contournement et approbation de pure forme

Points de validation ou garde-fous



Un point de validation fait attendre l'approbation à chaque conception significative ; les garde-fous vérifient en continu les règles objectives et laissent les personnes libres pour les décisions qui demandent du jugement.

Le consensus moderne, sans être universel, consiste à préférer les garde-fous aux points de validation partout où une règle peut être énoncée objectivement, à réserver les points de validation aux quelques décisions qui exigent réellement un jugement humain responsable, et à les tenir tôt et dans un temps limité. Une paved road (chemin balisé) (terme de Netflix ; Spotify parle de golden path) complète les deux : quand la façon prise en charge de construire un service inclut déjà les images approuvées, la journalisation, l'analyse de sécurité et le déploiement, la plupart des équipes se conforment sans qu'on le leur dise, et l'effort de gouvernance peut se concentrer sur les écarts délibérés. La section 10.7 présente les plateformes qui fournissent de tels chemins.

SI VOUS DÉBUTEZ

Pas à pas : une même règle comme point de validation et comme garde-fou

L'idée la plus difficile de cette section est de choisir le bon instrument pour chaque règle : pourquoi une règle qu'une machine peut vérifier a sa place dans un garde-fou, et pourquoi quelques décisions ont encore besoin d'un point de validation. Suivez deux règles du tableau ci-dessus.

- Prenez la règle selon laquelle chaque image de conteneur doit provenir du registre analysé de l'organisation. Supposons qu'elle soit appliquée par un point de validation : chaque équipe dresse la liste de ses images pour approbation par le comité de revue mensuel.
- L'équipe attend un créneau pendant des semaines. Le comité vérifie la liste une seule fois, si bien qu'une image modifiée le mois suivant n'est jamais vérifiée. Et comme la file est lente, les équipes prennent l'habitude de qualifier leurs changements de mineurs pour l'éviter, comme l'a décrit la section 11.2.
- Appliquez maintenant la même règle sous forme de garde-fou : le pipeline rejette tout déploiement dont l'image provient d'ailleurs. Il vérifie chaque mise en production, en quelques minutes, sans file d'attente, et son journal constitue une preuve pour un auditeur.
- Cela fonctionne parce que la règle est objective : une machine peut répondre par oui ou par non. Le garde-fou a tout de même besoin d'un responsable et d'une procédure d'exception, sinon les gens trouveront des moyens de le contourner.
- Prenez une seconde règle : une validation par la sécurité avant que la plateforme ne traite une nouvelle catégorie de données à caractère personnel. Aucune machine ne peut juger si la finalité et les risques sont acceptables : une personne responsable doit le faire, et un point de validation est approprié. Tenez-le tôt, limitez-le dans le temps et précisez à l'avance les éléments de preuve dont il a besoin.
- **La leçon :** si une machine peut vérifier une règle, faites-en un garde-fou ; réservez les points de validation aux quelques décisions à fort enjeu qui exigent un jugement humain responsable, et tenez-les tôt. Les décisions qui demandent du jugement mais pas d'approbation passent par le processus de consultation, présenté plus loin dans cette section.

Rédiger des principes d'architecture

TOGAF, le cadre d'architecture d'entreprise de l'Open Group (le TOGAF Standard, 10th Edition, 2022), fournit un modèle de principes largement utilisé, en quatre parties. Le **nom** est court et facile à retenir. L'**énoncé** exprime la règle en une ou deux phrases sans ambiguïté. La **justification** explique le bénéfice pour l'entreprise, en des termes qu'un commanditaire non

technique pourrait accepter. Les **implications** disent ce que le principe exige en pratique : ce qui doit être fait, ce qu'il coûte et ce qu'il exclut. TOGAF énumère aussi les critères d'un bon ensemble de principes : compréhensible, robuste (assez précis pour trancher les cas difficiles), complet, cohérent et stable.

Deux tests repèrent la plupart des principes faibles. D'abord, une organisation raisonnable pourrait-elle choisir l'inverse ? « Les systèmes doivent être sécurisés » échoue ; personne ne choisirait l'insécurité. « Préférer l'achat au développement pour les capacités génériques » réussit, car certaines organisations préfèrent à juste titre développer. Ensuite, le principe change-t-il une décision réelle ? Si personne ne peut citer un choix qu'il trancherait, c'est un slogan. Andrew Harmel-Law demande un ensemble ni trop petit pour couvrir les besoins essentiels ni trop grand pour que les équipes s'en souviennent, issu des équipes elles-mêmes et lié à la stratégie de l'entreprise, plutôt que des maximes générales de programmation ou des pratiques. Des principes moins nombreux et plus tranchants sont retenus et utilisés ; les longues listes ne le sont pas.

EXEMPLE COMMENTÉ

Exemple commenté : trois principes pour la plateforme de livraison de repas

Rédigés par les architectes avec les responsables d'équipe, discutés au forum consultatif et publiés avec le radar. Chacun suit la structure TOGAF.

Nom de P1	**Un propriétaire pour chaque fait**
Énoncé	Chaque type de donnée métier (commandes, cartes, positions des livreurs, paiements) a exactement un service propriétaire, qui est seul à l'écrire ; les autres utilisent son API ou ses événements.
Justification	Des données partagées modifiables couplent les mises en production des équipes et rendent les erreurs intraçables (section 7.1). Une propriété claire permet aux équipes de modifier leurs schémas et de déployer indépendamment, et c'est ainsi que l'entreprise livre chaque semaine.
Implications	Aucun service ne lit ni n'écrit les tables d'un autre. Le reporting utilise les événements publiés ou un entrepôt de données. Toute nouvelle donnée doit avoir un propriétaire nommé avant d'être stockée. Une fonction d'aptitude vérifie les identifiants de base de données de chaque service.
Nom de P2	**Asynchrone entre domaines**
Énoncé	Les domaines s'intègrent par défaut au moyen d'événements publiés ; les appels synchrones entre domaines ne sont utilisés que lorsque l'utilisateur attend la réponse.
Justification	Les chaînes synchrones multiplient l'indisponibilité et ajoutent de la latence (section 6.1) ; le pic du vendredi et les pannes des partenaires sont les plus grands risques opérationnels de l'entreprise.
Implications	Les équipes conçoivent pour la cohérence à terme et des consommateurs idempotents. Chaque appel synchrone entre domaines a un délai d'expiration, une solution de repli et un budget de latence. Davantage d'investissement opérationnel dans le courtier et le traçage.
Nom de P3	**Acheter le générique, développer le cœur, rester portable**
Énoncé	Les capacités génériques (identité, paiements, e-mail, supervision) sont achetées ou utilisées comme services managés ; les capacités différenciantes (optimisation de l'affectation, outils pour les restaurants) sont développées. La logique métier reste derrière des ports, et non à l'intérieur des services propriétaires d'un fournisseur.
Justification	Le temps d'ingénierie est la ressource la plus rare et doit aller là où l'entreprise affronte la concurrence (sous-domaines cœurs, section 5.7), tout en évitant une dépendance au fournisseur qui affaiblirait le pouvoir de négociation (section 10.8).

Implications	La question acheter ou développer est posée explicitement dans chaque ADR concerné. Les services managés sont accessibles via des adaptateurs. Certaines fonctionnalités managées qui créent une forte dépendance au fournisseur sont refusées, même quand elles sont pratiques.

Chaque principe tranche des choix réels et récurrents, et une organisation raisonnable pourrait choisir autrement.

Les implications comprennent des coûts, pas seulement des avantages.

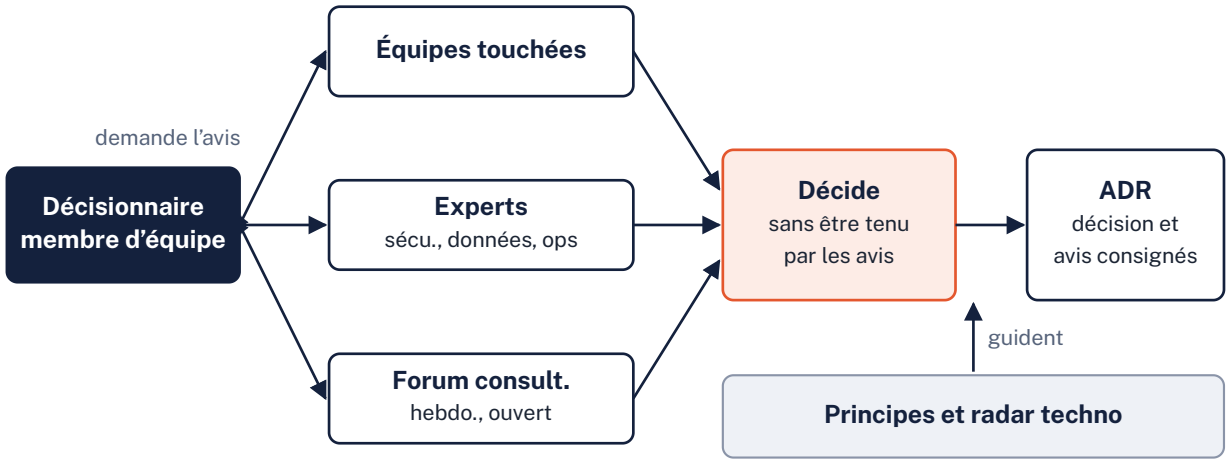
P1 et P2 peuvent en partie être appliqués sous forme de garde-fous ; P3 exige un jugement humain, il est donc appliqué au moyen du processus de consultation.

Prise de décision décentralisée : le processus de consultation

Si les décisions ne passent plus par un comité, qui les prend ? L'article d'Andrew Harmel-Law *Scaling the Practice of Architecture, Conversationally* (martinfowler.com, décembre 2021) a proposé une réponse empruntée aux organisations autogérées : le processus de consultation (advice process). Sa règle unique est que n'importe qui peut prendre une décision architecturale, à condition de solliciter, avant de décider, l'avis de toutes les personnes qui seront sensiblement concernées et de personnes ayant une expertise pertinente. La personne qui décide doit entendre et consigner les avis, mais n'est pas tenue de les suivre ; le but est de recueillir des contributions larges, pas d'obtenir un consensus. La responsabilité reste à la personne ou à l'équipe qui décide, et c'est ce qui fait fonctionner la règle : les personnes qui savent qu'elles assument les conséquences sollicitent sérieusement des avis.

Harmel-Law décrit quatre éléments d'appui. Les enregistrements de décision (ADR, section 4.5) sont l'outil de réflexion et de consignation, et incluent les avis reçus, de qui et quand. Un forum consultatif d'architecture, généralement hebdomadaire, d'environ une heure et ouvert à tous, donne de la visibilité aux nouveaux spikes, aux décisions proposées et à celles qui sont réexaminées ; point crucial, il donne des avis mais ne décide pas. Des principes d'architecture issus des équipes donnent une direction commune. Et un radar technologique interne montre le paysage technologique actuel de l'organisation et sa position à son égard.

Le processus de consultation



Les avis viennent des personnes concernées et des experts ; la décision et les avis sont consignés ensemble, et la responsabilité reste à la personne qui décide.

Le processus de consultation a des modes de défaillance qu'il vaut la peine de connaître. Les décisionnaires peuvent ne consulter que les personnes susceptibles d'être d'accord ; une liste publiée de qui doit être consulté pour quels types de décisions (la sécurité pour tout ce qui touche aux données à caractère personnel, l'équipe plateforme pour toute nouvelle infrastructure) y remédie. Les avis peuvent être sollicités trop tard, une fois le code écrit ; faire de l'ADR proposé le début du travail, et non sa fin, y remédie. Et certaines personnes trouvent d'abord inconfortable que le forum ne décide pas. Dans les environnements réglementés, le processus peut coexister avec un petit nombre de points de validation formels, appliqués uniquement là où la loi ou l'appétence au risque de l'organisation l'exige.

Radars technologiques

Thoughtworks publie son Technology Radar deux fois par an depuis 2010. Chaque édition place une centaine de blips (techniques, outils, plateformes, ainsi que langages et frameworks : les quatre quadrants) dans des anneaux qui expriment une recommandation. Adopt signifie éprouvé et recommandé ; Trial, digne d'être poursuivi sur un projet capable d'assumer le risque ; Assess, digne d'être exploré pour en comprendre l'effet ; et l'anneau le plus extérieur, appelé Hold pendant la majeure partie de l'histoire du radar et Caution dans ses éditions les plus récentes, signifie avancer avec prudence. Chaque blip est accompagné d'une courte justification écrite, qui en est la partie la plus précieuse.

De nombreuses organisations construisent leur propre radar, pour elles-mêmes plutôt que pour l'industrie. Thoughtworks fournit un outil open source, Build Your Own Radar, qui génère un radar à partir d'une feuille de calcul ou d'un fichier CSV avec les colonnes name, ring, quadrant,

isNew et description. La valeur tient moins à l’image qu’au processus : les équipes proposent des blips, un petit groupe réunissant des représentants de nombreuses équipes en discute, les anneaux sont attribués avec une raison écrite, et le radar est actualisé chaque trimestre ou chaque semestre. Un anneau Hold est un outil de gouvernance doux mais efficace : il dit « pas de nouveaux usages » sans interdire ce qui existe déjà. Un radar diffère d’une liste de technologies approuvées en ce qu’il est explicitement provisoire et argumenté, et qu’il consigne ce que l’organisation a appris, y compris ses erreurs.

EXEMPLE COMMENTÉ

Exemple commenté : une entrée du radar et l'enregistrement de décision qui la sous-tend

L'équipe Commandes veut remplacer son relais d'outbox par interrogation (polling) (section 6.7) par une capture des données modifiées (CDC) avec Debezium, qui lit le journal d'écriture anticipée (write-ahead log) de la base de données et publie les changements dans Kafka. Aucune équipe n'a encore utilisé Debezium, si bien que la décision concerne l'équipe plateforme, qui exploite Kafka Connect, et l'équipe données, responsable de l'exploitation des bases de données.

1. Ébauche	L'équipe rédige l'ADR-0031 avec le statut Proposé et le présente au forum consultatif hebdomadaire
2. Solliciter des avis	Les équipes plateforme, données et sécurité commentent la pull request ; le forum suggère d'envisager la CDC managée du fournisseur de cloud
3. Décider	L'équipe Commandes accepte la plupart des avis, rejette l'option managée en raison du principe P3 (portabilité) et consigne pourquoi
4. Protéger	Une fonction d'aptitude sur le délai des événements (p99 inférieur à 2 s) et une alerte sur le retard du slot de réplication sont ajoutées
5. Radar	Debezium entre dans le radar en Trial, limité aux commandes ; les relais par interrogation passent en Hold
6. Revue	Au bout d'un trimestre, le groupe du radar examine l'expérience ; si elle est positive, Debezium passe en Adopt et la note de revue de l'ADR est close

L'entrée du radar est le court résumé public ; l'ADR contient le raisonnement et les avis. Chacun renvoie à l'autre.

L'anneau Trial est limité dans sa portée, ce qui garde la décision réversible pendant que l'organisation apprend.

Rien n'a attendu dans une file : la décision a pris environ une semaine, consacrée pour l'essentiel à recueillir des avis réellement utiles.

```
name,ring,quadrant,isNew,description
Debezium for outbox relay,Trial,Tools,TRUE,"Change data capture from the order database's
write-ahead log to Kafka, replacing polling relays. Trial in ordering only; see ADR-0031.
Owner: ordering team. Review: Q3."
Polling outbox relay,Hold,Techniques,FALSE,"Adds load and latency at peak; no new uses.
Existing relays move when Debezium leaves Trial."
```

Les entrées du radar au format CSV de Build Your Own Radar (name, ring, quadrant, isNew, description).

```
# ADR-0031: Use Debezium change data capture for the ordering outbox relay

Status: Accepted (2026-05-14)   Decision-taker: Priya N., ordering team
Context: The polling relay queries the outbox table every 500 ms. At 100 orders/s
it adds measurable database load and up to 1 s of event delay to dispatch.
Options: (1) tune polling; (2) Debezium CDC on the write-ahead log; (3) managed CDC
from the cloud provider.
Decision: Option 2, for the ordering service only, as a Trial on the radar.
Advice received:
- Platform team (2026-05-06): supports; will run Kafka Connect, needs runbook.
- Data team (2026-05-07): concern about replication slot growth if Connect stops;
  asks for an alert on slot lag. Accepted.
- Security (2026-05-08): CDC user must have replication rights only. Accepted.
- Advisory forum (2026-05-09): suggested option 3; rejected for portability (P3).
Consequences: lower load and delay; a new component to operate; slot-lag alert
and a fitness function on event delay (p99 under 2 s) added.
Review: after one quarter, or before any second service adopts it.
```

L'ADR-0031 dans un format compact à la Nygard, avec les avis consignés comme l'exige le processus de consultation.

Le rôle de l'architecture d'entreprise aujourd'hui

L'architecture d'entreprise (section 1.2) a eu une réputation difficile dans les organisations agiles : des équipes centrales produisant des cadres, des normes et des processus d'approbation que les équipes de livraison vivaient comme des obstacles. Le rôle évolue. Dans les organisations où il fonctionne bien, les architectes d'entreprise entretiennent les principes et le radar plutôt que de les dicter, tiennent à jour la carte des capacités métier et des systèmes qu'aucune équipe ne peut voir seule, prennent ou facilitent les décisions de type porte à sens unique qui concernent réellement toute l'organisation (identité, données centrales, stratégie cloud, grands fournisseurs), animent le forum consultatif, et façonnent la plateforme et la paved road avec les équipes plateforme.

L'image de Gregor Hohpe, celle de l'architecte qui prend l'ascenseur entre la salle du conseil et la salle des machines (section 11.7), décrit la compétence que cela exige : traduire la stratégie en orientation technique et les contraintes techniques en langage métier. Les praticiens restent en désaccord sur le degré d'autorité centrale qui est sain. Certains soutiennent qu'une fonction

centrale forte est indispensable dans les entreprises réglementées, marquées par de nombreuses fusions ou très intégrées ; d'autres, que l'essentiel de sa valeur peut être apporté par des plateformes, des principes et des avis. La réponse dépend de la taille de l'organisation, de sa réglementation et de la part de son paysage de systèmes qui est réellement partagée.

La conformité par l'automatisation

Quel que soit le modèle organisationnel, la conformité la moins coûteuse est automatisée. Les fonctions d'aptitude (section 11.3) vérifient les caractéristiques d'architecture. La politique en tant que code (policy as code) fait de même pour les règles de l'organisation : les règles sont écrites dans un langage lisible par une machine, conservées sous contrôle de version, revues et testées comme du code, et évaluées automatiquement par rapport aux définitions d'infrastructure, aux manifestes de déploiement ou aux requêtes d'API. Open Policy Agent, projet diplômé (graduated) de la Cloud Native Computing Foundation, est un moteur généraliste largement utilisé, doté de son propre langage de politiques, Rego ; Conftest applique des politiques Rego aux fichiers de configuration dans un pipeline, et des contrôleurs d'admission comme OPA Gatekeeper ou Kyverno font appliquer des politiques à l'intérieur des clusters Kubernetes.

L'automatisation de la conformité a un autre avantage que les organisations réglementées apprécient particulièrement : la preuve. Un journal de pipeline montrant que chaque mise en production a passé les mêmes vérifications constitue une preuve bien plus solide pour un auditeur qu'un formulaire signé, et il est produit sans coût supplémentaire.

```
package main

import rego.v1

# Guardrail: images must come from the organisation's scanned registry
deny contains msg if {
    input.kind == "Deployment"
    some c in input.spec.template.spec.containers
    not startswith(c.image, "registry.fooddelivery.example/")
    msg := sprintf("%s: image %s is not from the approved registry",
[input.metadata.name, c.image])
}

# Guardrail: every container declares a memory limit
deny contains msg if {
    input.kind == "Deployment"
    some c in input.spec.template.spec.containers
    not c.resources.limits.memory
    msg := sprintf("%s: container %s has no memory limit", [input.metadata.name, c.name])
}
```

Deux garde-fous sous forme de politiques Rego, vérifiés sur les manifestes de déploiement Kubernetes avec Conftest (conftest test deploy/) à chaque exécution du pipeline.

Équilibrer autonomie et alignement

La question pratique est de savoir quelles décisions sont gouvernées, et comment. Une règle utile découle de la section 1.6 : plus une décision est réversible et locale, plus elle doit être prise près de l'équipe ; plus elle est irréversible et largement ressentie, plus elle demande d'avis et, parfois, d'approbation. Le tableau présente une répartition typique ; chaque organisation ajuste la sienne.

Qui décide quoi

Type de décision	Qui décide	Mécanisme	Exemple
Locale et réversible	L'équipe	Jugement de l'équipe ; ADR si elle est significative	Une bibliothèque au sein d'un service ; la structure interne des modules
Touche d'autres équipes, réversible à un coût modéré	L'équipe, après consultation	Processus de consultation, forum consultatif, radar	Adopter Debezium ; un nouvel événement publié pour d'autres équipes
À l'échelle de l'organisation ou difficile à inverser	Un décisionnaire senior nommé, après une large consultation	Processus de consultation avec les architectes d'entreprise ; modification d'un principe ou du radar	Choix du fournisseur d'identité ; fournisseur de cloud principal
Règles objectives et vérifiables	Personne, au cas par cas	Garde-fous : politique en tant que code et fonctions d'aptitude	Registres d'images approuvés ; aucun bucket de stockage public
Réglementée ou à fort enjeu	Un approbateur responsable	Point de validation précoce, limité dans le temps, avec preuves	Traitement de données concernant la santé ; fonctions liées à la sûreté

La gouvernance doit elle-même être mesurée. Les signaux utiles sont le temps que prennent les décisions significatives entre la proposition et l'acceptation, le nombre d'exceptions aux normes demandées et accordées, la part des services qui suivent la paved road, le nombre de technologies en Hold qui continuent de se répandre, et le fait que des incidents remontent ou non à des décisions prises sans consultation. Si les décisions sont lentes, la gouvernance est trop lourde ; si des incidents continuent de remonter à des décisions non consultées, elle est trop légère.

GUIDE DE DÉCISION

Guide de décision : point de validation, garde-fou ou consultation ?

Utilisez-le pour décider comment une règle ou un type de décision donné doit être gouverné.

Q1 **La règle peut-elle être énoncée objectivement et vérifiée par une machine ?**

- **Oui** Faites-en un garde-fou : politique en tant que code ou fonction d'aptitude dans le pipeline, avec un responsable et une procédure d'exception.
- **Non** Continuez.

Q2 **La décision est-elle locale à une équipe et peu coûteuse à inverser ?**

- **Oui** Laissez l'équipe décider ; consignez un ADR si elle est significative.
- **Non** Continuez.

Q3 **La loi, la réglementation ou l'appétence au risque de l'organisation exigent-elles une validation par une personne responsable ?**

- **Oui** Utilisez un point de validation, mais tenez-le tôt, limitez-le dans le temps et rendez explicites les preuves dont il a besoin.
- **Non** Continuez.

Q4 **S'agit-il d'une porte à sens unique qui touche de nombreuses équipes ?**

- **Oui** Utilisez le processus de consultation en consultant largement, y compris les architectes d'entreprise, et avec un décisionnaire senior nommé.
- **Non** Utilisez le processus de consultation : consultez les personnes concernées et les experts, puis décidez et consignez.

ATTENTION

Erreurs de gouvernance courantes

- Des principes avec lesquels personne ne pourrait être en désaccord, et qui ne tranchent donc rien.
- Des normes sans procédure d'exception, que les équipes ignorent alors discrètement.
- Un forum consultatif qui se transforme peu à peu en comité d'approbation.
- Un radar tenu par une seule personne, qui consigne des opinions plutôt que l'expérience de l'organisation.
- Des points de validation tardifs dans la livraison, où les seules issues réalistes sont l'approbation ou une reprise coûteuse.
- Automatiser des règles dont personne n'est responsable, si bien que les faux positifs sont contournés au lieu d'être corrigés.

VÉRIFIEZ VOTRE COMPRÉHENSION

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

Un garde-fou maintient le travail dans des limites en continu, généralement au moyen de vérifications automatisées, sans l'arrêter ; un point de validation arrête le travail jusqu'à ce que quelqu'un l'approuve.

▼ Quelles sont les quatre parties d'un principe d'architecture à la manière de TOGAF ?

Nom, énoncé, justification et implications.

▼ Dans le processus de consultation, qui prend la décision, et cette personne doit-elle suivre les avis ?

La personne ou l'équipe qui prend la décision, qui doit solliciter et consigner les avis des personnes concernées et des experts, mais n'est pas tenue de les suivre.

▼ Que signifie généralement, pour les usages existants, le placement d'une technologie dans l'anneau Hold d'un radar ?

Cela signifie qu'aucun nouvel usage ne doit commencer ; les usages existants ne sont pas interdits, même s'ils peuvent être migrés au fil du temps.

▼ **À vous : Toute nouvelle base de données doit être approuvée par un comité d'architecture mensuel, et les files d'attente atteignent six semaines. Les équipes évitent désormais de demander. La sécurité doit rester assurée. Comment reconcevriez-vous cette gouvernance à l'aide de garde-fous (guardrails) et du processus de consultation (advice process) ?**

Remplacez le point de validation par des garde-fous là où les règles sont objectives : la politique sous forme de code peut vérifier automatiquement le chiffrement, les sauvegardes et les moteurs approuvés, et fournir des preuves d'audit. Publiez un radar indiquant quelles bases de données sont Adopt, Trial ou Hold. Pour tout ce qui sort de l'ordinaire, utilisez le processus de consultation : l'équipe décide après avoir consulté les personnes concernées et les experts, et consigne un ADR. Le forum donne de la visibilité mais n'approuve pas. Ne gardez un point de validation précoce, s'il en faut un, que pour les changements réellement à haut risque.

SOURCES

- Andrew Harmel-Law: Scaling the Practice of Architecture, Conversationally (martinfowler.com, 2021): https://martinfowler.com/articles/scaling-architecture-conversationally.html
- Thoughtworks Technology Radar: FAQ: https://www.thoughtworks.com/radar/faq
- Thoughtworks: Build Your Own Radar: https://www.thoughtworks.com/radar/byor
- Open Policy Agent documentation: https://www.openpolicyagent.org/docs/latest/

SECTION 11.5 SUR 11.8 · AVANCÉ · 26 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Team Topologies et conception sociotechnique

AVANT DE LIRE

Cette section s'appuie sur :

- 3.7 Loi de Conway
- 5.7 Conception pilotée par le domaine
- 10.7 Ingénierie de plateforme

Termes déjà vus qu'elle utilise : Loi de Conway, Contexte délimité (bounded context), Plateforme interne de développement (IDP)

L'IDÉE EN BREF

Pensez à un groupe de restaurants. La cuisine de chaque restaurant sert ses propres clients du début à la fin. Un fournisseur central livre les ingrédients et l'équipement à toutes les cuisines à partir d'un catalogue simple, afin que les cuisiniers n'aient pas à exploiter une ferme. Un atelier de pâtisserie spécialisé fabrique le seul produit trop difficile à maîtriser pour chaque cuisine. Et un chef invité passe quelques semaines à enseigner une nouvelle technique à une cuisine, puis repart. Team Topologies donne aux organisations du logiciel les mêmes quatre types d'équipes et trois manières de travailler ensemble, afin que chaque équipe soit responsable d'un travail qu'elle peut réellement garder en tête.

CE QU'IL FAUT SAVOIR

- L'architecture sociotechnique conçoit ensemble la structure du logiciel et celle des équipes, car la loi de Conway fait que chacune façonne l'autre.
- Team Topologies (Skelton et Pais, 2019 ; deuxième édition 2025) définit quatre types d'équipes : orientée flux (stream-aligned, le cas par défaut), plateforme, facilitatrice (enabling) et sous-système compliqué.
- Les équipes interagissent selon trois modes délibérés : la collaboration (découverte limitée dans le temps), X en tant que service, ou X-as-a-Service (consommer une interface claire), et la facilitation (aider une autre équipe à acquérir une capacité).
- La charge cognitive de l'équipe, adaptée de la théorie de la charge cognitive de John Sweller (1988), limite ce dont une équipe peut être responsable ; réduisez la charge extrinsèque grâce aux plateformes et à l'automatisation, et bornez la charge intrinsèque par des frontières bien placées.

- L'API d'une équipe (team API) documente ce que l'équipe offre et attend, à la fois ses interfaces techniques et ses manières de travailler ; la plateforme viable la plus fine (thinnest viable platform) est la plus petite plateforme qui accélère réellement ses utilisateurs.
- Les plans de fracture, comme les contextes délimités, le périmètre réglementaire, le rythme des changements, le risque et l'isolation des performances, montrent où découper le logiciel et les équipes ; la technologie seule est généralement la ligne de découpe la plus faible.
- Réorganiser les équipes ne restructure pas un code couplé : les changements d'équipes et les changements d'architecture doivent donc être planifiés et évoluer ensemble.

La conception des équipes est un travail d'architecture

Le chapitre 3 a montré que la loi de Conway n'est pas une curiosité mais une force : le système qu'un groupe d'équipes construit tend à reproduire la manière dont ces équipes communiquent. Si c'est vrai, alors décider comment les équipes sont formées, de quoi chacune est responsable et comment elles interagissent est une décision architecturale, aux conséquences aussi durables que le choix d'une base de données ou d'un style d'intégration. La discipline qui consiste à concevoir ensemble la structure du logiciel et celle des équipes s'appelle l'architecture sociotechnique.

Le cadre pratique le plus influent pour y parvenir est Team Topologies, de Matthew Skelton et Manuel Pais (IT Revolution, 2019 ; deuxième édition 2025). Il part d'un objectif simple, le *flux rapide* (fast flow) : un courant régulier et soutenable de petits changements, de l'idée à la production, avec peu d'attente entre les équipes. Il propose ensuite un petit vocabulaire pour concevoir des organisations qui l'atteignent : quatre types d'équipes, trois modes d'interaction, une limite appelée charge cognitive de l'équipe, et une technique pour trouver les lignes de découpe le long desquelles le logiciel et les équipes peuvent être divisés.

Plusieurs prémisses traversent l'approche. L'équipe, et non l'individu, est l'unité de livraison : le travail est confié à des équipes, la responsabilité appartient aux équipes, et les équipes sont durables, afin d'accumuler la connaissance de leur domaine et de leur code au lieu d'être assemblées pour un projet puis dispersées. Les équipes sont petites, généralement de cinq à neuf personnes, car la confiance et la compréhension partagée déclinent vite au-delà de cette taille (les auteurs s'appuient sur les recherches de Robin Dunbar sur la taille des groupes). Et les frontières des équipes doivent être conçues pour le flux des changements, pas pour la commodité des lignes hiérarchiques ni d'après l'historique de qui a construit quoi.

Pour les architectes, cela change le métier. Une frontière de service proposée n'est bonne que si une équipe peut en être entièrement responsable ; une équipe proposée n'est viable que si le code dont elle est responsable lui permet de travailler sans demander la permission à trois autres équipes. L'architecte qui dessine des boîtes sans se demander qui sera responsable de chacune ne conçoit que la moitié du système.

> **SI VOUS DÉBUTEZ**
>
> ### Si vous débutez
>
> Une *chaîne de valeur* (value stream) est la suite de travaux qui transforme un besoin en quelque chose dont un client ou un utilisateur tire profit, par exemple tout ce qu'il faut pour permettre aux clients de programmer une commande pour plus tard. *Pluridisciplinaire* (cross-functional) signifie que l'équipe réunit toutes les compétences dont ce travail a besoin (conception, frontend, backend, tests, exploitation) au lieu de confier le travail à d'autres équipes. La section 3.7 de ce cours explique la loi de Conway et la correspondance entre organisation et système sur laquelle s'appuie cette section.

Les quatre types d'équipes fondamentaux

Team Topologies soutient que presque toutes les équipes d'une organisation logicielle devraient relever de l'un de quatre types. L'intérêt d'avoir si peu de types est la clarté : chaque type a un objectif différent, une manière différente de mesurer la réussite et une relation différente avec les autres équipes, si bien que connaître le type d'une équipe indique à tous ce qu'ils peuvent en attendre.

Une équipe orientée flux (stream-aligned team) est alignée sur un flux de changements : un produit, un parcours client, un groupe d'utilisateurs ou une capacité métier. Elle est responsable de sa partie de bout en bout, de la découverte à l'exploitation en production en passant par la construction, et doit pouvoir livrer la plupart des changements sans attendre personne. Les équipes orientées flux sont le cas par défaut et doivent constituer la grande majorité ; les trois autres types existent pour alléger leur charge.

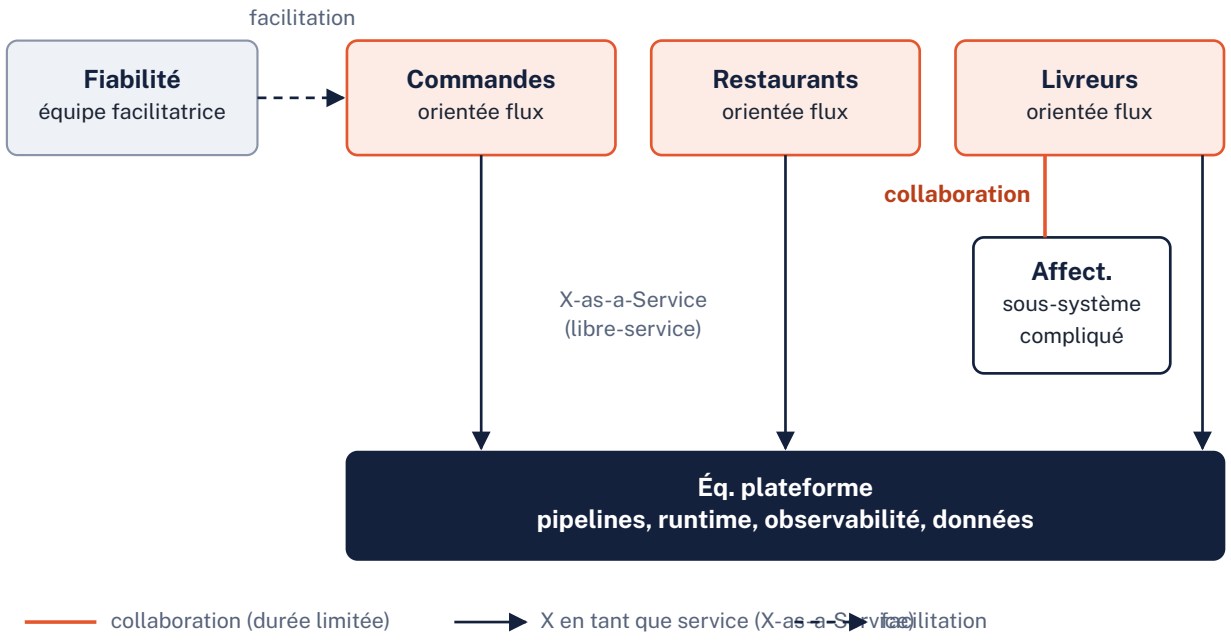
Une équipe plateforme fournit des services internes (pipelines de déploiement, environnement d'exécution, observabilité, infrastructure de données) sous la forme d'un produit que les équipes orientées flux consomment en libre-service. Une bonne plateforme se juge à la vitesse de ses utilisateurs, pas à sa propre liste de fonctionnalités. Une équipe facilitatrice (enabling team) se compose de spécialistes qui aident d'autres équipes à acquérir une capacité qui leur manque, comme l'automatisation des tests, les pratiques d'observabilité ou un nouveau framework, puis passent à autre chose ; sa réussite se mesure au fait que l'autre équipe n'a plus besoin d'elle. Une équipe sous-système compliqué est responsable d'une partie qui exige

une expertise rare, comme un moteur de tarification reposant sur des modèles actuariels, un solveur d'optimisation d'itinéraires ou un codec vidéo, afin que les équipes orientées flux puissent l'utiliser sans avoir à la maîtriser. Le livre insiste sur le fait que ce type doit être utilisé avec parcimonie, et seulement là où la spécialité est réelle.

Comparaison des quatre types d’équipes

Type d’équipe	Objectif	Responsable de	Signes de bonne santé	Exemple de la livraison de repas
Orientée flux	Livrer un flux continu de changements pour une partie de l’activité	Fonctionnalités, code, données et exploitation de son flux, de bout en bout	Met en production de façon indépendante ; la plupart des changements ne nécessitent aucune autre équipe ; la charge d’astreinte est supportable	Équipe Commandes clients : cartes présentées aux clients, panier, paiement, suivi des commandes
Plateforme	Réduire la charge cognitive des équipes orientées flux grâce à des produits internes en libre-service	Les services internes de la plateforme, leur documentation et leur fiabilité	Les équipes l’adoptent volontairement ; intégrer un nouveau service prend des heures, pas des semaines	Équipe plateforme de livraison logicielle : pipelines, environnement d’exécution des conteneurs, journalisation et métriques, courtier d’événements
Facilitatrice	Combler les lacunes de capacité d’autres équipes, puis partir	Savoir-faire, guides, exemples ; pas le code de production d’autres équipes	Les interventions sont limitées dans le temps et se terminent quand l’autre équipe est autonome	Équipe facilitatrice fiabilité, qui accompagne les équipes sur les objectifs de niveau de service et la gestion des incidents
Sous-système compliqué	Encapsuler une spécialité trop pointue pour chaque équipe orientée flux	Un composant spécialisé derrière une interface claire	Les consommateurs l’utilisent comme un service sans avoir besoin des spécialistes dans la pièce	Équipe d’optimisation de l’affectation : l’algorithme qui affecte les livreurs aux commandes

Types d'équipes et modes d'interaction



La plupart des équipes sont orientées flux ; une plateforme les sert en tant que service, une équipe facilitatrice les accompagne (facilitation), et une équipe sous-système compliqué travaille étroitement avec un flux pendant une durée limitée.

Trois modes d'interaction

Les types d'équipes disent ce que sont les équipes ; les modes d'interaction disent comment elles travaillent ensemble. Skelton et Pais soutiennent que l'interaction ambiguë et improvisée entre équipes est une source majeure de retard, car chaque conversation non planifiée est une dépendance. Ils recommandent donc que chaque paire d'équipes qui doivent interagir choisisse délibérément l'un des trois modes, pour une période déterminée.

En **collaboration**, deux équipes travaillent étroitement ensemble vers un objectif commun, avec des responsabilités floues, par exemple pendant qu'elles découvrent comment une nouvelle capacité doit être découpée. Ce mode produit un apprentissage rapide mais coûte aux deux équipes de la capacité et de la charge cognitive : il doit donc être limité dans le temps. En X en tant que service (X-as-a-Service), une équipe fournit quelque chose (une API, une capacité de plateforme, une bibliothèque) qu'une autre consomme avec peu ou pas de collaboration ; ce mode passe bien à l'échelle, mais ne fonctionne qu'une fois la frontière bien comprise et l'interface de bonne qualité. En **facilitation**, une équipe, généralement une équipe facilitatrice, en aide une autre à apprendre ou à lever un obstacle.

Les modes sont faits pour changer. Une séquence courante consiste à découvrir puis à stabiliser : une équipe plateforme collabore avec une équipe orientée flux pendant quelques semaines pour concevoir une nouvelle capacité, puis la transforme en offre en libre-service et passe en X en tant que service. Les modes servent aussi de mécanisme de détection. Si une équipe censée consommer un service en tant que service a sans cesse besoin de réunions, l'interface ou la frontière est probablement mauvaise ; si deux équipes collaborent depuis un an, elles ne forment peut-être en réalité qu'une seule équipe, ou la séparation entre elles est peut-être mal placée.

Les trois modes d'interaction

Mode	À quoi il ressemble	À utiliser quand	Coût ou signal d'alerte
Collaboration	Deux équipes travaillent comme une seule sur un problème commun pendant un temps ; séances de conception communes, éléments de backlog partagés	Découvrir une nouvelle frontière, technologie ou capacité dont personne ne connaît encore la bonne interface	Charge cognitive élevée pour les deux équipes ; si cela dure indéfiniment, la frontière est probablement mauvaise
X en tant que service (X-as-a-Service)	Une équipe consomme une API, une plateforme ou un composant qu'une autre fournit, au moyen de la documentation et du libre-service	La frontière et l'interface sont bien comprises et stables	Lent à s'adapter si l'interface est médiocre ; des demandes d'assistance constantes montrent que ce n'est pas encore un vrai service
Facilitation	Coaching, travail en binôme, exemples et ateliers courts qui aident une équipe à acquérir une capacité	Une équipe manque d'une compétence ou est bloquée par un obstacle qu'elle peut apprendre à lever	Dérive vers le fait de faire le travail de l'autre équipe, ce qui crée de la dépendance au lieu de la capacité

La charge cognitive comme contrainte de conception

L'idée de charge cognitive vient de la psychologie de l'éducation. La théorie de la charge cognitive de John Sweller, publiée pour la première fois en 1988, partait de l'observation que la mémoire de travail humaine ne peut retenir et traiter que quelques éléments nouveaux à la fois, si bien que l'enseignement doit être conçu pour ne pas la surcharger. La théorie distingue la charge intrinsèque (la difficulté inhérente à la matière), la charge extrinsèque (la difficulté causée par la manière dont la tâche est présentée) et, dans des travaux ultérieurs, la charge

pertinente (l'effort qui construit une compréhension durable). Les chercheurs débattent encore de la question de savoir si la charge pertinente est une catégorie distincte, et Team Topologies utilise les trois comme une heuristique pratique plutôt que comme une mesure.

La section 3.7 a présenté la version de cette idée appliquée aux équipes : le total qu'une équipe doit avoir en tête pour construire, exploiter et faire évoluer ce dont elle est responsable. Ce qui compte ici, c'est la manière de l'utiliser comme contrainte de conception. Quand une architecture confie à une équipe plus qu'elle ne peut comprendre, les effets sont prévisibles : une livraison plus lente, des corrections superficielles, des astreintes fragiles, des connaissances concentrées chez une ou deux personnes et, à terme, des départs. Ce sont les mêmes symptômes que l'on impute souvent à de mauvais ingénieurs ou à du mauvais code.

Le livre propose des règles empiriques pour dimensionner les responsabilités d'une équipe selon la complexité du domaine : confier chaque domaine à une seule équipe ; une équipe peut généralement gérer deux ou trois domaines simples ; une équipe responsable d'un domaine complexe ne devrait être responsable de rien d'autre ; et aucune équipe ne devrait avoir deux domaines compliqués. On peut aussi interroger directement les équipes, par exemple avec un court questionnaire sur leur degré de compréhension de leurs systèmes, leur confiance pendant les astreintes et le temps consacré à l'outillage plutôt qu'au domaine. Des files de tickets qui s'allongent, de longs délais d'intégration des nouveaux arrivants et des transferts fréquents vers la seule personne qui comprend un composant sont des signaux indirects utiles.

Les réponses architecturales reflètent les trois types de charge. Réduisez la charge extrinsèque avec des plateformes, de l'automatisation, des chemins balisés (paved paths) et une bonne documentation. Bornez la charge intrinsèque en traçant les frontières des services et des modules de sorte que la part du domaine de chaque équipe soit cohérente et d'une taille gérable, et en plaçant les spécialités pointues derrière des interfaces de sous-systèmes compliqués. Protégez la charge pertinente en laissant du temps pour l'apprentissage ; c'est là que les équipes facilitatrices aident.

Les API d'équipe et la plateforme viable la plus fine

Si les équipes interagissent selon des modes délibérés, chaque équipe a besoin d'un visage clair face au reste de l'organisation. Team Topologies l'appelle l'API de l'équipe (team API) : tout ce que l'équipe offre et attend. Elle comprend l'API technique (endpoints, événements, bibliothèques, avec leurs règles de versionnage et de dépréciation), mais aussi l'interface humaine : la documentation et les guides de prise en main, la feuille de route et les priorités actuelles de l'équipe, ses pratiques de travail, les canaux par lesquels on peut la joindre, la manière de lui demander du travail et les délais de réponse à attendre.

Mettre par écrit l'API d'une équipe est un exercice peu coûteux et révélateur. Il oblige l'équipe à dire de quoi elle est responsable, ce qui met généralement au jour des services sans responsable et des données disputées, et il donne aux autres équipes un moyen d'interagir sans réunion. Beaucoup d'organisations la conservent sous la forme d'une courte page dans le dépôt de l'équipe, revue chaque trimestre avec la documentation d'architecture.

La plateforme mérite une attention particulière, car c'est là que les bonnes intentions tournent le plus souvent mal. Une plateforme construite comme un grand projet interne, avec une longue feuille de route et une adoption obligatoire, peut facilement ajouter de la charge cognitive au lieu d'en retirer. Skelton et Pais recommandent la plateforme viable la plus fine (thinnest viable platform) : le plus petit ensemble de services, de documentation et de normes qui accélère réellement ses utilisateurs. Pour une petite organisation, il peut s'agir d'une page bien tenue qui énumère les services du fournisseur cloud que les équipes doivent utiliser, et comment ; pour une grande, ce peut être une plateforme interne de développement. L'ingénierie de plateforme est traitée en détail au chapitre 10 ; l'essentiel ici est qu'une plateforme est un produit dont les clients sont d'autres équipes, et qu'elle doit être conçue et mesurée comme tel.

Plans de fracture : où découper le logiciel et les équipes

Quand une partie du système est devenue trop grande pour une seule équipe, il faut la découper, et le découpage doit fonctionner à la fois pour le code et pour les équipes. Team Topologies appelle plans de fracture les lignes de découpe naturelles le long desquelles le logiciel peut être divisé. Le livre en énumère plusieurs, et le premier est généralement le meilleur : un contexte délimité issu de la conception pilotée par le domaine, car une frontière qui suit le langage et les règles du métier tend à garder ensemble les changements liés.

Les autres sont utiles quand ils correspondent à une vraie différence dans la manière dont les parties du système doivent être traitées. Découper selon une frontière technologique seule (par exemple une équipe mobile et une équipe backend) est généralement le choix le plus faible, car la plupart des changements la traversent alors : c'est la structure d'équipes en couches contre laquelle met en garde la loi de Conway.

Plans de fracture, avec des exemples tirés de la livraison de repas

Plan de fracture	Découper quand	Exemple
Domaine métier (contexte délimité)	Des parties du domaine ont leur propre langage, leurs propres règles et leurs propres experts	Commandes, intégration des restaurants et gestion des livreurs en tant que contextes distincts
Conformité réglementaire	Une partie est soumise à des règles qui ne doivent pas peser sur le reste	Traitement des paiements par carte maintenu dans un périmètre restreint et audité pour PCI DSS
Rythme des changements	Certaines parties changent chaque jour, d'autres chaque année	Les promotions et les campagnes changent chaque semaine ; les règles de facturation changent rarement
Localisation des équipes	Les équipes travaillent dans des fuseaux horaires qui se chevauchent peu	Une équipe des opérations livreurs, dans un autre pays, est responsable de l'intégration des livreurs
Risque	Certaines parties tolèrent les expérimentations, d'autres non	Expérimentations de recommandation séparées de l'acceptation des commandes
Isolation des performances	Une partie a des besoins de charge ou de latence très différents	Mises à jour en direct de la position des livreurs séparées de la consultation des cartes
Technologie	Une partie a besoin d'une pile technologique réellement différente	Une ancienne intégration des tablettes des restaurants qui ne fonctionne que sur du matériel ancien
Personas utilisateurs	Différents groupes d'utilisateurs ont des besoins et des parcours différents	Les clients, le personnel des restaurants et les livreurs ont chacun leur app et leur équipe

La loi de Conway, la manœuvre inversée et ses limites

Team Topologies est, en pratique, une boîte à outils pour la manœuvre de Conway inversée décrite à la section 3.7 : décider de l'architecture voulue, puis façonner les équipes et les modes d'interaction pour que la loi de Conway pousse le système vers elle. Ses apports sont pratiques. Les types d'équipes disent quel genre d'équipe doit être responsable de chaque partie ; les modes d'interaction disent quelles frontières entre équipes sont censées être minces (X en tant que service) et lesquelles sont temporairement épaisses (collaboration) ; la charge cognitive dit de combien chaque équipe peut être responsable ; les plans de fracture disent où couper.

Les limites vues à la section 3.7 s'appliquent toujours et méritent d'être rappelées dans ce contexte. Réorganiser les équipes ne restructure pas une base de code fortement couplée : pendant des mois après une réorganisation, les nouvelles équipes se battront contre l'ancienne architecture, et c'est pourquoi les changements d'équipes et les changements d'architecture doivent être planifiés comme un seul programme. Des frontières figées trop tôt dans des équipes rendent les erreurs coûteuses : beaucoup d'organisations gardent donc les frontières incertaines à l'intérieur d'une seule équipe ou d'un seul monolithe modulaire jusqu'à disposer de preuves. Et des réorganisations fréquentes détruisent la connaissance des équipes durables dont dépend l'approche.

Les praticiens ne s'accordent pas non plus sur la mesure dans laquelle il faut appliquer le cadre à la lettre. Ses partisans apprécient le vocabulaire commun, qui transforme des plaintes vagues (nous attendons sans cesse l'équipe data) en questions sur lesquelles on peut travailler en conception (quel mode d'interaction voulons-nous avec l'équipe data, et qu'est-ce qui en ferait un X en tant que service ?). Ses détracteurs avertissent que les quatre types sont souvent appliqués comme des étiquettes à une organisation inchangée, que les vraies équipes mélangent parfois les types (une équipe plateforme qui est aussi responsable d'un sous-système compliqué) et que le cadre dit peu de choses du financement, de la structure managériale ou des incitations. Les deux points sont justes : les types sont une aide à la conception, pas un organigramme, et les auteurs eux-mêmes les présentent comme un point de départ pour un ajustement continu.

D'OÙ CELA VIENT

D'où vient la pensée sociotechnique

La notion de système sociotechnique vient du Tavistock Institute de Londres. En 1951, Eric Trist et Ken Bamforth ont publié une étude sur les mines de charbon britanniques montrant qu'une nouvelle méthode d'extraction, plus mécanisée, avait fait baisser la productivité parce qu'elle démantelait de petits groupes de travail auto-organisés. Leur conclusion, selon laquelle les systèmes techniques et sociaux doivent être conçus ensemble, est la même intuition que celle à laquelle Conway est parvenu pour le logiciel en 1968, et que Team Topologies applique aujourd'hui.

EXEMPLE COMMENTÉ

Exemple commenté : réorganiser les équipes de la livraison de repas autour des flux

La plateforme de livraison de repas compte désormais 38 ingénieurs répartis en cinq équipes organisées par couche technique : apps mobiles, frontend web, API backend, base de données et exploitation. Le délai entre une idée et sa mise en production est d'environ six semaines, les mises en production sont coordonnées toutes les deux semaines entre toutes les équipes, et les trois derniers incidents ont chacun duré plusieurs heures parce que personne n'était responsable d'un parcours client complet. L'architecte et les responsables de l'ingénierie repensent l'organisation en cinq étapes.

1. Cartographier les flux	Un event storming et un examen des 50 dernières fonctionnalités font apparaître trois grands flux de changements : les commandes des clients (cartes, panier, paiement, suivi), les restaurants partenaires (intégration, cartes, horaires d'ouverture, versements) et la livraison (intégration des livreurs, affectation, suivi en direct). 44 des 50 fonctionnalités relevaient principalement de l'un d'eux.
2. Vérifier les plans de fracture du logiciel	Commandes, restaurants et livraison sont déjà des contextes délimités distincts, avec leurs propres services et bases de données (section 11.1) : le code peut donc suivre. L'affectation des livreurs utilise un solveur d'optimisation que seuls trois ingénieurs comprennent : un candidat au sous-système compliqué. Les paiements par carte restent chez Commandes, mais dans un petit module audité séparément (plan de fracture réglementaire).
3. Dimensionner les équipes selon la charge cognitive	Trois équipes orientées flux de 7 personnes (chacune avec des compétences mobile, web, backend, tests et exploitation). Une équipe sous-système compliqué de 5 personnes pour l'optimisation de l'affectation (les trois spécialistes plus deux ingénieurs qui apprennent le domaine). Une équipe plateforme de 9 personnes, responsable des pipelines, de l'environnement d'exécution des conteneurs, de l'observabilité et du courtier d'événements. Une petite équipe facilitatrice de 3 personnes pour les pratiques de fiabilité. Total : 38.
4. Choisir les modes d'interaction	Les équipes orientées flux consomment la plateforme en X en tant que service dès le premier jour, en utilisant ses pipelines existants. Livraison et Affectation collaborent pendant un trimestre pour repenser l'interface d'affectation, puis passent en X en tant que service. L'équipe facilitatrice accompagne Commandes pendant huit semaines (objectifs de niveau de service, runbooks d'astreinte), puis passe à Restaurants.

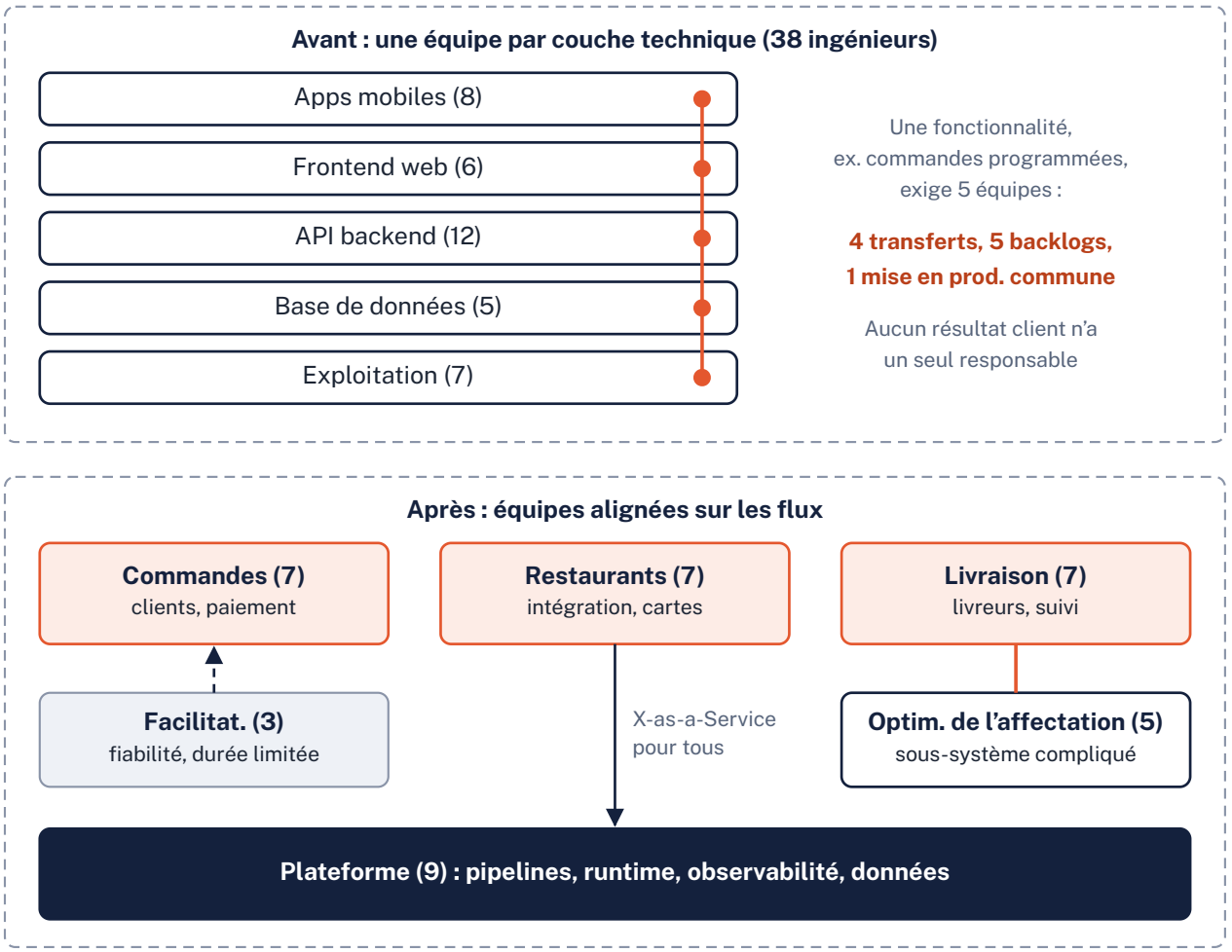
5. Publier les API d'équipe et mesurer	Chaque équipe rédige une API d'équipe d'une page, chaque service et chaque table de base de données reçoit exactement une équipe responsable, et les responsables suivent chaque trimestre le délai de mise en œuvre, la fréquence de déploiement, le nombre de tickets inter-équipes par fonctionnalité et un questionnaire sur la confiance pendant les astreintes.
Résultat attendu	Au bout de deux trimestres, la plupart des fonctionnalités devraient être livrées par une seule équipe sans mise en production coordonnée ; un travail inter-équipes persistant signalerait une frontière à revoir.

La réorganisation part des flux de changements et des frontières existantes du logiciel, pas des lignes hiérarchiques actuelles.

Les spécialistes ne sont regroupés que là où le domaine l'exige (l'affectation), et la plateforme existe pour alléger la charge des équipes orientées flux, pas pour les contrôler.

Les modes d'interaction ont des dates de fin : on attend donc de l'organisation qu'elle évolue plutôt qu'elle ne se fige dans un nouvel organigramme.

Avant et après



Les équipes par couche font passer chaque fonctionnalité par toutes les équipes ; des équipes orientées flux, soutenues par une plateforme, une équipe facilitatrice et une équipe spécialisée, peuvent livrer seules la plupart des fonctionnalités.

ATTENTION

Erreurs courantes

- Renommer les équipes existantes avec les étiquettes de Team Topologies alors que la responsabilité, le financement et les interactions restent les mêmes.
- Une équipe plateforme qui fonctionne par tickets et approbations, si bien que les équipes orientées flux l'attendent : c'est un goulot d'étranglement, pas une plateforme.
- Une équipe facilitatrice qui finit par faire en permanence le travail des autres équipes, ce qui crée de la dépendance au lieu de la capacité.
- Une collaboration sans date de fin, qui fusionne discrètement la charge cognitive de deux équipes.
- Découper les équipes selon des lignes technologiques (frontend, backend, base de données) et s'attendre à un flux rapide.
- Tracer d'abord les frontières des services et désigner les responsables plus tard, ce qui laisse des services orphelins ou à responsabilité partagée.

Questions fréquentes

Les équipes composant sont-elles toujours une erreur ?
Pas toujours. Une équipe responsable d'un composant partagé peut être le bon choix quand le composant exige une spécialité pointue (un sous-système compliqué) ou qu'il est proposé comme un véritable produit en libre-service (une partie d'une plateforme). Cela devient un problème quand la plupart des fonctionnalités exigent des modifications du composant et que les équipes orientées flux doivent faire la queue pour les obtenir.

Où se situent les architectes dans ce modèle ?
Généralement pas dans une équipe séparée qui contrôle les passages. Les architectes agissent souvent comme une équipe facilitatrice, en aidant les équipes orientées flux à prendre de bonnes décisions et en gardant visibles les qualités à l'échelle du système, et ils participent à la conception de la structure des équipes elle-même. La section 11.4 explique comment la gouvernance peut fonctionner sans points de validation.

À quelle fréquence les frontières des équipes devraient-elles changer ?
Rarement et délibérément. On s'attend à ce que les modes d'interaction changent tous les quelques mois ; la composition des équipes et leurs responsabilités doivent rester stables bien plus longtemps, car c'est dans les équipes durables que s'accumule la connaissance du domaine et du système.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Citez trois choses que contient l'API d'une équipe en plus de ses interfaces de code.

Par exemple sa documentation et ses guides de prise en main, sa feuille de route et ses priorités, ses pratiques de travail, les canaux par lesquels on peut la joindre et la manière de lui demander du travail.

▼ Pourquoi s'attend-on à ce que les modes d'interaction changent avec le temps ?

Parce que la collaboration convient à la découverte d'une nouvelle frontière et qu'une fois la frontière et l'interface comprises, les équipes doivent se stabiliser en X en tant que service ; les modes suivent le degré de maturité de la frontière.

▼ Qu'est-ce que la plateforme viable la plus fine, et pourquoi la viser ?

Le plus petit ensemble de services, de documentation et de normes qui accélère réellement les équipes qui l'utilisent ; une plateforme plus grande risque d'ajouter de la charge cognitive au lieu d'en retirer.

▼ Pourquoi une séparation entre équipe frontend et équipe backend est-elle généralement un plan de fracture faible ?

Parce que la plupart des fonctionnalités la traversent : chaque changement exige alors les deux équipes, des transferts et des mises en production coordonnées, c'est-à-dire la structure en couches que la loi de Conway transforme en un système en couches et couplé.

▼ À vous : Dans une société de billetterie, attribuez un type d'équipe et un mode d'interaction à : une équipe chargée du parcours d'achat ; un petit groupe de spécialistes des algorithmes d'attribution des places ; un groupe qui fournit les outils de CI et de déploiement ; deux experts qui aident les équipes à apprendre à tester.

Parcours d'achat : orientée flux, le choix par défaut. Spécialistes de l'attribution des places : sous-système compliqué, offrant X en tant que service à l'équipe du parcours, avec une collaboration limitée dans le temps sur les nouvelles interfaces. Groupe d'outillage : plateforme, offrant X en tant que service et restant la plateforme viable la plus fine. Experts en tests : facilitatrice, en mode facilitation, avec pour but de partir une fois la compétence transférée. Vérifiez que la charge cognitive de chaque équipe reste gérable, et rappelez-vous que renommer les équipes ne change pas à soi seul un code couplé.

SOURCES

- Team Topologies: key concepts: https://teamtopologies.com/key-concepts
- Martin Fowler: Team Topologies (2023): https://martinfowler.com/bliki/TeamTopologies.html

- Melvin Conway: How Do Committees Invent? (1968): https://melconway.com/Home/Committees_Paper.html

SECTION 11.6 SUR 11.8 · AVANCÉ · 29 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Moderniser les systèmes hérités : strangler fig et migration incrémentale

AVANT DE LIRE

Cette section s'appuie sur :

- 1.7 Dette technique
- 5.7 Conception pilotée par le domaine
- 7.1 Propriété des données

Termes déjà vus qu'elle utilise : Dette technique, Système de référence (system of record), API gateway (passerelle d'API)

L'IDÉE EN BREF

Imaginez une ville qui a besoin d'un nouveau pont. Démolir l'ancien pour en construire un nouveau couperait la ville du monde pendant des années. À la place, les ingénieurs construisent le nouveau pont à côté de l'ancien, l'ouvrent une voie à la fois, y font passer la circulation progressivement et ne démolissent l'ancien pont que lorsque plus rien ne l'emprunte. Si une nouvelle voie pose problème, la circulation revient simplement sur l'ancien pont pendant un moment. Moderniser un vieux logiciel fonctionne mieux de la même manière : on le remplace morceau par morceau pendant qu'il continue de fonctionner, afin que les clients ne perdent jamais le service dont ils dépendent.

CE QU'IL FAUT SAVOIR

- Un système hérité apporte encore de la valeur, mais il est difficile, risqué ou coûteux à modifier ; modernisez pour des raisons métier, comme des changements bloqués, la fin du support, des compétences rares ou le coût, pas par effet de mode.
- Les réécritures big bang échouent parce que la valeur arrive tard, que la cible bouge, que des comportements cachés se perdent et que la migration des données et la bascule viennent en dernier ; l'essai de Spolsky de 2000 et la migration de TSB en 2018 sont des mises en garde classiques.
- Les sept R (retirer, conserver, réhéberger, relocaliser, racheter, changer de plateforme, refactoriser ou réarchitecturer) donnent à chaque application d'un portefeuille une stratégie délibérée ; reconstruire n'est qu'une option parmi d'autres.

- Le strangler fig (Fowler, 2004) fait passer le trafic par une façade et déplace le comportement tranche par tranche vers de nouveaux composants jusqu'à ce que le système hérité puisse être éteint ; branch by abstraction fait de même à l'intérieur d'une seule base de code.
- Les exécutions en parallèle comparent en production les résultats de l'ancien et du nouveau, tandis que l'ancien résultat continue de faire foi ; le dark launching sollicite les nouveaux chemins sous une charge réelle sans que les utilisateurs les voient.
- Les données ont besoin d'un seul système de référence à chaque étape ; la capture des données modifiées, l'interception d'événements et les couches anticorruption maintiennent les systèmes synchronisés, et l'historique est migré par cohortes, avec répétitions préalables et rapprochements.
- L'architecture de transition est un coût délibéré et temporaire ; une migration n'est terminée que lorsque l'ancien système et le code de transition ont été supprimés.

Ce qui fait d'un système un système hérité

Un système hérité (legacy) n'est pas simplement un vieux système. C'est un système qui fait encore quelque chose de précieux, souvent quelque chose sans quoi l'organisation ne pourrait pas fonctionner, mais qui est devenu difficile, risqué ou coûteux à modifier. Michael Feathers, dans *Working Effectively with Legacy Code* (2004), en a donné une définition volontairement provocatrice : le code hérité est du code sans tests, car sans tests chaque changement est un pari. Les autres causes courantes sont des compétences rares (un langage ou une plateforme que peu de gens connaissent encore), une technologie qui n'est plus supportée (un système d'exploitation ou une base de données arrivé en fin de support chez son fournisseur), une connaissance perdue (les personnes qui comprenaient la conception sont parties) et une dette technique accumulée dans la structure elle-même, le type coûteux décrit à la section 1.7.

Il faut le dire clairement : la plupart des systèmes hérités sont des systèmes qui ont réussi. Ils ont survécu parce qu'ils fonctionnaient et rapportaient de l'argent. Ce succès explique aussi pourquoi ils sont difficiles à remplacer : des années d'utilisation réelle y ont incorporé des milliers de petites décisions, de corrections et de règles métier qui ne sont écrites nulle part, sauf dans le code et dans les habitudes des personnes qui l'utilisent.

La modernisation a donc besoin d'une raison qui puisse s'énoncer en termes métier. Parmi les bonnes raisons : le coût ou le risque du changement bloque des objectifs métier importants ; la plateforme perd le support du fournisseur ou ses correctifs de sécurité ; les compétences pour la maintenir deviennent introuvables ; les coûts d'exploitation sont hors de proportion avec la valeur ; ou le système ne peut pas répondre à de nouvelles exigences d'échelle, de disponibilité

ou de réglementation. Les mauvaises raisons sont la mode, l'ennui et la conviction qu'une nouvelle technologie est automatiquement meilleure. La section 11.8 montre comment transformer une bonne raison en analyse de rentabilité.

Pourquoi les réécritures échouent

La réponse évidente à un vieux système pénible est de tout recommencer. En avril 2000, Joel Spolsky a publié un essai, « Things You Should Never Do, Part I », soutenant que la décision de tout réécrire de zéro compte parmi les pires erreurs stratégiques qu'un éditeur de logiciels puisse commettre. Son exemple était Netscape, dont le navigateur est resté près de trois ans sans version majeure pendant la réécriture du code, période durant laquelle les concurrents ont pris le marché. Son argument central portait sur la connaissance : le vieux code paraît désordonné en partie parce qu'il contient des années de corrections de bugs pour des situations dont personne ne se souvient, et le jeter, c'est jeter aussi ces corrections.

Les raisons pour lesquelles une réécriture big bang échoue sont aujourd'hui bien comprises. La valeur n'arrive qu'à la fin, si bien que le métier attend des mois ou des années et perd patience. L'ancien système continue de changer pendant que le nouveau est construit : la cible bouge, et l'organisation soit gèle l'ancien système (et l'activité avec lui), soit paie pour faire chaque changement deux fois. Des comportements cachés sont oubliés, car personne ne peut énumérer toutes les règles que l'ancien système applique. Les estimations sont optimistes, car les fonctionnalités visibles ne sont qu'une fraction du travail réel. La migration des données et la bascule sont laissées pour la fin, au moment où elles sont les plus difficiles et les plus risquées. Et l'équipe se divise entre un groupe prestigieux chargé du nouveau système et un groupe démoralisé chargé de l'ancien. L'effet du second système de Fred Brooks, tiré de *The Mythical Man-Month* (1975), ajoute un risque supplémentaire : les concepteurs qui remplacent un système qu'ils connaissent ont tendance à charger le nouveau de toutes les idées que l'ancien ne pouvait pas accueillir.

Les grandes migrations qui basculent en une seule étape peuvent aussi échouer de façon très publique. En avril 2018, la banque britannique TSB a transféré les données et les services de ses clients vers une nouvelle plateforme informatique ; la plateforme a immédiatement connu de graves problèmes, toutes ses agences et une part importante de ses 5,2 millions de clients ont été touchées, et la banque n'a retrouvé un fonctionnement normal qu'en décembre 2018. En décembre 2022, la Financial Conduct Authority et la Prudential Regulation Authority ont infligé à TSB des amendes d'un total de 48,65 millions de livres sterling, estimant qu'elle n'avait pas correctement planifié, organisé et contrôlé la migration, ni géré les risques liés à son fournisseur externalisé.

Les praticiens ne s'accordent pas tous à dire que les réécritures sont toujours une erreur. Une réécriture peut être le bon choix quand le système est petit, que son comportement est bien spécifié et couvert par des tests, que la plateforme est vraiment dans une impasse, ou que le métier lui-même a tellement changé que l'ancien comportement n'est plus souhaité. Même alors, les réécritures les plus sûres sont livrées de façon incrémentale, l'ancien et le nouveau système fonctionnant côte à côte, ce qui est l'objet du reste de cette section.

Stratégies de migration : les R

Quand une organisation passe en revue tout un portefeuille d'applications, par exemple avant de quitter un centre de données, elle a besoin d'un vocabulaire pour décider quoi faire de chacune. Gartner a décrit cinq options de ce type au début des années 2010. En 2016, Stephen Orban, d'AWS, a publié six stratégies, et AWS en a ensuite ajouté une septième, ce qui donne les sept R utilisés aujourd'hui dans ses recommandations de migration. D'autres fournisseurs cloud et cabinets de conseil utilisent des listes semblables, avec des noms légèrement différents. La liste est surtout utile pour rappeler que reconstruire n'est qu'une option parmi plusieurs, et généralement la plus coûteuse.

Les sept R (d’après AWS Prescriptive Guidance)

Stratégie	Signification	Convient quand	Changement d’architecture
Retirer	Éteindre l’application et archiver ses données	Personne ne l’utilise, ou sa fonction existe déjà ailleurs	Aucun ; supprimer les dépendances envers elle
Conserver	La laisser où elle est pour l’instant et y revenir plus tard	Aucune raison pressante de la déplacer, ou d’autres travaux doivent passer d’abord	Aucun pour l’instant
Réhéberger (lift and shift)	La déplacer sans modification vers une nouvelle infrastructure, comme des machines virtuelles dans le cloud	Le facteur principal est de quitter rapidement un centre de données	Aucun ; les problèmes la suivent
Relocaliser	Déplacer des plateformes entières, comme un parc virtualisé, sans modifier les applications	Une plateforme compatible existe à destination	Aucun
Racheter	La remplacer par un produit, souvent un logiciel en tant que service	La capacité est générique et un bon produit existe	Intégration et migration des données ; perte de la personnalisation
Changer de plateforme	La déplacer avec quelques optimisations ciblées, comme une base de données gérée	Des gains peu coûteux sont possibles sans refonte	Faible et local
Refactoriser ou réarchitecturer	La repenser, souvent de façon incrémentale, pour répondre à de nouveaux besoins de qualité ou métier	La capacité est cœur de métier, change souvent et sa structure bloque le changement	Important ; utiliser les patrons incrémentaux ci-dessous

GUIDE DE DÉCISION

Guide de décision : choisir une stratégie pour une application

Posez ces questions pour chaque application du portefeuille, dans l'ordre. Les réponses viennent des données d'utilisation, du responsable métier et de l'architecture, pas des préférences technologiques.

Q1 **Quelqu'un a-t-il encore besoin de ce que fait cette application ?**

- **Oui** Passez à la question suivante.
- **Non** Retirez-la, après avoir archivé les données que vous êtes légalement tenu de conserver.

Q2 **Existe-t-il un produit qui répond au besoin, pour une capacité qui ne différencie pas l'entreprise ?**

- **Oui** Envisagez le rachat, et prévoyez un budget pour l'intégration et la migration des données.
- **Non** Passez à la question suivante.

Q3 **Existe-t-il une raison pressante d'agir maintenant, comme la fin du support, la sortie d'un centre de données, le coût ou un changement bloqué ?**

- **Oui** Passez à la question suivante.
- **Non** Conservez-la et fixez une date pour réexaminer la décision.

Q4 **Le problème principal tient-il à l'endroit où elle s'exécute plutôt qu'à la manière dont elle est construite ?**

- **Oui** Réhébergez-la ou relocalisez-la, ou changez-la de plateforme là où un service géré apporte des gains peu coûteux.
- **Non** Passez à la question suivante.

Q5 **S'agit-il d'une capacité cœur de métier dont la structure bloque des changements fréquents et précieux ?**

- **Oui** Refactorisez-la ou réarchitecturez-la de façon incrémentale, avec un strangler fig et branch by abstraction.
- **Non** Changez-la de plateforme et contenez-la : ajoutez des tests, corrigez les pires points chauds et évitez une grande refonte.

Le strangler fig

Martin Fowler a nommé le patron incrémental le plus utilisé d'après les figuiers étrangleurs (strangler figs) qu'il a vus dans les forêts tropicales du Queensland en 2001 : des plantes grimpantes qui germent dans les branches d'un arbre hôte, poussent jusqu'au sol et entourent peu à peu l'hôte jusqu'à ce qu'il meure et que le figuier tienne debout seul. Il l'a décrit pour la première fois sous le nom de strangler application en 2004, l'a ensuite renommé strangler fig application et a profondément révisé l'article en 2024. La version logicielle, généralement appelée simplement strangler fig (figuier étrangleur), maintient l'ancien système en fonctionnement pendant que de nouveaux composants poussent autour de lui et reprennent son comportement un morceau à la fois, jusqu'à ce que l'ancien système ne fasse plus rien et puisse être éteint.

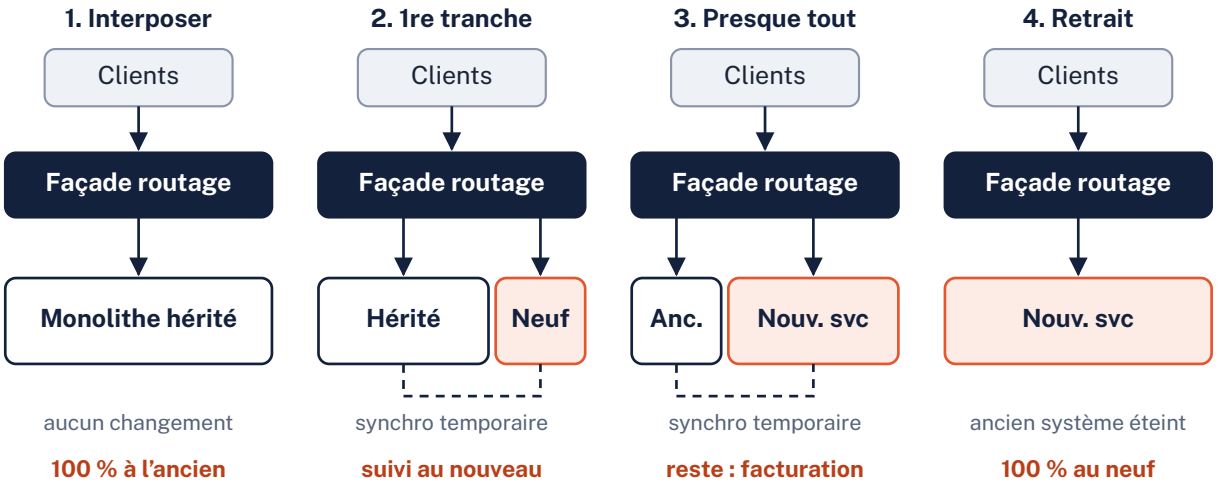
Le mécanisme comporte trois parties. D'abord, une couche d'interception est placée devant le système hérité : un reverse proxy, une API gateway ou une façade de routage par laquelle passent toutes les requêtes. Au départ, elle envoie tout à l'ancien système et ne change rien. Ensuite, une tranche de comportement est choisie le long d'un point d'articulation (seam : un endroit où le comportement peut être redirigé sans modifier une grande partie de l'ancien code, au sens de Feathers), reconstruite sous la forme d'un nouveau composant et activée en changeant une règle de routage, généralement d'abord pour une petite part du trafic. Enfin, le processus se répète jusqu'à ce que le système hérité ne traite plus rien ; il est alors retiré. Tout au long, la couche de routage rend chaque étape réversible : si le nouveau composant se comporte mal, la règle est rebasculée.

Le code qui permet à l'ancien et au nouveau de coexister (la façade de routage, les adaptateurs, les tâches de synchronisation des données) relève de l'architecture de transition : du logiciel construit pour être jeté. Les équipes rechignent souvent à l'écrire, car cela semble du gaspillage. L'argument de Fowler, et l'expérience qui le sous-tend, est que ce coût achète un risque réduit et une valeur bien plus précoce, ce qui est presque toujours un bon compromis. Le vrai danger est l'inverse : une migration qui cale à mi-chemin, laissant les deux systèmes et tout le code de transition en production indéfiniment. Chaque tranche doit donc retirer quelque chose de l'ancien système, et l'avancement doit se mesurer à ce qui a été éteint, pas seulement à ce qui a été construit.

Ian Cartwright, Rob Horn et James Lewis, de Thoughtworks, ont documenté une famille de techniques apparentées sous le titre *Patterns of Legacy Displacement* sur martinfowler.com (à partir de 2021). Ils présentent le travail comme quatre activités qui ne forment pas une séquence fixe : comprendre les résultats visés, décomposer le problème en parties, livrer ces parties de façon incrémentale, et faire évoluer l'organisation pour que le nouveau système ne

se dégrade pas comme l'ancien. Ils remettent aussi en question le réflexe de reproduire chaque fonctionnalité : une grande partie du comportement d'un système hérité n'est peut-être plus souhaitée, et le vérifier coûte moins cher que de le reconstruire.

Une migration strangler fig en quatre étapes



Une façade de routage permet au trafic de passer tranche par tranche du système hérité aux nouveaux services, avec une synchronisation temporaire des données, jusqu'à ce que le système hérité puisse être éteint.

Branch by abstraction (branchement par abstraction)

Le strangler fig agit à la périphérie d'un système, là où les requêtes peuvent être routées. Certains remplacements doivent se faire au cœur d'une seule base de code : remplacer un framework de persistance, un moteur de tarification ou une couche de messagerie maison utilisée par des centaines de points d'appel. Branch by abstraction (branchement par abstraction), un nom introduit par Paul Hammant lorsqu'il défendait le développement basé sur le tronc et que Martin Fowler a décrit en 2014, y parvient sans branche de gestion de versions à longue durée de vie. Le système reste livrable à chaque étape, si bien que le changement peut s'étaler sur des semaines ou des mois pendant que d'autres travaux continuent sur la même ligne principale.

Branch by abstraction, pas à pas

1. **Introduire une abstraction**
   Créez une interface devant le composant à remplacer, avec une implémentation qui délègue simplement au code existant. Le comportement ne change pas.

2. **Faire passer les appelants par l'abstraction**
   Modifiez progressivement les points d'appel pour qu'ils utilisent l'interface au lieu de l'ancien composant directement. Chaque modification est petite et mise en production normalement.

3. **Construire la nouvelle implémentation**
   Écrivez le remplaçant derrière la même interface, en le testant avec les mêmes tests et, idéalement, par rapport au comportement en production lors d'une exécution en parallèle.

4. **Basculer progressivement**
   Utilisez un feature flag ou la configuration pour envoyer certains appelants ou certains clients vers la nouvelle implémentation, en augmentant la part à mesure que la confiance grandit.

5. **Supprimer l'ancienne implémentation**
   Une fois que plus rien n'utilise l'ancien code, supprimez-le, et supprimez aussi l'abstraction si elle ne justifie plus sa place.

Prouver l'équivalence : exécution en parallèle et dark launching

La question la plus difficile de toute migration est de savoir si la nouvelle implémentation se comporte vraiment comme l'ancienne. Une exécution en parallèle (parallel run) y répond avec des preuves issues de la production : les deux implémentations reçoivent les mêmes entrées, l'ancienne continue de faire foi et c'est sa réponse qui est renvoyée à l'utilisateur, puis les deux résultats sont comparés et chaque écart est enregistré. GitHub a publié sa bibliothèque Scientist en 2016 exactement dans ce but : elle exécute côte à côte un contrôle et un candidat et signale les divergences ; des outils semblables existent pour de nombreux langages, et la technique est simple à écrire soi-même, comme le montre l'esquisse ci-dessous.

Le dark launching (lancement masqué), que Fowler décrit comme le fait d'appeler un nouveau comportement backend en production sans que les utilisateurs en voient les effets, en est proche. Il sert surtout à tester les performances et la stabilité du nouveau chemin sous une charge réelle avant qu'un utilisateur en dépende, tandis qu'une exécution en parallèle se concentre sur l'exactitude. Tous deux diffèrent d'une canary release (déploiement canari), dans laquelle une petite part des utilisateurs reçoit réellement le nouveau comportement.

Les deux techniques ont leurs pièges. Le candidat ne doit avoir aucun effet de bord visible de l'extérieur : il ne doit pas envoyer un second e-mail, débiter une carte deux fois ni écrire dans la base de données partagée ; les exécutions en parallèle sont donc plus faciles pour les lectures et les calculs, et les écritures exigent un bac à sable ou un stockage séparé. Les comparaisons doivent ignorer les différences légitimes, comme les horodatages, l'ordre et le formatage des

nombres à virgule flottante, sinon le rapport des divergences se noie dans le bruit. Et exécuter les deux chemins consomme de la capacité : la part du trafic échantillonnée est donc souvent limitée.

```
def quote_delivery_fee(order):
    old = legacy_pricing.quote(order)          # still authoritative
    try:
        new = new_pricing.quote(order)         # must have no side effects
        if new.amount_pence ≠ old.amount_pence:
            metrics.increment("pricing.mismatch")
            log.info("pricing mismatch", order_id=order.id,
                     old=old.amount_pence, new=new.amount_pence)
        else:
            metrics.increment("pricing.match")
    except Exception:
        metrics.increment("pricing.candidate_error")   # never hurts the user
    return old
```

Une exécution en parallèle minimale : le résultat de l'ancien système est toujours renvoyé, tandis que la réponse du candidat est comparée et les divergences comptées.

Les données : la partie la plus difficile

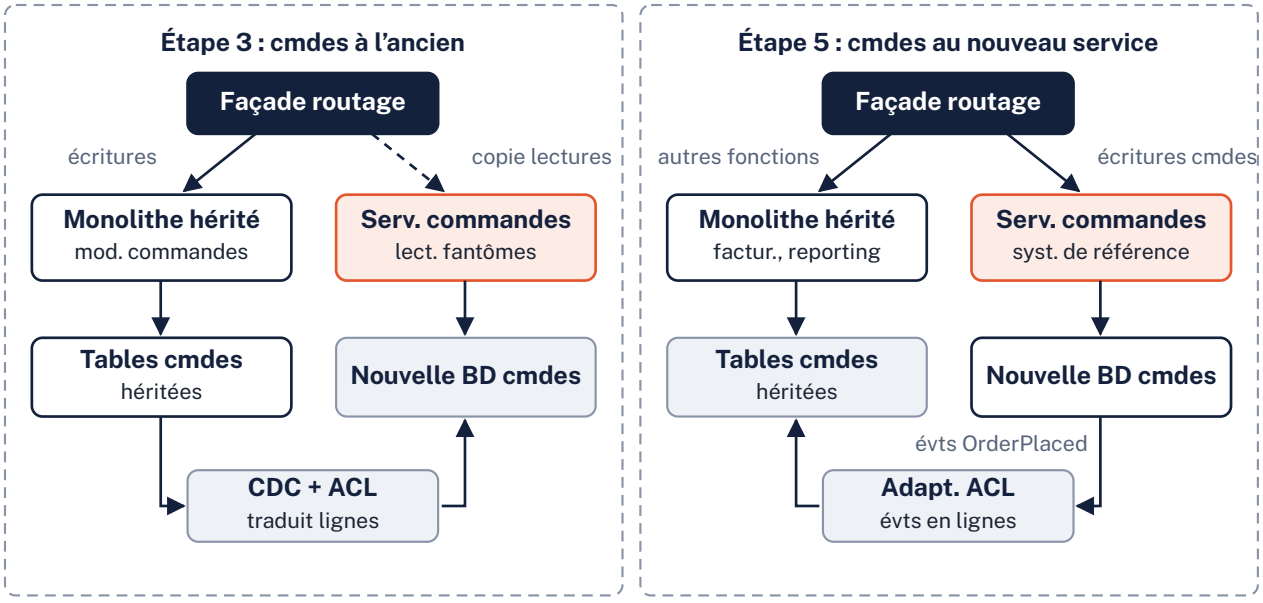
Router les requêtes est relativement facile ; déplacer les données et les garder cohérentes pendant que deux systèmes en partagent la responsabilité, c'est là que la plupart des migrations peinent. La première règle est celle du chapitre 7 sur la propriété des données : pour chaque type de données, à chaque étape de la migration, l'équipe doit pouvoir dire quel système est le système de référence (system of record), quelles copies existent et à quel point chacune peut être périmée. Le diagramme ci-dessous montre l'inversion typique : le système hérité possède d'abord les données et le nouveau composant en lit une copie ; plus tard, le nouveau composant les possède et le système hérité en reçoit une copie pour les modules qui en ont encore besoin.

La capture des données modifiées est la manière habituelle de copier des données hors d'une base de données héritée sans modifier le code hérité : un outil comme Debezium lit le journal des transactions de la base et publie chaque changement sous forme d'événement, le même mécanisme que celui utilisé à la section 6.7 pour relayer une outbox. L'interception d'événements, l'un des patrons de remplacement des systèmes hérités (legacy displacement), agit à l'autre bout : elle capte les requêtes ou messages qui modifient l'état en route vers le système hérité et en transmet des copies (ou, plus tard, les originaux) au nouveau composant. Dans les deux cas, ne laissez jamais les deux systèmes accepter indépendamment des écritures sur les mêmes données avec une synchronisation bidirectionnelle, sauf si vous n'avez pas d'autre choix ; les conflits sont alors presque garantis.

Les données héritées correspondent rarement au nouveau modèle. Les codes de statut sont des entiers aux significations non documentées, une même table sert à trois usages, et les dates sont stockées en heure locale. Une couche anticorruption, le patron de la conception pilotée par le domaine qui protège un modèle de celui d'un voisin, traduit entre les deux pour que les bizarreries du système hérité ne s'infiltrent pas dans le nouveau code. Elle fait partie de l'architecture de transition et doit disparaître avec le système hérité.

La migration finale des données, qui déplace l'historique et effectue la bascule, mérite son propre plan. Profilez d'abord les données héritées pour trouver les doublons, les orphelins et les valeurs invalides. Écrivez explicitement la correspondance et testez-la sur une copie complète, à la taille de la production. Faites un rapprochement après chaque exécution avec des nombres de lignes, des sommes des colonnes monétaires et des sommes de contrôle, pas par des contrôles ponctuels. Répétez la bascule, chronométrez-la et mettez par écrit le retour arrière. Lorsque c'est possible, migrez par cohortes (une région, un segment de clientèle, une gamme de produits) plutôt que tout à la fois, afin qu'un problème ne touche qu'une fraction des clients et vous apprenne quelque chose avant la cohorte suivante.

Le système de référence bascule pendant la migration



Au début, la capture des données modifiées copie les lignes héritées dans le nouveau stockage ; une fois que le nouveau service possède les commandes, ses événements sont retraduits en lignes héritées pour les modules pas encore migrés.

SI VOUS DÉBUTEZ

Pas à pas : pourquoi deux systèmes ne doivent pas posséder tous deux les commandes

L'idée qui demande le plus d'attention ici est de garder un seul système de référence pour chaque type de données pendant que la propriété passe de l'ancien système au nouveau. Le chapitre 7 a enseigné la règle ; ce qui est nouveau ici, c'est que le propriétaire change alors que les deux systèmes ont encore des utilisateurs.

- À mi-parcours, supposons que le module de commandes hérité et le nouveau service de commandes acceptent tous deux des modifications des commandes, avec une tâche qui copie les changements dans les deux sens toutes les quelques secondes.
- Un client modifie l'adresse de livraison d'une commande dans l'app ; la façade de routage envoie cette modification au nouveau service de commandes. Quelques secondes plus tard, un agent du support annule la même commande dans les outils de support, qui utilisent encore le module hérité.
- Si la tâche copie des commandes entières, c'est la dernière copie arrivée qui l'emporte : l'annulation peut être défaite, ou le changement d'adresse perdu. Avec des commandes qui changent toute la journée, de tels conflits sont presque garantis.
- Le remède : à chaque étape, seul le système de référence accepte les modifications des commandes, et l'autre en détient une copie en lecture seule. Au début, le propriétaire est le module hérité, et la capture des données modifiées transmet en continu chacune de ses modifications au nouveau service.
- Plus tard, la propriété est transférée : le nouveau service accepte les modifications, et ses événements sont retraduits en lignes héritées pour les modules pas encore migrés. Le transfert est une étape planifiée, avec un retour arrière prêt, jamais une période où les deux décident ; l'exemple commenté ci-dessous le fait ville par ville, si bien que chaque commande a toujours un seul propriétaire.
- **La leçon :** pour chaque type de données, à chaque étape, nommez le seul système qui en est propriétaire et le sens dans lequel circulent les copies. Laisser les deux systèmes accepter des modifications des mêmes données est un dernier recours.

Les leçons de Kill It with Fire

Kill It with Fire: Manage Aging Computer Systems (and Future Proof Modern Ones) (No Starch Press, 2021), de Marianne Bellotti, s'appuie sur son expérience de la modernisation de systèmes publics et commerciaux, y compris de parcs mainframe. Plusieurs de ses thèmes méritent d'être appliqués à toute modernisation.

La modernisation doit se justifier par la valeur qu'elle apporte et le risque qu'elle supprime, pas par l'âge de la technologie ; une vieille technologie bien comprise et qui fonctionne peut être la partie la moins risquée du parc. Les choix technologiques évoluent par cycles : une équipe qui remplace un système uniquement parce que sa conception est passée de mode risque de construire l'héritage de la génération suivante. L'amélioration incrémentale du système existant est généralement plus sûre que le remplacement, et une réécriture complète exige une justification exceptionnellement solide. Enfin, la modernisation est un problème d'organisation autant qu'un problème technique : elle a besoin de premières victoires visibles pour entretenir l'élan et le moral, de critères clairs pour savoir quand elle est terminée, et d'une attention aux incitations qui ont fait se dégrader l'ancien système, faute de quoi le nouveau suivra le même chemin.

EXEMPLE COMMENTÉ

Exemple commenté : extraire le module de commandes du monolithe par étranglement, en six étapes

Cet exemple revient sur la manière dont la plateforme de livraison de repas a obtenu le service de commandes séparé évalué à la section 11.1. Plus tôt dans son histoire, le monolithe d'origine de la plateforme contenait un module de commandes qui gérait la passation, la modification, l'annulation et le suivi des commandes. Les changements liés aux commandes étaient devenus le principal goulot d'étranglement : ils exigeaient de déployer tout le monolithe, et les pics du vendredi soir ralentissaient toutes les autres fonctionnalités. Quatorze autres modules, dont la facturation, le reporting et les outils du support client, lisaient directement les tables des commandes. L'équipe a planifié une migration incrémentale vers un service de commandes séparé.

1. Fixer les résultats visés et trouver le point d'articulation	Résultats visés : des changements liés aux commandes déployables chaque jour et de façon indépendante ; trois fois le pic actuel du vendredi sans effet sur les autres fonctionnalités ; aucune commande perdue ou dupliquée. L'analyse des journaux de requêtes confirme que seul le module de commandes écrit dans les tables des commandes et que tout le trafic des commandes entre par les endpoints /api/orders. L'équipe enregistre plusieurs semaines de requêtes de production et les transforme en tests de caractérisation qui capturent le comportement actuel, bizarreries comprises.
2. Insérer la façade de routage	Une API gateway est placée devant le monolithe et, au départ, lui route tout. Mise en production seule, elle ne change aucun comportement ; l'équipe vérifie la latence ajoutée (quelques millisecondes au 95e centile) et s'exerce à basculer les routes. Retour arrière : retirer la gateway du chemin.
3. Construire le nouveau service dans l'ombre	Le nouveau service de commandes a son propre stockage. Le système hérité reste le système de référence : la capture des données modifiées copie les lignes de commandes, en passant par une couche anticorruption qui convertit les codes de statut entiers et les dates en heure locale vers le nouveau modèle. Les endpoints de lecture fonctionnent en exécution en parallèle, comparés aux réponses du système hérité ; le taux de divergence part d'environ 2 % (arrondi des frais, fuseaux horaires) et est ramené sous 0,01 % avant d'aller plus loin.
4. Déplacer les lectures	Les lectures du suivi des commandes sont routées vers le nouveau service pour 5 %, puis 25 %, puis la totalité du trafic, en surveillant les taux d'erreur et la latence à chaque étape. Retour arrière : une règle de routage.

5. Déplacer les écritures, une ville à la fois	Le nouveau service devient le système de référence pour les commandes passées dans une ville pilote, puis pour toutes les villes en six semaines. Il publie les événements OrderPlaced et OrderCancelled au moyen d'une outbox transactionnelle ; un adaptateur de transition les traduit en lignes au format hérité, pour que la facturation et le reporting continuent de fonctionner sans changement. Un rapprochement quotidien compare les nombres et les totaux monétaires entre les deux stockages. Ces modules passent ensuite aux événements ou à l'API des commandes au cours des trimestres suivants.
6. Migrer l'historique et retirer	Trois années de commandes historiques sont copiées par lots à débit limité et rapprochées par nombres et sommes de contrôle. Quand les journaux d'audit de la base de données ne montrent aucune lecture des anciennes tables de commandes pendant 30 jours, l'équipe supprime l'adaptateur, le pipeline de capture des données modifiées et les anciennes règles de routage, puis supprime du monolithe le code et les tables des commandes. La migration n'est terminée qu'à ce moment-là.

Chaque étape livre quelque chose d'utilisable ou supprime un risque, et chaque étape sauf la dernière dispose d'un retour arrière peu coûteux.

Le système de référence des commandes est sans ambiguïté à chaque étape, et les données circulent dans un seul sens à la fois.

L'architecture de transition est planifiée, mesurée et délibérément supprimée ; la réussite se mesure à ce qui a été éteint.

ATTENTION

Erreurs courantes

- Commencer par la partie la plus difficile et la plus centrale du système au lieu d'une tranche qui apporte de la valeur et apprend à l'équipe comment migrer.
- Viser une parité fonctionnelle complète sans se demander quelles anciennes fonctionnalités sont encore utilisées.
- Une synchronisation bidirectionnelle dans laquelle les deux systèmes acceptent des écritures sur les mêmes données.
- Des migrations qui calent à mi-chemin, laissant deux systèmes et tout le code de transition en fonctionnement pendant des années.
- Traiter la migration des données comme une tâche finale au lieu de la concevoir, de la répéter et d'en faire le rapprochement dès le début.
- Geler tout changement du système hérité pendant toute la durée, ce qui transforme la migration en une longue attente pour le métier.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi une réécriture big bang tend-elle à perdre des comportements dont les utilisateurs dépendent ?

Parce que des années de corrections de bugs et de règles métier n'existent que dans l'ancien code et dans les habitudes des utilisateurs ; personne ne peut toutes les énumérer, si bien qu'une réécriture à partir d'une spécification en oublie certaines.

▼ Qu'apporte la façade de routage à une migration strangler fig ?

Elle permet de déplacer le trafic tranche par tranche, et de le ramener si quelque chose tourne mal, sans modifier le système hérité à chaque étape.

▼ Qu'est-ce que l'architecture de transition, et pourquoi suivre sa suppression ?

Du code temporaire, comme des règles de routage, des adaptateurs et des tâches de synchronisation, qui permet à l'ancien et au nouveau de coexister ; suivre sa suppression protège contre une migration qui cale, avec les deux systèmes en fonctionnement indéfiniment.

▼ Quand le rachat peut-il être un meilleur choix que la refactorisation ?

Quand la capacité est générique plutôt que différenciante et qu'un bon produit existe : la reconstruire consacrerait des efforts là où l'entreprise ne se distingue pas de la concurrence.

▼ À vous : Un système de billetterie vieux de 15 ans vend toujours des billets, mais les modifications sont lentes et risquées. Le conseil d'administration privilégie une réécriture de deux ans avec une bascule unique. Que recommanderiez-vous, et comment géreriez-vous les données pendant la migration ?

Évitez le big bang : la valeur arrive tard, les exigences bougent et des comportements cachés se perdent. Utilisez un strangler fig, en faisant passer le trafic par une façade et en déplaçant une tranche à la fois, en commençant par quelque chose d'utile mais à faible risque, comme la consultation des places. Utilisez des exécutions en parallèle pour comparer les résultats pendant que l'ancien système reste celui qui fait foi. Chaque jeu de données a besoin d'un seul système de référence à chaque étape, maintenu en phase par la capture des données modifiées. Terminez en retirant l'ancien système.

SOURCES

- Martin Fowler: Strangler Fig Application: https://martinfowler.com/bliki/StranglerFigApplication.html

- Joel Spolsky: Things You Should Never Do, Part I (2000): https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/
- Cartwright, Horn and Lewis: Patterns of Legacy Displacement: https://martinfowler.com/articles/patterns-legacy-displacement/
- AWS Prescriptive Guidance: migration strategies (the 7 Rs): https://docs.aws.amazon.com/prescriptive-guidance/latest/large-migration-guide/migration-strategies.html
- Martin Fowler: Branch By Abstraction (2014): https://martinfowler.com/bliki/BranchByAbstraction.html

SECTION 11.7 SUR 11.8 · AVANCÉ · 28 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Leadership, influence et prise de décision

AVANT DE LIRE

Cette section s'appuie sur :

- 1.4 Le rôle de l'architecte
- 1.6 Décisions et compromis
- 4.8 Communiquer

Termes déjà vus qu'elle utilise : Enregistrement de décision d'architecture (ADR), Partie prenante, Matrice pouvoir/intérêt, Kata d'architecture

L'IDÉE EN BREF

Pensez à une guide de montagne qui conduit un groupe d'alpinistes expérimentés. Elle ne peut ordonner à personne de monter. Les gens la suivent parce qu'elle connaît l'itinéraire, explique honnêtement les risques, écoute quand quelqu'un repère un danger, reconnaît quand elle n'est pas sûre et a déjà eu raison par le passé. Quand le groupe n'est pas d'accord sur l'itinéraire, elle veille à ce que chacun soit entendu, convient de la manière dont le choix sera fait, puis aide tout le monde à s'y engager. La plupart des architectes sont dans la même position : responsables de l'itinéraire, mais sans pouvoir de commander ; ils dirigent donc par la confiance, un raisonnement clair et de bonnes décisions prises ensemble.

CE QU'IL FAUT SAVOIR

- Les architectes portent généralement des responsabilités sans autorité formelle ; leur influence repose donc sur l'expertise, l'information et les relations plutôt que sur le pouvoir lié à la position (French et Raven, 1959).
- La confiance croît avec la crédibilité, la fiabilité et l'intimité, et diminue avec l'orientation vers soi (Maister, Green et Galford, 2000) ; la sécurité psychologique permet aux équipes de soulever les problèmes tant qu'ils sont peu coûteux à corriger.
- Réglez les droits de décision et la méthode de décision avant de débattre du fond ; le consentement adopte une proposition sauf si une objection argumentée subsiste, tandis que le consensus exige le soutien de tous.
- La négociation raisonnée sépare les personnes du problème, se concentre sur les intérêts plutôt que sur les positions et s'accorde sur des critères objectifs ; choisissez délibérément un mode de gestion des conflits et faites remonter les désaccords ensemble.

- L'ascenseur de l'architecte de Hohpe relie le penthouse (la stratégie) à la salle des machines (la réalité technique), en faisant monter les risques et les options et descendre la stratégie et ses raisons.
- Faire grandir les autres par le mentorat, le coaching, le parrainage et la délégation, et soutenir les parcours de carrière staff-plus (Larson, 2021 ; Reilly, 2022), évite que l'architecte devienne un goulet d'étranglement.
- Les groupes ajoutent leurs propres pièges décisionnels, comme la déférence envers l'ancienneté, l'escalade de l'engagement, le bikeshedding et la décision par épuisement, chacun avec une parade pratique.

Diriger sans autorité

La plupart des architectes n'encadrent pas les personnes qui construisent ce qu'ils conçoivent. Les équipes rendent compte à des engineering managers, les priorités appartiennent aux product managers, les budgets aux directeurs. Pourtant, on attend de l'architecte qu'il veille à ce que les décisions importantes soient bien prises et respectées dans de nombreuses équipes, ce qui est la responsabilité centrale décrite à la section 1.4. L'écart entre la responsabilité et l'autorité formelle est la condition qui définit le métier, et le combler est une affaire d'influence.

L'étude classique de John French et Bertram Raven sur le pouvoir social (1959) distinguait les bases du pouvoir qui viennent avec une position (l'autorité légitime, la capacité de récompenser et la capacité de punir) de celles qui appartiennent à une personne (l'expertise et le pouvoir de référence, c'est-à-dire le respect et l'identification que ressentent les autres) ; Raven a ensuite ajouté le pouvoir informationnel, la capacité d'apporter des faits qui font changer d'avis. Les architectes détiennent rarement beaucoup de pouvoir lié à leur position, et quand c'est le cas, l'utiliser coûte cher : une décision imposée par le rang est appliquée, pas appropriée, et ceux qui n'étaient pas d'accord cessent de vous signaler les problèmes. Une influence architecturale durable repose presque entièrement sur l'expertise, l'information et les relations.

Ce n'est pas une compétence relationnelle plaquée sur le travail technique. Une conception solide que personne n'adopte a le même effet que pas de conception du tout, et une conception médiocre que les équipes comprennent et soutiennent fait souvent mieux qu'une conception brillante qu'elles contournent. La suite de cette section traite l'influence comme un ensemble de pratiques qui s'apprennent : instaurer la confiance, animer les décisions, gérer les conflits, communiquer avec la hiérarchie, faire grandir les autres et éviter les pièges dans lesquels tombent les groupes.

Instaurer la confiance et la crédibilité

The Trusted Advisor (2000) de David Maister, Charles Green et Robert Galford propose un modèle utile. La confiance croît avec la crédibilité (vous savez de quoi vous parlez), la fiabilité (vous faites ce que vous dites) et l'intimité (les gens se sentent en sécurité pour aborder avec vous des sujets difficiles), et elle diminue avec l'orientation vers soi (le sentiment que vous servez vos propres intérêts, comme avoir raison, paraître brillant ou protéger votre conception). L'enseignement du modèle pour les architectes est que l'orientation vers soi se trouve sous la barre de fraction : elle divise tout le reste. Un architecte brillant et fiable, mais qui a visiblement besoin de gagner chaque discussion, n'inspire pas confiance.

En pratique, la crédibilité se construit par de petits gestes visibles. Ayez raison sur les petites choses avant de demander qu'on vous fasse confiance sur les grandes. Restez assez proche du code pour en comprendre le coût réel : écrivez le spike, relisez la pull request, assistez à une revue d'incident. Dites ce que vous ne savez pas et comment vous le découvririez. Tenez chacun de vos engagements, surtout les plus ennuyeux. Reconnaissez publiquement le mérite des autres et assumez vos propres erreurs, publiquement aussi. Expliquez le raisonnement derrière chaque principe pour que chacun puisse l'appliquer dans des situations que vous n'aviez pas prévues.

La confiance dépend aussi du climat qui vous entoure. Les recherches d'Amy Edmondson sur la sécurité psychologique, à commencer par une étude d'équipes de travail en 1999, ont montré que les équipes dont les membres estiment pouvoir sans risque admettre leurs erreurs et contester les autres apprennent davantage et obtiennent de meilleurs résultats. Les architectes façonnent ce climat plus qu'ils ne le pensent. Réagir aux mauvaises nouvelles avec curiosité plutôt qu'en cherchant un coupable, solliciter explicitement les objections (qu'est-ce qui ferait échouer ceci ?) et remercier ceux qui en soulèvent, voilà ce qui permet d'entendre parler d'une conception défaillante tant qu'elle est encore peu coûteuse à corriger.

Animer les décisions

Beaucoup de débats d'architecture portent en réalité sur le processus : qui décide, qui doit être consulté et comment les désaccords seront tranchés. Régler les droits de décision avant de discuter du fond retire une bonne part de la tension. La section 11.4 a décrit le processus de consultation (advice process), dans lequel chacun peut prendre une décision à condition de demander l'avis des personnes concernées et des personnes compétentes ; d'autres organisations utilisent un forum de revue d'architecture, un responsable des décisions désigné par domaine, ou des matrices de rôles comme RACI. Le mécanisme précis compte moins que le fait qu'il soit connu à l'avance.

Quels que soient les droits de décision en vigueur, l'architecte est souvent le facilitateur : la personne qui formule la question, s'assure que les bonnes personnes et les bons éléments probants sont présents et guide le groupe vers une décision qui tienne. Différentes méthodes de décision conviennent à différentes situations, et choisir la méthode délibérément fait partie du métier.

Deux méthodes sont souvent confondues. Le consensus recherche une proposition que tout le monde soutient activement ; il crée un engagement fort, mais il est lent et donne à chacun un droit de veto de fait. La prise de décision par consentement, développée dans la sociocratie de Gerard Endenburg aux Pays-Bas à partir des années 1970, adopte une proposition sauf si quelqu'un soulève une objection argumentée selon laquelle elle causerait un préjudice ou ferait reculer le groupe. La question passe de êtes-vous tous d'accord ? à quelqu'un ne peut-il pas vivre avec cela, et pourquoi ? Une formule issue de la pratique sociocratique résume bien le seuil : assez bon pour maintenant, assez sûr pour essayer. Le consentement fonctionne particulièrement bien pour les décisions de type porte à double sens, que l'on peut réexaminer à mesure que les éléments probants arrivent. Le consensus approximatif (rough consensus) de l'IETF, décrit à la section 4.8, en est un proche parent.

Quand l'accord ne peut pas être trouvé, quelqu'un doit décider, et l'on doit déjà savoir qui. Le disagree and commit (exprimer son désaccord, puis s'engager), également abordé à la section 4.8, demande à chacun de défendre pleinement sa position avant la décision et de la soutenir pleinement ensuite. Cela ne fonctionne que si le désaccord a réellement été entendu, si la dissidence est consignée (dans l'enregistrement de décision d'architecture, avec ses raisons) et s'il existe un signal convenu qui rouvrirait la question.

Choisir une méthode de décision

Méthode	Comment ça marche	À utiliser quand	Principal risque
Décider seul	La personne responsable décide, éventuellement après une vérification rapide	Portes à double sens, urgences, décisions de portée limitée	Connaissances manquées ; faible engagement des autres
Consulter, puis décider (processus de consultation)	La personne qui décide doit demander l'avis des personnes concernées et des experts, puis décide et consigne pourquoi	La plupart des décisions d'architecture dans les équipes autonomes	Avis demandé pour la forme et ignoré
Consentement	Adopter sauf si une objection argumentée et majeure subsiste ; les objections améliorent la proposition	Décisions qui touchent un groupe et peuvent être réexaminées	Objections détournées en veto sur de simples préférences
Consensus	Continuer jusqu'à ce que tout le monde soutienne la proposition	Petits groupes, valeurs ou principes à fort enjeu qui exigent un engagement profond	Lenteur ; impasse ; décisions prises par défaut
Vote	Vote à la majorité ou pondéré entre les participants	Choisir parmi des options toutes acceptables	Une minorité perdante sans engagement ; la popularité l'emporte sur les éléments probants
Déléguer	Confier la décision, avec des contraintes, à une équipe ou une personne plus proche du travail	La décision est locale et les contraintes peuvent être énoncées	Déléguer sans le contexte ni les contraintes

SI VOUS DÉBUTEZ

Pas à pas : le consentement n'est pas le consensus

Le point qui fait le plus souvent trébucher ici est la différence entre le consensus, où chacun doit soutenir activement une proposition, et le consentement, où elle est adoptée sauf si quelqu'un soulève une objection argumentée. Suivez une petite décision selon les deux méthodes.

- Après la séance de risk-storming de la section 11.2, les six participants doivent convenir de la date à laquelle réexaminer la conception. L'architecte propose tous les trois mois.
- Par consensus : un développeur veut des revues mensuelles, l'ingénieure d'exploitation seulement avant les changements majeurs. La discussion se poursuit jusqu'à ce que les six soutiennent activement une même réponse, si bien que n'importe lequel d'entre eux peut la bloquer. Elle peut s'éterniser jusqu'à l'enlisement, ou jusqu'à ce que la voix la plus insistante l'emporte.
- Par consentement : la personne qui anime demande : « Quelqu'un ne peut-il pas vivre avec un rythme de tous les trois mois, et pourquoi ? » Le développeur préfère toujours un rythme mensuel, mais peut vivre avec un rythme trimestriel. Une préférence n'est pas une objection ; elle ne bloque donc pas la proposition.
- L'ingénieur sécurité objecte, avec une raison : un changement important du parcours de paiement est prévu le mois prochain, et un intervalle de trois mois le laisserait sans revue alors qu'il est le plus récent et le plus risqué. C'est une affirmation argumentée de préjudice ; la proposition est donc améliorée : tous les trois mois, plus une revue après le changement du paiement.
- Personne ne s'oppose à la proposition améliorée ; elle est donc adoptée. Elle n'a pas besoin d'être parfaite : assez bonne pour maintenant, assez sûre pour essayer, et facile à réexaminer plus tard.
- **La leçon :** le consentement demande « Quelqu'un ne peut-il pas vivre avec cela, et pourquoi ? » plutôt que « Sommes-nous tous d'accord ? ». Il convient aux décisions qui peuvent être réexaminées. Réservez le consensus aux rares décisions qui exigent l'engagement profond de tous, et convenez de la méthode avant le début du débat.

Conflits et politique organisationnelle

La politique organisationnelle, c'est simplement la manière dont des groupes aux intérêts différents, aux ressources limitées et avec leur propre carrière parviennent à des décisions. Ce n'est pas une corruption du travail d'architecture, mais son environnement. L'ignorer ne rend pas un architecte plus intègre, seulement moins efficace. Les outils des chapitres précédents

s’appliquent directement : la cartographie des parties prenantes et la matrice pouvoir/intérêt de la section 1.5 montrent qui est concerné et qui peut bloquer, et les traces écrites des compromis de la section 1.6 rendent les positions visibles.

Getting to Yes (1981) de Roger Fisher et William Ury reste le guide le plus utile de la négociation raisonnée, et ses quatre idées se transposent directement. Séparez les personnes du problème : la responsable des paiements n’est pas obstructive, elle s’inquiète pour les données de carte. Concentrez-vous sur les intérêts, pas sur les positions : nous ne devons pas utiliser Kafka est une position ; nous ne pouvons pas assumer davantage d’astreinte est un intérêt que plusieurs conceptions pourraient satisfaire. Imaginez des options à bénéfice mutuel avant de décider. Insistez sur des critères objectifs, convenus avant de comparer les options. Leur idée de la meilleure solution de rechange à un accord négocié (BATNA) est utile aussi : savoir ce qui se passe si aucun accord n’est trouvé indique à chacun ce que vaut l’accord.

Le modèle des modes de gestion des conflits de Kenneth Thomas et Ralph Kilmann (1974) décrit cinq façons de réagir à un conflit, selon le poids que vous accordez à vos propres préoccupations et à celles de l’autre partie. Aucune n’est toujours la bonne ; tout l’art est de choisir délibérément plutôt que de retomber dans ses habitudes.

Quelques habitudes pratiques aident dans tous les cas. Ne prenez jamais une partie prenante par surprise en réunion publique avec une proposition qui la concerne ; parlez-lui d’abord, en tête-à-tête (les organisations japonaises appellent cette consultation préparatoire nemawashi). Quand il faut faire remonter un désaccord, faites-le ensemble : les deux parties présentent le désaccord et les options à la personne qui peut décider, plutôt que de faire pression chacune de leur côté. Et séparez les désaccords sur les conceptions des jugements sur les personnes, dans ce que vous dites comme dans ce que vous écrivez.

Cinq modes de gestion des conflits (d'après Thomas et Kilmann)

Mode	Signification	Utile quand	Utilisé à l'excès, il conduit à
Compétition	Défendre fermement sa propre position	Questions de sûreté, de sécurité ou juridiques sur lesquelles vous devez tenir bon	Une obéissance sans engagement ; les gens cessent de signaler les problèmes
Collaboration	Travailler ensemble pour satisfaire pleinement les intérêts des deux parties	Décisions importantes où les préoccupations des deux parties sont légitimes	Des décisions lentes sur des sujets qui n'en demandaient pas tant
Compromis	Chaque partie renonce à une part de ce qu'elle veut	Le temps presse et les deux parties ont un pouvoir égal	Des conceptions incohérentes qui ne satisfont personne
Évitement	Reporter ou éluder la question	La question est anodine ou les émotions ont besoin de temps pour retomber	Des problèmes qui grossissent jusqu'à devenir des crises
Accommodement	Céder à la position de l'autre partie	Vous aviez tort, ou la question compte bien plus pour l'autre partie	Une crédibilité perdue et des principes abandonnés

Communiquer avec les dirigeants

La section 4.8 a traité de l'adaptation d'un message à son public et de l'importance de commencer par la réponse. Les dirigeants méritent un peu plus d'attention, parce qu'ils détiennent les budgets et l'autorité dont dépend le travail d'architecture, et parce que c'est le plus souvent en expliquant trop et en demandant trop peu que les architectes les perdent.

Formulez chaque sujet dans les termes que les dirigeants gèrent déjà : chiffre d'affaires, coût, risque, délai de mise sur le marché, expérience client, réglementation et personnes. Apportez des options plutôt qu'un plan unique, chacune avec son coût, son risque et ce à quoi elle renonce, et dites laquelle vous recommandez et pourquoi. Quantifiez là où vous le pouvez honnêtement, et dites quel degré de confiance accorder aux chiffres ; une fourchette avec une base explicite est plus crédible qu'une fausse précision. Dites explicitement de quelle décision ou de quel soutien vous avez besoin de leur part, et pour quand. Préparez-vous à la question que feriez-vous si c'était votre argent ? Et ne laissez jamais un dirigeant apprendre l'existence d'un risque grave par quelqu'un d'autre.

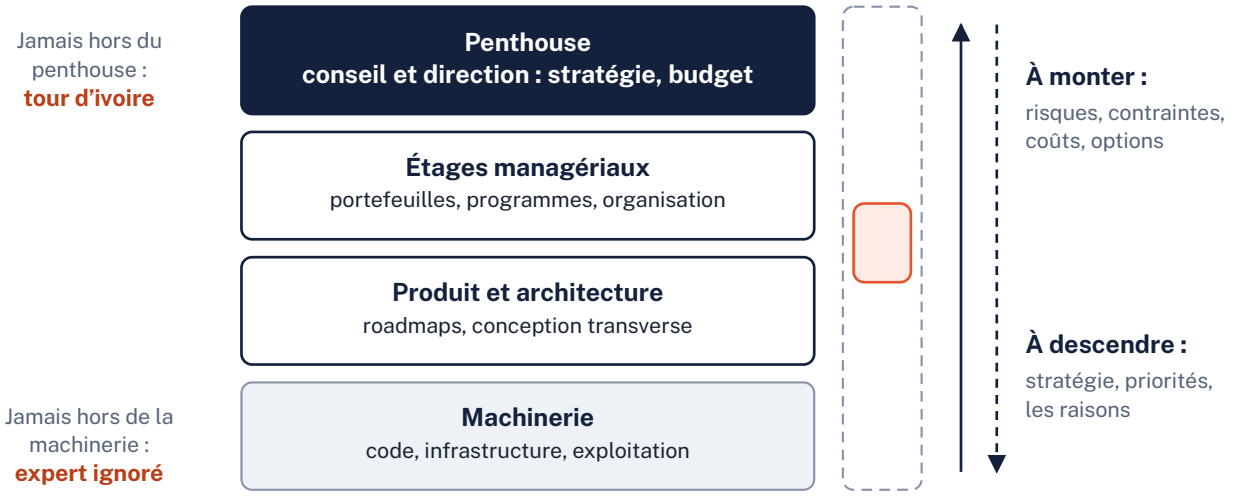
Les cadres supérieurs s'intéressent souvent moins au fonctionnement d'un système qu'au fait de savoir si ses responsables le maîtrisent. Montrer que les risques sont connus, attribués, mesurés et en voie de réduction (par exemple au moyen d'un registre des risques d'architecture ou des fonctions d'aptitude de la section 11.3) inspire plus confiance que n'importe quel diagramme. La section 11.8 montre comment présenter le travail d'architecture sous la forme que les dirigeants utilisent pour comparer les investissements : l'analyse de rentabilité.

L'ascenseur de l'architecte

The Software Architect Elevator (O'Reilly, 2020) de Gregor Hohpe donne l'image la plus mémorable de ce travail. Les grandes organisations ressemblent à de hauts immeubles. Au penthouse, les dirigeants fixent la stratégie et les budgets ; dans la salle des machines, les ingénieurs construisent et exploitent les systèmes ; entre les deux s'étagent de nombreux niveaux d'encadrement, et l'information qui les traverse est filtrée et déformée en chemin. L'ascenseur de l'architecte (architect elevator) de Hohpe circule entre le penthouse et la salle des machines et transporte l'information dans les deux sens : il descend la stratégie et ses raisons jusqu'aux personnes qui construisent les systèmes, et il monte les réalités techniques (contraintes, risques, coûts et options dont l'entreprise ignorait disposer) jusqu'aux personnes qui prennent les décisions stratégiques.

Pour Hohpe, la valeur de l'architecte réside dans le lien entre les étages. Un architecte qui reste au penthouse devient un stratège dans sa tour d'ivoire, dont les plans ignorent ce qui est techniquement possible ; celui qui reste dans la salle des machines devient un expert respecté dont les avertissements n'atteignent jamais les personnes qui pourraient agir. Bien prendre l'ascenseur, c'est changer de langage et de niveau de détail à chaque étage sans changer le fond, ce qui est exactement la compétence d'adaptation au public de la section 4.8, appliquée à toute une organisation.

Prendre l'ascenseur



La valeur de l'architecte vient du lien entre les étages

D'après Gregor Hohpe : l'architecte relie la stratégie, au penthouse, à la réalité technique, dans la salle des machines, en traduisant dans les deux sens.

Faire grandir d'autres architectes

Un architecte qui prend personnellement toutes les décisions importantes devient un goulet d'étranglement, et la capacité architecturale de l'organisation s'en va avec lui. Faire grandir les autres fait donc partie du rôle ; ce n'est pas un supplément facultatif. Il est utile de distinguer trois activités. Le mentorat partage votre expérience et vos conseils. Le coaching pose des questions qui aident la personne à trouver sa propre réponse, ce qui forge le jugement plus vite. Le parrainage met votre crédibilité au service de quelqu'un pour lui ouvrir des opportunités : le recommander pour piloter une conception, mettre son nom sur la proposition, l'inviter à la réunion de direction.

Parmi les techniques pratiques : travailler en binôme sur les conceptions et les revues, laisser des ingénieurs moins expérimentés rédiger l'ADR d'une décision et le relire avec eux, organiser des katas d'architecture (section 11.2) pour s'exercer, déléguer des décisions dans des garde-fous énoncés, et expliciter votre raisonnement à voix haute pendant les discussions de conception pour que les autres voient comment les compromis sont pesés, et pas seulement la conclusion.

De nombreuses organisations proposent désormais une filière de carrière technique parallèle au management. *Staff Engineer: Leadership beyond the management track* (2021) de Will Larson a popularisé le terme staff-plus pour désigner les contributeurs individuels de niveau staff, principal et au-delà, et a décrit quatre archétypes courants à partir d'entretiens avec des personnes occupant ces rôles. *The Staff Engineer's Path* (O'Reilly, 2022) de Tanya Reilly

organise ce travail autour de la vision d'ensemble, de l'exécution de grands projets et de la montée en compétence des ingénieurs qui vous entourent. Dans sa conférence de 2019 « Being Glue », Reilly a aussi attiré l'attention sur le travail de coordination (débloquer les gens, écrire le document manquant, repérer le vide entre les équipes) qui maintient les projets en vie mais qui est souvent peu valorisé lors des promotions ; les architectes devraient remarquer qui le fait et veiller à ce qu'il compte.

Archétypes staff-plus (d'après Will Larson)

Archétype	Rôle principal	Contribution architecturale typique
Tech lead (responsable technique)	Oriente l'approche et l'exécution d'une équipe, en partenariat avec son manager	Porte les décisions d'architecture de l'équipe et les garde alignées sur le système global
Architect (architecte)	Responsable de l'orientation, de la qualité et de l'approche d'un domaine critique	Fixe les principes, relit les conceptions et veille aux qualités clés dans plusieurs équipes
Solver (résolveur)	S'attaque à des problèmes difficiles et ambigus et trouve une voie, en passant souvent de l'un à l'autre	Débloque la migration enlisée, la crise de performances ou la frontière mal définie
Right hand (bras droit)	Démultiplie l'attention d'un dirigeant en empruntant son périmètre et son autorité	Pilote les initiatives techniques à l'échelle de l'organisation et relie la stratégie à l'exécution

Retour sur les pièges décisionnels

La section 1.6 a recensé les biais individuels qui faussent les décisions architecturales : l'ancrage, les coûts irrécupérables, la pensée de groupe, l'architecture pilotée par le CV, l'effet de mode et le biais de confirmation. Diriger des décisions de groupe ajoute ses propres pièges, qui viennent du statut, de la fatigue et de l'inertie organisationnelle plutôt que du raisonnement individuel.

Pièges de groupe et pièges organisationnels

Piège	À quoi il ressemble	Parade
Déférence envers l'avis le mieux payé	La personne la plus haut placée parle en premier et la discussion converge vers son point de vue	Demandez à la personne la plus haut placée de parler en dernier ; recueillez les avis par écrit avant la réunion
Escalade de l'engagement	Une initiative qui échoue reçoit davantage d'investissements parce qu'on y a déjà tant engagé (Barry Staw, 1976)	Fixez des critères d'arrêt et des dates de revue dès le départ ; faites la revue avec des personnes qui n'y ont pas participé
Bikeshedding	Un long débat sur des détails anodins et faciles à comprendre, tandis que la décision difficile n'obtient que quelques minutes (loi de futilité de Parkinson, 1957)	Limitez le temps selon l'importance ; renvoyez les détails de type porte à double sens à l'équipe qui en est responsable
Décision par épuisement	L'option défendue par la personne la plus endurante l'emporte une fois que tous les autres sont fatigués	Convenez à l'avance de la méthode de décision, de la personne qui décide et de l'échéance
Faux consensus	Le silence est pris pour un accord ; les objections apparaissent après la décision	Demandez explicitement les objections, nominativement si nécessaire, et consignez-les
Syndrome du « pas inventé ici »	Les solutions externes sont rejetées parce qu'elles n'ont pas été construites en interne	Comparez le développement interne et l'achat selon les mêmes critères, y compris le coût d'exploitation de ce que vous construisez
Paralysie par l'analyse	On demande toujours plus d'informations ; rien n'est décidé au dernier moment responsable	Fixez la date de décision et les éléments probants qui suffiraient ; prenez rapidement les décisions réversibles

EXEMPLE COMMENTÉ

Exemple commenté : mener une décision contestée sur un bus d'événements

Cet exemple revient sur la décision qui a introduit le courtier d'événements de la plateforme de livraison de repas, avant que l'architecture évaluée à la section 11.1 n'existe. L'équipe plateforme propose que tous les services communiquent par un cluster Kafka managé commun à toute l'entreprise. L'équipe des commandes est enthousiaste, l'équipe des paiements s'y oppose fermement, la finance s'inquiète du coût, et le CTO veut une décision d'ici un mois. Selon le processus de consultation de l'entreprise, c'est l'architecte en chef de l'équipe plateforme qui décide. Une architecte principale est chargée d'animer la démarche.

1. Reformuler la question	Kafka pour tout ? est une position. Le problème sous-jacent, révélé par deux pannes un vendredi, est que la prise de commande dépend d'appels synchrones aux notifications et à l'affectation. La question devient : comment les services doivent-ils publier les événements métier, en commençant par les événements de commande, pour que les consommateurs ne se trouvent plus sur le chemin de la commande ? La portée et les droits de décision sont écrits en tête d'un projet d'ADR.
2. Écouter d'abord en tête-à-tête	Avant toute réunion, la facilitatrice parle à chaque partie. Paiements : les données de carte doivent rester hors des nouveaux systèmes, et l'équipe ne peut pas assumer davantage d'astreinte après un incident pénible avec un courtier auto-hébergé. Commandes : a besoin de découplage pour les pics. Finance : veut plafonner les coûts de fonctionnement. Sécurité : a besoin de chiffrement et d'un contrôle d'accès par topic. L'équipe plateforme : veut un seul standard pris en charge plutôt que cinq. Chacun est consigné comme un intérêt, pas comme une position.
3. S'accorder sur les critères avant les options	Au forum consultatif d'architecture, le groupe convient de critères pondérés : garanties de livraison des messages et ordre par commande (5), charge opérationnelle pour les équipes produit (5), périmètre de conformité (4), coût de fonctionnement à trois fois le volume actuel (3), compétences (3) et réversibilité (3).
4. Générer des options, y compris celles des opposants	A : Kafka managé, obligatoire pour tous les services. B : Kafka managé pour les événements du domaine des commandes via une outbox transactionnelle, les autres équipes pouvant y adhérer plus tard. C : les services plus simples de publication-abonnement et de files d'attente du fournisseur cloud. D : conserver les appels synchrones et ajouter des nouvelles tentatives et des disjoncteurs. L'équipe des paiements apporte l'option D et l'analyse des coûts de C.

5. Réunir des éléments probants	Un spike de deux semaines exécute B et C avec une charge proche de la production : latence de bout en bout, comportement lors du basculement du courtier et coût mensuel estimé. La sécurité confirme que les événements de commande peuvent être conçus pour ne contenir aucune donnée de carte, ce qui les maintient hors du périmètre des cartes de paiement.
6. Tour de consentement	Proposition : B, avec l'équipe plateforme qui exploite le cluster en X en tant que service (X-as-a-Service). L'équipe des paiements objecte qu'elle risque encore d'être entraînée dans l'exploitation du courtier ; l'objection est levée en inscrivant la responsabilité et les horaires de support de l'équipe plateforme dans son API d'équipe. La préoccupation de la finance devient une alerte de coût et un seuil de revue. Une ingénieure de l'équipe des commandes préfère toujours C ; elle peut vivre avec B, et son raisonnement est consigné.
7. Décider, consigner et communiquer	Le décideur accepte B. L'ADR consigne les options, les scores, les résultats du spike, la dissidence et les déclencheurs de revue (plus de trois équipes adoptantes, ou des coûts supérieurs au seuil convenu, ou six mois, selon ce qui survient en premier). Le CTO et le directeur financier reçoivent une synthèse d'une page sur les bénéfices, le coût et le risque ; les équipes ont droit à une présentation commentée ; l'entrée du radar technologique passe à Trial (essai).
8. Assurer le suivi	Une vérification de compatibilité des schémas s'exécute dans le pipeline de chaque producteur, le scénario de panne est testé à nouveau et, lors de la revue à six mois, les éléments probants servent à décider s'il faut étendre, maintenir ou annuler la décision.

La facilitatrice n'a jamais plaidé pour une option ; le travail a porté sur la formulation, les intérêts, les critères, les éléments probants et une méthode de décision claire.

La dissidence a changé le résultat (périmètre réduit, responsabilité de la plateforme rendue explicite) et a été consignée au lieu d'être écartée en silence.

La décision a été rendue réversible en pratique grâce à une portée limitée et à des déclencheurs de revue convenus.

ATTENTION

Anti-patrons du leadership architectural

- Gagner les discussions grâce à l'ancienneté, ce qui coupe le flux des mauvaises nouvelles.
- Décider en privé et annoncer en public, si bien que la consultation passe pour du théâtre.
- Traiter chaque désaccord comme une résistance à gérer plutôt que comme une information à exploiter.
- Promettre aux dirigeants une certitude que le travail d'ingénierie ne peut pas garantir.
- Accaparer les décisions intéressantes, si bien que personne d'autre n'apprend à les prendre.
- Rester neutre sur des questions de sûreté, de sécurité ou d'éthique sur lesquelles l'architecte devrait tenir une ligne ferme.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi parler aux parties prenantes en tête-à-tête avant une réunion de décision ?

Pour comprendre leurs intérêts en privé, éviter de les prendre par surprise en public et faire émerger les objections tôt, quand elles peuvent encore façonner la proposition.

▼ En quoi le mentorat, le coaching et le parrainage diffèrent-ils ?

Le mentorat partage votre expérience et vos conseils, le coaching pose des questions qui aident la personne à trouver sa propre réponse, et le parrainage met votre crédibilité au service de la création d'opportunités pour elle.

▼ Que perd un architecte qui ne quitte jamais la salle des machines ?

La capacité d'influencer la stratégie : sa connaissance des risques, des contraintes et des options n'atteint jamais les personnes qui prennent les décisions d'investissement et d'organisation.

▼ Pourquoi la dissidence doit-elle être consignée dans l'enregistrement de décision ?

Pour que le raisonnement soit conservé, que les opposants sachent qu'ils ont été entendus et, si leurs craintes se réalisent, que l'organisation sache quoi réexaminer et pourquoi.

▼ À vous : L'équipe plateforme veut un cluster Kafka partagé unique ; l'équipe Paiements s'y oppose fermement ; vous n'avez d'autorité sur aucune des deux équipes, et le CTO veut une décision dans le mois. Comment géreriez-vous cela, sans imposer de résultat ?

Décidez d'abord qui tranche et comment, disons par consentement avec le CTO pour départager. Séparez ensuite les positions des intérêts : Paiements peut craindre le périmètre d'audit ou une perte de contrôle, et l'équipe plateforme veut avoir moins à exploiter. Convenez de critères objectifs, comme le coût, la latence et la conformité, et testez-les avec un petit spike. Les options pourraient inclure un cluster séparé pour Paiements. Consignez les désaccords dans l'ADR. En cas de blocage, faites remonter le sujet ensemble avec un énoncé commun des options, et non en rivaux.

SOURCES

- Gregor Hohpe: The Architect Elevator: https://architectelevator.com/
- Will Larson: Staff archetypes: https://staffeng.com/guides/staff-archetypes/
- Andrew Harmel-Law: Scaling the Practice of Architecture, Conversationally: https://martinfowler.com/articles/scaling-architecture-conversationally.html
- RFC 7282: On Consensus and Humming in the IETF: https://www.rfc-editor.org/rfc/rfc7282

SECTION 11.8 SUR 11.8 · AVANCÉ · 31 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Éthique, durabilité et analyse de rentabilité

AVANT DE LIRE

Cette section s'appuie sur :

- 1.7 Dette technique
- 2.7 Sécurité, utilisabilité et autres qualités
- 8.7 Protection de la vie privée dès la conception

Termes déjà vus qu'elle utilise : Dette technique, Accessibilité, Modélisation des menaces, Système hérité (legacy)

L'IDÉE EN BREF

Avant de payer une nouvelle toiture, un propriétaire avisé se demande ce qu'elle coûte, combien elle lui fera économiser en réparations et en chauffage, en combien de temps elle sera rentabilisée et ce qui se passera s'il attend encore un hiver. Un bon artisan répond honnêtement à ces questions. Un bon artisan refuse aussi d'installer un câblage qui pourrait provoquer un incendie, même si le client le veut moins cher, et propose une isolation qui réduit les factures d'énergie. Les architectes ont besoin de ces deux réflexes : présenter une analyse financière honnête du travail qu'ils proposent, et assumer la responsabilité des préjudices ou du gaspillage que leurs conceptions pourraient causer.

CE QU'IL FAUT SAVOIR

- Une analyse de rentabilité compare des options, y compris ne rien faire, selon leurs coûts, leurs bénéfices et leurs risques exprimés en argent, en temps et en probabilité, et en recommande une ; un financement par étapes transforme un gros pari en options plus petites.
- La valeur actuelle nette actualise le flux de trésorerie de chaque année : $VAN = \Sigma\ C_t \div (1 + r)^t$; une VAN positive signifie que l'investissement rapporte plus que le taux d'actualisation, que fixe normalement la finance.
- Le délai de récupération est le temps nécessaire pour que le flux de trésorerie cumulé devienne positif ; présentez-le avec la VAN et une analyse de sensibilité des estimations qui comptent le plus.
- La dette technique devient une analyse de rentabilité quand les intérêts sont exprimés en coût annuel (temps, retard, incidents, risque) et le principal en investissement ; les chiffres de remédiation des outils mesurent le principal, pas les intérêts.

- Le Code d'éthique de l'ACM (2018) et le BCS Code of Conduct exigent des architectes qu'ils prennent en compte l'intérêt public, évitent de nuire, évaluent les risques de façon approfondie et soient honnêtes à leur sujet.
- Les préjudices prévisibles comprennent l'atteinte à la vie privée, les failles de sécurité, les biais, l'inaccessibilité, les atteintes à la sûreté et les dark patterns manipulateurs ; plusieurs exigent un soutien architectural et sont désormais réglementés.
- La pratique du logiciel vert couvre l'efficacité carbone et énergétique, la conscience carbone, l'efficacité matérielle, la mesure et les réductions réelles ; la spécification SCI (ISO/IEC 21031:2024) mesure les émissions par unité fonctionnelle.

Pourquoi les architectes ont besoin d'analyses de rentabilité

Le travail d'architecture est en concurrence, pour l'argent et les personnes, avec tout ce qu'une organisation veut faire d'autre. Une migration, une plateforme, un programme de résilience ou un remboursement de dette produit rarement une fonctionnalité visible par le client ; il est donc facile de le reporter au profit d'un travail qui en produit. Les architectes qui obtiennent le financement de ce travail ne sont généralement pas ceux qui ont les conceptions les plus élégantes, mais ceux qui savent exprimer leur argumentaire dans les termes que les décideurs utilisent pour comparer les investissements : ce que cela coûte, ce que cela rapporte, en combien de temps, avec quelle certitude, et ce qui se passe si on ne le fait pas. Cet argumentaire structuré est une analyse de rentabilité (business case).

C'est le versant métier de tout ce que ce cours a dit sur les compromis. Une analyse de rentabilité est un enregistrement de décision pour un investissement : contexte, options, critères, conséquences et recommandation, les conséquences étant exprimées autant que possible en argent, en temps et en risque. Elle ne remplace pas le jugement architectural, et une grande partie de ce que l'architecture protège (sécurité, apprentissage, optionalité) est difficile à chiffrer. Mais mettre des chiffres honnêtes là où c'est possible, et dire clairement là où ça ne l'est pas, donne le droit d'être entendu sur le reste.

Les organisations utilisent différents formats. Le Green Book du gouvernement britannique, publié par le HM Treasury, structure les dossiers du secteur public autour de cinq questions : le volet stratégique (pourquoi un changement est nécessaire), le volet économique (quelle option offre le meilleur rapport qualité-prix), le volet commercial (peut-on se le procurer), le volet financier (est-ce abordable) et le volet managérial (peut-on le mener à bien). La plupart des entreprises utilisent quelque chose de plus léger, mais ces mêmes questions méritent une réponse pour tout investissement d'architecture important.

Options, coûts, bénéfices et risques

Toute analyse de rentabilité devrait comparer au moins trois options : ne rien faire (la référence), faire le minimum et l'option recommandée, plus toute autre solution sérieuse. Ne rien faire est rarement gratuit. Le contrat de support du système hérité augmente, le taux d'incidents croît avec le trafic, et les changements deviennent de plus en plus lents ; cette dérive est la référence par rapport à laquelle on mesure les bénéfices des autres options, et l'omettre fait paraître chaque investissement moins intéressant qu'il ne l'est.

Les coûts et les bénéfices devraient être listés par catégorie, pour qu'aucun ne soit oublié ni compté deux fois. Les risques ont explicitement leur place dans l'analyse : à la fois les risques que l'investissement réduit, que l'on peut valoriser comme des pertes attendues évitées (probabilité multipliée par impact), et les risques de l'investissement lui-même, comme un dépassement ou des perturbations pendant la migration. Le coût du retard (cost of delay) de Don Reinertsen, tiré de *The Principles of Product Development Flow* (2009), offre un éclairage supplémentaire utile : la valeur perdue chaque semaine où un travail n'est pas livré. Diviser le coût du retard par la durée (weighted shortest job first) donne une manière défendable d'ordonner des travaux concurrents.

Enfin, structurez l'investissement en une série d'étapes avec des points de décision plutôt qu'en un seul gros engagement. La migration incrémentale de la section 11.6 en est un bon exemple : chaque étape livre quelque chose et produit des éléments probants ; le financement peut donc être débloqué étape par étape, et le programme arrêté si les bénéfices n'apparaissent pas. Des jalons de ce type transforment un pari risqué en un ensemble d'options plus petites ; c'est la réflexion sur la réversibilité de la section 1.6 appliquée à l'argent.

Ce que contient une analyse de rentabilité d'architecture

Catégorie	Éléments typiques	Comment estimer
Coûts ponctuels	Temps d'ingénierie, architecture de transition, migration des données, licences, formation, fonctionnement en parallèle, mise hors service	Taille de l'équipe multipliée par la durée et le coût journalier complet ; devis des fournisseurs
Coûts récurrents	Infrastructure, licences et abonnements, support, astreinte, nouvelles compétences à conserver	Factures actuelles et modèles de capacité ; tarifs des fournisseurs au volume prévu
Gains d'efficacité	Moins d'effort par changement, délai de mise en œuvre plus court, moins de travail manuel et moins de transferts entre équipes	Volume de changements multiplié par le temps gagné par changement, d'après les données de livraison
Réduction des risques	Incidents moins nombreux ou plus courts, failles de sécurité évitées, sanctions réglementaires évitées	Perte attendue : fréquence multipliée par impact, avant et après
Revenus et options	Fonctionnalités rendues possibles, marchés qui s'ouvrent, expérimentations plus rapides	Coût du retard des opportunités rendues possibles ; souvent présenté à part, comme potentiel de gain
Coûts évités	Mises à niveau de fin de support, renouvellement du matériel, hausses des frais de licence qui ne s'appliquent plus	Contrats et feuilles de route des fournisseurs

SI VOUS DÉBUTEZ

Pas à pas : pourquoi l'argent futur compte moins

L'idée la plus difficile de cette section est l'actualisation : pourquoi l'argent qui arrive dans quelques années compte moins que l'argent d'aujourd'hui, et comment la valeur actuelle nette s'en sert pour mettre en balance un coût payé maintenant et des bénéfices qui viennent plus tard. Des chiffres simples préparent à l'exemple commenté ci-dessous.

- La migration du module de commandes de la section 11.6 coûte environ £420 000 maintenant ; ses bénéfices arrivent au cours des quatre années suivantes. Additionner simplement les livres reviendrait à considérer qu'une livre de l'année 4 vaut une livre d'aujourd'hui, ce qui n'est pas le cas.
- L'argent détenu aujourd'hui peut être mis au travail, et un bénéfice prévu peut ne jamais arriver. La finance fixe donc un taux d'actualisation, ici 10 %, qui reflète ce que l'argent coûte à l'organisation et le risque de l'investissement.
- À 10 %, £100 000 aujourd'hui deviennent £110 000 dans un an et £121 000 dans deux ans. À l'inverse, £121 000 qui arrivent dans deux ans valent £100 000 aujourd'hui.
- L'actualisation, c'est ce calcul fait à l'envers : on divise un flux de trésorerie par 1,1 une fois pour chaque année d'attente. Quatre divisions transforment un bénéfice de £190 000 en année 4 en seulement environ £129 800 aujourd'hui.
- La valeur actuelle nette additionne les flux de trésorerie actualisés de chaque année, y compris l'investissement, qui est payé maintenant et n'est donc pas actualisé. Un total positif signifie que l'investissement rapporte plus que le taux d'actualisation. Comme le coût vient en premier et les bénéfices ensuite, l'actualisation fait paraître un tel dossier moins favorable qu'un simple total.
- **La leçon :** n'additionnez jamais des livres de différentes années comme si elles étaient équivalentes. Convertissez le flux de trésorerie de chaque année en valeur d'aujourd'hui au taux fixé par la finance, puis additionnez-les. Le résultat ne vaut que ce que valent les estimations ; c'est pourquoi l'exemple commenté teste aussi des bénéfices plus faibles.

Valeur actuelle nette et délai de récupération

L'argent d'aujourd'hui vaut plus que l'argent de demain : il peut être investi, et l'argent futur est moins certain. Pour comparer des flux de trésorerie qui arrivent à des années différentes, les équipes financières convertissent chacun en sa valeur d'aujourd'hui, à l'aide d'un taux d'actualisation qui reflète le coût du capital de l'organisation et le risque de l'investissement. Le taux est normalement fixé par la direction financière, et les architectes devraient l'utiliser plutôt que d'en choisir un eux-mêmes.

La valeur actuelle nette d'un investissement est la somme de tous ses flux de trésorerie, chacun actualisé à aujourd'hui : VAN = $\Sigma\ C_t \div (1 + r)^t$, sommée sur les années t = 0 à n, où C_t est le flux de trésorerie net de l'année t (négatif pour les coûts) et r le taux d'actualisation annuel. Le flux de l'année 0 est l'investissement initial et n'est pas actualisé. Une VAN positive signifie que l'investissement rapporte plus que le taux d'actualisation ; pour choisir entre des options de risque comparable, celle qui a la VAN la plus élevée crée le plus de valeur.

Le délai de récupération est plus simple : combien de temps faut-il pour que le flux de trésorerie cumulé devienne positif ? Les dirigeants l'apprécient parce qu'il est facile à comprendre et qu'il rend compte du risque de manière sommaire (plus l'argent revient tôt, moins il y a de choses qui peuvent mal tourner), mais il ignore tout ce qui se passe après le point de récupération. Le délai de récupération actualisé fait le même calcul avec des flux de trésorerie actualisés. Présentez à la fois la VAN et le délai de récupération, et montrez leur sensibilité aux estimations qui comptent le plus.

EXEMPLE COMMENTÉ

Exemple commenté : l'analyse de rentabilité de la migration strangler fig du module de commandes

L'architecte de la plateforme de livraison de repas construit l'analyse de rentabilité de la migration du module de commandes décrite à la section 11.6. La finance fixe le taux d'actualisation à 10 %. Les bénéfices sont estimés à partir des données de livraison et d'incidents ; les revenus des nouvelles fonctionnalités sont laissés de côté et mentionnés seulement comme potentiel de gain.

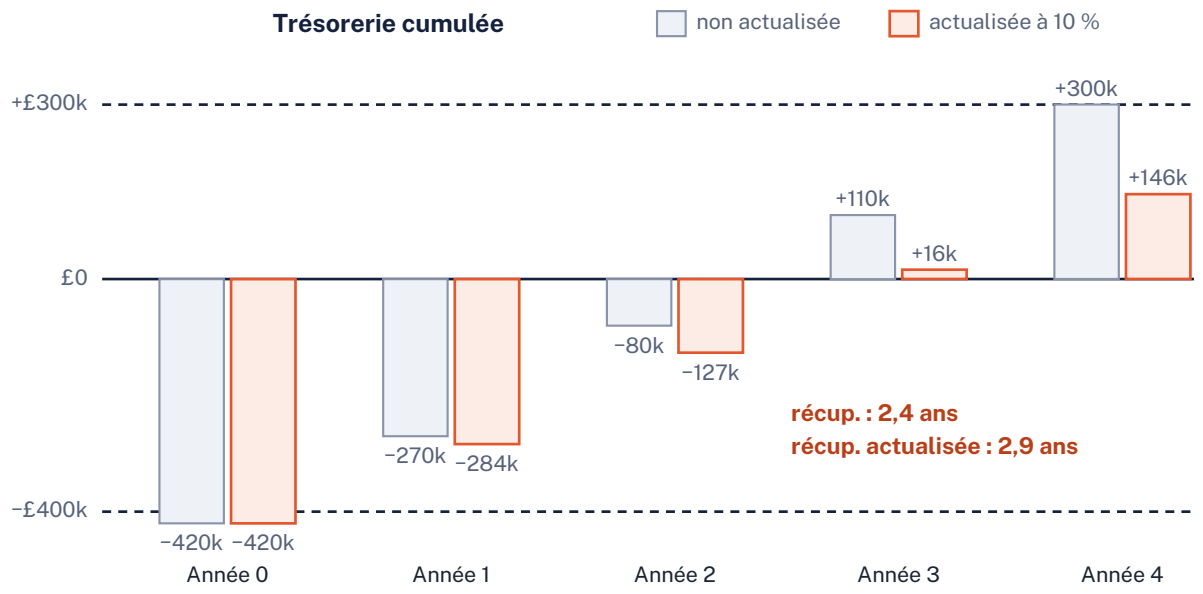
Investissement (année 0)	Six ingénieurs pendant environ cinq mois (environ 630 jours-ingénieur au même coût journalier complet de £600, soit £378 000), plus l'infrastructure de transition et le fonctionnement en parallèle (£42 000) : £420 000.
Bénéfice annuel lié à des changements plus rapides (années 2 à 4)	Environ 100 changements liés aux commandes par an, chacun économisant 2 jours-développeur de coordination, d'attente de mise en production et de reprise, à un coût journalier complet de £600 : 100 × 2 × £600 = £120 000.
Bénéfice annuel lié à la baisse des incidents de pic	Six incidents par an lors du pic du vendredi, chacun coûtant environ £20 000 en remboursements, bons d'achat et commandes perdues ; la migration devrait en supprimer les deux tiers : 4 × £20 000 = £80 000.
Surcoût net de fonctionnement	Nouveau service et courtier, moins la capacité libérée dans le monolithe : −£10 000 par an.
Flux de trésorerie nets	Année 0 : −£420 000. Année 1 (montée en charge des bénéfices pendant la migration) : +£150 000. Années 2, 3 et 4 : +£190 000 chacune.
Flux de trésorerie actualisés	$-420{,}000 + 150{,}000 \div 1.1 + 190{,}000 \div 1.1^2 + 190{,}000 \div 1.1^3 + 190{,}000 \div 1.1^4 =$ $-420{,}000 + 136{,}364 + 157{,}025 + 142{,}750 + 129{,}773$.
Valeur actuelle nette	VAN ≈ +£145 900 sur quatre ans. Avec un taux d'actualisation de 15 %, elle serait encore d'environ +£87 700.
Délai de récupération	Flux de trésorerie cumulé non actualisé : −420 000 ; −270 000 ; −80 000 ; +110 000. Délai de récupération = 2 + 80 000 ÷ 190 000 ≈ 2,4 ans. Avec les flux actualisés, le total cumulé devient positif en année 3 : 2 + 126 612 ÷ 142 750 ≈ 2,9 ans.
Sensibilité	Si les bénéfices sont inférieurs de 30 % aux estimations, la VAN tombe à environ −£23 900. Le dossier résiste aux variations du taux d'actualisation, mais il est sensible aux estimations des bénéfices.
Recommandation	Approuver les étapes 1 à 4 de la migration (£250 000), mesurer le délai de mise en œuvre des changements et les incidents liés aux commandes, et ne débloquer le financement des étapes 5 et 6 que si les économies mesurées confirment au moins 80 % de l'estimation.

Le calcul est simple ; sa valeur tient à l'explicitation des hypothèses, pour que la finance puisse les contester et que l'équipe puisse les mesurer.

L'analyse de sensibilité montre où porter l'effort : valider tôt le temps de développement économisé compte bien plus qu'affiner le taux d'actualisation.

Le financement par étapes transforme un gros pari unique en une suite d'options, chacune débloquée sur la base d'éléments probants.

Le délai de récupération en un coup d'œil



Les flux de trésorerie cumulés deviennent positifs au cours de l'année 3 ; l'actualisation retarde la récupération et réduit la valeur finale à une VAN d'environ £146 000.

```
def npv(rate, cash_flows):
    """cash_flows[0] is year 0 (the investment, usually negative)."""
    return sum(c / (1 + rate) ** t for t, c in enumerate(cash_flows))

flows = [-420_000, 150_000, 190_000, 190_000, 190_000]
print(round(npv(0.10, flows)))        # 145911
print(round(npv(0.15, flows)))        # 87663

pessimistic = [flows[0]] + [c * 0.7 for c in flows[1:]]
print(round(npv(0.10, pessimistic)))  # -23862
```

Le même calcul en quelques lignes, pratique pour tester différentes hypothèses.

La dette technique en argent et en temps

La section 1.7 a présenté la dette technique comme un principal (le coût de correction d'un problème) et des intérêts (le surcoût payé à chaque changement tant qu'il subsiste), et a estimé les intérêts d'un module enchevêtré. Une analyse de rentabilité a besoin des mêmes grandeurs sous une forme que la finance reconnaît. Les intérêts deviennent un coût annuel ; le principal devient l'investissement ; et rembourser la dette devient un investissement dont le rendement est constitué des intérêts qu'on ne paie plus, plus toute réduction des risques et toute opportunité rendue possible.

Plusieurs types d'éléments probants peuvent être convertis. Temps de développement : les jours supplémentaires par changement, tirés de comparaisons avec des changements similaires ailleurs ou des estimations des développeurs, multipliés par le volume de changements et un coût journalier complet. Retard : l'allongement des délais multiplié par le coût du retard du travail concerné. Incidents : le nombre et la durée des incidents imputables à la dette, multipliés par leur coût. Risque : la probabilité et l'impact des défaillances que la dette rend plus probables, comme une faille de sécurité via un composant qui n'est plus pris en charge. Opportunité : les initiatives métier que la dette bloque purement et simplement.

Méfiez-vous des chiffres de dette produits par les outils d'analyse statique, qui indiquent généralement un effort de remédiation estimé (le temps qu'il faudrait pour corriger chaque problème signalé). C'est une estimation du principal, pas des intérêts, et cela ne dit rien de la fréquence à laquelle le code est modifié, ni même s'il l'est. La dette dans du code rarement touché coûte peu ; les hotspots de la section 1.7 sont là où les intérêts sont réellement payés, et là où le remboursement rapporte le plus. Présenter la dette comme un portefeuille de petits investissements, chacun avec un rendement et un délai de récupération estimés, est généralement bien plus convaincant qu'un chiffre global unique de la dette totale.

La cartographie de Wardley

Les dossiers d'investissement comparent des options ; une **carte de Wardley**, imaginée par Simon Wardley, aide à voir quelles options existent. La carte s'ancre sur un utilisateur et son besoin. L'axe vertical est la chaîne de valeur, avec en haut ce que l'utilisateur voit et en dessous ce dont cela dépend. L'axe horizontal est l'évolution, de la genèse au sur-mesure, puis au produit et à la commodité, et les composants dérivent vers la droite à mesure que la concurrence les fait mûrir. Pour la plateforme de livraison de repas, le besoin est de se faire livrer un repas. L'app de commande se situe en haut ; l'optimisation de l'affectation est faite sur mesure et différenciante ; le paiement par carte est un produit ; la puissance de calcul est une commodité. Construire à la main ce qui se trouve loin à droite est un signal d'alerte. Le GOV.UK Service Manual suggère de cartographier pour savoir quoi construire et quoi acheter. Les positions sont des jugements : la valeur réside donc dans la conversation.

Construire, acheter ou une solution intermédiaire

Le conseil du Government Digital Service est un bon choix par défaut : achetez les services courants et construisez là où la capacité est au cœur de votre organisation. Cela correspond aux sous-domaines cœur et génériques de la section 5.7. La plateforme de livraison de repas achèterait les paiements et la messagerie, mais construirait l'affectation. Entre ces deux pôles se trouvent l'adoption d'un logiciel open source, le paramétrage d'un produit et le partenariat. Le GDS prévient qu'une forte personnalisation coûte cher et rend les produits plus difficiles à mettre à jour : préférez donc le paramétrage. Il prévient aussi que *vous ne pouvez pas externaliser le risque* : quelle que soit la voie choisie, il vous faut encore les personnes et les connaissances nécessaires pour la superviser. Comparez les options sur plusieurs années, avec les méthodes ci-dessus, et comptez l'intégration, l'exploitation, le temps du personnel, les hausses de prix et le coût de la sortie (section 10.8). Gardez le fournisseur derrière votre propre interface. Réexaminez la décision à mesure que les composants évoluent.

Évaluer les fournisseurs et mener une preuve de concept

Fixez les critères avant la première démonstration, sinon c'est le meilleur vendeur qui les fixera. Séparez les critères éliminatoires des critères notés, et pondérez ces derniers comme à la section 1.6. Les critères éliminatoires portent souvent sur la localisation des données, les conditions de sortie et les preuves de sécurité. Lorsqu'un fournisseur traite des données à caractère personnel, l'article 28, paragraphe 1, du RGPD britannique n'autorise que des sous-traitants présentant des garanties suffisantes. Testez ensuite l'affirmation la plus risquée avec une **preuve de concept** : une question, une durée fixe et une note de réussite convenue à l'avance. Utilisez vos propres données et votre charge de pointe, et incluez les défaillances : que fait un nouveau fournisseur de cartographie pour l'affectation lorsque son API ralentit pendant le dîner du vendredi ? Essayez au moins deux candidats si vous le pouvez. Considérez le code comme jetable, et convenez à l'avance du résultat qui mettrait fin à l'évaluation, afin que l'effort déjà consenti n'en décide pas.

Éthique : ce que demandent les codes professionnels

Les décisions architecturales façonnent ce que les systèmes font aux personnes qui les utilisent et à celles qu'ils touchent sans leur consentement ; les architectes portent donc des responsabilités éthiques aussi bien que techniques. Deux codes professionnels sont particulièrement pertinents. Le Code d'éthique et de conduite professionnelle de l'ACM (ACM Code of Ethics and Professional Conduct), mis à jour par l'Association for Computing Machinery en 2018, s'ouvre sur des principes généraux : contribuer à la société et au bien-être humain, en reconnaissant que toutes les personnes sont parties prenantes de l'informatique ; éviter de nuire ; être honnête et digne de confiance ; être équitable et agir pour ne pas discriminer ; respecter le travail nécessaire pour produire des idées nouvelles ; respecter la vie privée ; et

honorer la confidentialité. Ses responsabilités professionnelles comprennent le fait de fournir des évaluations complètes et approfondies des systèmes informatiques et de leurs impacts, y compris l'analyse des risques possibles, et de concevoir des systèmes dont la sécurité est robuste et utilisable ; ses principes de leadership demandent une attention particulière pour les systèmes qui s'intègrent à l'infrastructure de la société.

Le BCS Code of Conduct, de BCS, The Chartered Institute for IT au Royaume-Uni, s'organise autour de quatre devoirs : envers l'intérêt public (y compris la santé publique, la vie privée, la sécurité, le bien-être, l'environnement et l'égalité d'accès), envers la compétence et l'intégrité professionnelles, envers l'autorité compétente (l'employeur ou le client) et envers la profession. Le Software Engineering Code of Ethics commun à l'ACM et à l'IEEE Computer Society (1999) ajoute huit principes, à commencer par l'intérêt public, qui prime sur les intérêts du client et de l'employeur.

Pour les architectes, ces codes se traduisent par des obligations concrètes. Analysez les effets prévisibles d'une conception, et pas seulement sa justesse fonctionnelle. Informez honnêtement les décideurs des risques, y compris ceux qu'ils préféreraient ne pas entendre. Refusez de concevoir des systèmes destinés à tromper ou à nuire, et faites remonter le problème si on vous le demande. Consignez dans l'enregistrement de décision les désaccords fondés sur des motifs éthiques. Et sachez vers qui vous tourner si la remontée interne échoue : au Royaume-Uni, par exemple, le Public Interest Disclosure Act 1998 protège les travailleurs qui signalent certains types d'actes répréhensibles.

Préjudices prévisibles et réponses architecturales

Préjudice	Exemple de la livraison de repas	Réponses architecturales
Vie privée	Les adresses personnelles et l'historique des commandes des clients sont conservés indéfiniment et largement partagés avec l'analytique	Minimisation des données, durées de conservation limitées, accès fondé sur la finalité, pseudonymisation ; voir le chapitre 8 sur la protection de la vie privée dès la conception
Sécurité	Les pièces d'identité des livreurs dans un bucket accessible depuis Internet	Modélisation des menaces, moindre privilège, chiffrement, gestion des secrets, contrôles de la chaîne d'approvisionnement
Biais et résultats inéquitables	Un algorithme d'affectation qui propose systématiquement moins de courses aux livreurs de certains quartiers ou dotés de véhicules plus anciens	Critères d'équité dans les exigences, suivi des résultats par groupe, explicabilité, examen humain des décisions contestées
Accessibilité	Une étape de paiement impossible à terminer avec un lecteur d'écran	Le niveau AA des WCAG 2.2 comme contrainte, des composants accessibles dans le design system, des tests automatisés et manuels
Sûreté (safety)	Des incitations qui récompensent la vitesse des livreurs, ou des informations sur les allergènes affichées de façon incohérente	Exigences de sûreté et analyse des dangers, plafonds sur les incitations, une source de vérité unique pour les allergènes
Manipulation	Des frais de service cachés, affichés seulement à la dernière étape ; des abonnements faciles à souscrire et difficiles à résilier	Une tarification transparente dans le contrat d'API, des parcours d'inscription et de résiliation symétriques, une revue de la conception persuasive

Dark patterns et autres préjudices prévisibles

Le tableau recense les préjudices le plus souvent à la portée d'un architecte. Plusieurs sont traités en profondeur ailleurs : la vie privée et la sécurité au chapitre 8, l'accessibilité en tant que qualité à la section 2.7, et les biais des systèmes d'apprentissage automatique au chapitre 12. L'accessibilité est de plus en plus une contrainte juridique autant qu'éthique ; dans l'UE, l'acte européen sur l'accessibilité s'applique depuis le 28 juin 2025 à de nombreux services aux consommateurs, dont le commerce électronique.

La conception manipulatrice mérite une mention particulière, car elle est souvent demandée délibérément. Harry Brignull a créé le terme dark pattern (interface truquée) en 2010 pour désigner des interfaces conçues pour amener les gens par la ruse à des choix qu'ils n'auraient

pas faits autrement ; il préfère aujourd'hui parler de deceptive patterns (patrons trompeurs). Parmi les exemples familiers : la tarification au goutte-à-goutte (drip pricing, des frais révélés seulement à la fin), le confirmshaming (rendre l'option du refus honteuse), la fausse urgence et la fausse rareté, les cases de consentement précochées et les abonnements faciles à souscrire et difficiles à résilier. Les autorités ont réagi : le règlement européen sur les services numériques (Digital Services Act, 2022) interdit aux fournisseurs de plateformes en ligne de concevoir leurs interfaces en ligne de façon à tromper ou à manipuler les destinataires de leur service, et la Federal Trade Commission américaine a publié en 2022 un rapport de ses services sur les dark patterns. Beaucoup de ces patrons nécessitent un soutien architectural (une API de tarification qui masque les frais jusqu'à l'étape de paiement, un parcours de résiliation acheminé par un service de rétention) ; les architectes sont donc bien placés pour les repérer et les contester.

La manière la plus simple de rendre les préjudices visibles est de les chercher délibérément, aux mêmes moments que les risques techniques. Ajoutez une question sur les préjudices à la modélisation des menaces et aux revues de conception : qui pourrait être lésé par ce système quand il fonctionne comme prévu, quand il tombe en panne, ou quand il est détourné ? Des pratiques comme le consequence scanning, développé par le think tank britannique Doteveryone, donnent aux équipes un format court et structuré pour cette conversation pendant la planification.

Durabilité : le logiciel vert

Le logiciel ne brûle pas de carburant, mais l'électricité et le matériel qu'il utilise, si, et les décisions architecturales figent une grande partie de cette empreinte pour des années. La Green Software Foundation, lancée en 2021 sous l'égide de la Linux Foundation, publie six domaines fondamentaux de la pratique du logiciel vert : l'efficacité carbone (émettre le moins de carbone possible), l'efficacité énergétique (consommer le moins d'énergie possible), la conscience carbone (faire plus quand l'électricité est plus propre et moins quand elle est plus sale), l'efficacité matérielle (utiliser le moins de carbone incorporé possible), la mesure, et les engagements climatiques (comprendre exactement comment une réduction est obtenue).

L'informatique consciente du carbone (carbon-aware computing) exploite le fait que l'intensité carbone de l'électricité varie selon l'heure et la région, en fonction de la part de l'éolien, du solaire, du gaz et des autres sources. Le déplacement de la demande (demand shifting) déplace le travail flexible, comme les traitements par lots, l'entraînement de modèles, les sauvegardes et la génération de rapports, vers des moments ou des régions où l'électricité est plus propre ; le modelage de la demande (demand shaping) réduit ce que fait le logiciel quand elle est plus sale, par exemple en servant des médias de plus basse résolution ou en différant les recalculs non urgents. Les services de données sur le réseau électrique, comme la Carbon Intensity API pour la Grande-Bretagne, et des outils open source comme le Carbon Aware SDK

de la fondation fournissent les signaux. À titre d'illustration simple, un traitement par lots nocturne qui consomme 400 kWh émet environ 120 kg CO_2e à un niveau illustratif de 300 g par kWh en début de soirée, mais environ 48 kg s'il est déplacé à une nuit venteuse à 120 g par kWh.

Le carbone incorporé (embodied carbon), c'est-à-dire les émissions liées à la fabrication et à l'élimination du matériel, est la raison pour laquelle l'efficacité matérielle compte : une utilisation plus élevée d'un nombre réduit de machines, la réduction à zéro en période d'inactivité et l'allongement de la durée de vie des appareils (y compris les téléphones des clients, en n'exigeant pas de nouveau matériel à chaque version) le réduisent tous. La spécification Software Carbon Intensity de la fondation, publiée sous la référence ISO/IEC 21031:2024 et présentée à la section 2.7, combine les émissions opérationnelles et incorporées par unité fonctionnelle, par exemple par commande livrée, et exclut la compensation carbone pour que seules les réductions réelles améliorent le score.

Parmi les leviers architecturaux : le dimensionnement au plus juste et l'autoscaling, la suppression des environnements inactifs, la limitation de la conservation des données (les données stockées ont un coût en énergie et en matériel), des formats de données efficaces et la mise en cache, le choix de régions dont le réseau électrique est plus propre lorsque la latence et la résidence des données le permettent, et la remise en question de la nécessité d'un appel à un grand modèle pour chaque fonctionnalité. Méfiez-vous de l'effet rebond, la forme moderne de l'observation faite en 1865 par William Stanley Jevons à propos du charbon : une efficacité qui rend quelque chose moins cher peut en augmenter l'usage total ; mesurez donc les émissions totales autant que l'intensité.

Liste de contrôle : une revue d'architecture éthique et durable

À utiliser en complément des méthodes de revue technique des sections 11.1 et 11.2. Chaque point devrait produire soit une décision consignée, soit une action avec un responsable.

- [ ] **Nommer les personnes concernées**
  Listez les personnes concernées par le système, y compris celles qui ne l'utilisent jamais directement (livreurs, voisins, personnes dont les données sont traitées), ainsi que leurs intérêts.
- [ ] **Se demander comment il pourrait nuire**
  Pour le fonctionnement prévu, la panne et le détournement, demandez-vous qui pourrait être lésé et comment ; consignez les réponses avec le modèle de menaces.
- [ ] **Minimiser les données à caractère personnel**
  Vérifiez que chaque donnée à caractère personnel a une finalité, une durée de conservation et une règle d'accès, et que le système peut honorer l'effacement.
- [ ] **Vérifier l'équité des décisions automatisées**
  Identifiez les décisions prises ou classées par des algorithmes, définissez des critères d'équité et prévoyez comment les résultats seront suivis par groupe et pourront être contestés.
- [ ] **Confirmer l'accessibilité**
  Traitez le niveau AA des WCAG 2.2 (ou la norme légale applicable) comme une contrainte et confirmez comment il sera testé.
- [ ] **Rechercher la conception manipulatrice**
  Examinez les parcours de tarification, de consentement, d'inscription et de résiliation à la recherche de coûts cachés, de cases précochées, de fausse urgence et d'asymétrie.
- [ ] **Examiner les implications pour la sûreté**
  Identifiez toute conséquence physique ou sanitaire, par exemple liée aux allergènes, aux incitations ou à l'usage des véhicules, et les contrôles qui l'encadrent.
- [ ] **Estimer l'empreinte carbone**
  Estimez l'énergie, l'intensité carbone du réseau électrique et les émissions incorporées par unité fonctionnelle (SCI), et identifiez les principaux contributeurs.
- [ ] **Utiliser les options conscientes du carbone**
  Identifiez le travail qui peut être déplacé dans le temps ou vers une autre région, la capacité inactive qui peut être supprimée et les données qui peuvent être effacées plus tôt.

- [ ] **Rendre l'analyse de rentabilité honnête**
  Incluez les coûts de fonctionnement, la référence « ne rien faire » et une analyse de sensibilité ; séparez les bénéfices mesurés du potentiel de gain espéré.
- [ ] **Consigner les désaccords et les remontées**
  Consignez les objections éthiques dans l'enregistrement de décision et connaissez la voie à suivre pour faire remonter les préoccupations non résolues.
- [ ] **Prévoir la mesure**
  Choisissez des indicateurs (signalements de préjudices, défauts d'accessibilité, SCI, coût par commande) et une date pour les comparer aux attentes ci-dessus.

ATTENTION

Erreurs courantes

- Supposer que l'option « ne rien faire » ne coûte rien, ce qui fait paraître chaque investissement moins intéressant qu'il ne l'est.
- Compter deux fois le même bénéfice, par exemple comme temps de développement économisé puis comme livraison plus rapide des mêmes fonctionnalités.
- La fausse précision : une VAN à la livre près sur des estimations fiables à 30 % près.
- Omettre les coûts de fonctionnement, les coûts de transition et la mise hors service.
- Traiter l'éthique comme une case de conformité à cocher à la fin plutôt que comme une question posée pendant la conception.
- Revendiquer des réductions de carbone grâce à la compensation ou à des changements comptables plutôt qu'à des réductions réelles d'énergie et de matériel.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Pourquoi une analyse de rentabilité doit-elle inclure une option « ne rien faire » ?**

Parce que ne rien faire a ses propres coûts et risques croissants, qui forment la référence par rapport à laquelle on mesure les bénéfices de toutes les autres options.

▼ **Pourquoi un architecte devrait-il utiliser le taux d'actualisation fixé par la finance plutôt que d'en choisir un ?**

Parce qu'il reflète le coût du capital et la politique de risque de l'organisation, et qu'utiliser le même taux rend les investissements d'architecture comparables à tous les autres investissements en concurrence pour l'argent.

▼ **Qu’est-ce que le carbone incorporé ? Citez un moyen architectural de le réduire.**

Les émissions liées à la fabrication, au transport et à l’élimination du matériel ; il diminue avec une utilisation plus élevée d’un nombre réduit de machines, la réduction à zéro en période d’inactivité ou l’allongement de la durée de vie des appareils.

▼ **Quelle est la différence entre le déplacement de la demande et le modelage de la demande ?**

Le déplacement de la demande déplace le travail flexible vers des moments ou des régions où l’électricité est plus propre ; le modelage de la demande change la quantité de travail que fait le logiciel selon la propreté de l’électricité.

▼ **À vous : Une migration coûte £200 000 aujourd’hui et fait économiser £90 000 par an pendant trois ans. Le taux d’actualisation du service financier est de 10 %. Calculez la VAN et le délai de récupération. Testez ensuite des économies de £75 000 par an. Que dites-vous au conseil d’administration ?**

VAN à £90 000 : 81 818 + 74 380 + 67 618 - 200 000 = environ +£23 800. Délai de récupération : environ 2,2 ans. À £75 000, la VAN est d’environ -£13 500 : le dossier devient donc négatif. Dites au conseil d’administration qu’il dépasse le taux, mais de peu, et qu’il dépend de l’estimation des économies. Recommandez un financement par étapes : financez une première tranche, mesurez les économies réelles, puis décidez du reste. Montrez aussi l’option de ne rien faire.

SOURCES

- ACM Code of Ethics and Professional Conduct: https://www.acm.org/code-of-ethics
- BCS Code of Conduct: https://www.bcs.org/membership-and-registrations/become-a-member/bcs-code-of-conduct/
- Green Software Foundation: Principles of green software: https://learn.greensoftware.foundation/introduction
- Software Carbon Intensity (SCI) specification: https://sci.greensoftware.foundation/
- Harry Brignull: Deceptive Patterns: https://www.deceptive.design/
- Finding a path (Simon Wardley, chapter 2): https://medium.com/wardleymaps/finding-a-path-cdb1249078c0
- On being lost (Simon Wardley, chapter 1): https://medium.com/wardleymaps/on-being-lost-2ef5f05eb1ec
- The Four Stages of Evolution (wardleymaps.com): https://www.wardleymaps.com/book-passages/wardley-on-evolution

- Choosing technology: an introduction (GOV.UK Service Manual): https://www.gov.uk/service-manual/technology/choosing-technology-an-introduction
- To build or to buy, that's the technology question (GDS, 2021): https://technology.blog.gov.uk/2021/02/03/to-build-or-to-buy-thats-the-technology-question
- UK GDPR Article 28: https://www.legislation.gov.uk/eur/2016/679/article/28

Lectures complémentaires

- *Software Architecture in Practice, 4th edition*, Len Bass, Paul Clements and Rick Kazman (2021), Addison-Wesley. Le manuel de référence ; ses chapitres sur l'évaluation de l'architecture décrivent l'ATAM et sa variante légère, par les auteurs de la méthode.
- *ATAM: Method for Architecture Evaluation (CMU/SEI-2000-TR-004)*, Rick Kazman, Mark Klein and Paul Clements (2000), Software Engineering Institute, Carnegie Mellon University. Le rapport technique d'origine qui définit les étapes, les rôles et les résultats de l'ATAM, avec un cas commenté.
  https://www.sei.cmu.edu/library/atam-method-for-architecture-evaluation/
- *Building Evolutionary Architectures: Automated Software Governance, 2nd edition*, Neal Ford, Rebecca Parsons, Patrick Kua and Pramod Sadalage (2022), O'Reilly Media. La source des fonctions d'aptitude architecturales et la présentation la plus claire de l'architecture évolutive.
  https://oreilly.com/library/view/building-evolutionary-architectures/9781492097532
- *Performing a Project Premortem*, Gary Klein (2007), Harvard Business Review. Le court article qui a introduit le pre-mortem : imaginer d'abord l'échec, puis le prévenir.
  https://hbr.org/2007/09/performing-a-project-premortem
- *Scaling the Practice of Architecture, Conversationally*, Andrew Harmel-Law (2021), martinfowler.com. Décrit le processus de consultation, les forums consultatifs, les principes élaborés par les équipes et un radar technologique interne.
  https://martinfowler.com/articles/scaling-architecture-conversationally.html
- *Design It! From Programmer to Software Architect*, Michael Keeling (2017), Pragmatic Bookshelf. Un guide pratique de l'architecture comme activité d'équipe, avec de nombreuses activités d'atelier, de conception et d'évaluation pour les équipes.
- *Kill It with Fire: Manage Aging Computer Systems (and Future Proof Modern Ones)*, Marianne Bellotti (2021), No Starch Press. Un guide franc, fondé sur l'expérience, pour moderniser les systèmes hérités sans réécriture vouée à l'échec.

Glossaire

Analyse de rentabilité (business case)
Argumentaire structuré en faveur d'un investissement, qui expose le problème, les options (y compris ne rien faire), leurs coûts, leurs bénéfices et leurs risques, ainsi qu'une recommandation que les décideurs peuvent comparer à d'autres utilisations de l'argent.

API de l’équipe (team API)
Tout ce qu’une équipe offre aux autres équipes et attend d’elles : ses interfaces de code, sa documentation, ses règles de versionnage, ses façons de travailler, ses canaux de communication et la manière de lui demander du travail.

Approche architecturale
Décision de conception importante ou combinaison de décisions, comme un style, un patron ou une tactique, que l’architecture utilise pour atteindre une qualité : par exemple un courtier d’événements avec une outbox, ou une région en warm standby.

Architecture de transition
Logiciel temporaire, comme des couches de routage, des adaptateurs et des tâches de synchronisation, construit uniquement pour permettre aux anciens et aux nouveaux systèmes de coexister pendant une migration, et retiré une fois celle-ci terminée.

Architecture évolutive
Architecture qui permet un changement guidé et incrémental selon plusieurs dimensions, le guidage étant assuré par des fonctions d’aptitude (Ford, Parsons, Kua et Sadalage).

Architecture sociotechnique
Concevoir ensemble la structure du logiciel et celle des équipes, car chacune façonne l’autre. L’idée remonte aux recherches sur les systèmes sociotechniques menées au Tavistock Institute, à commencer par l’étude d’Eric Trist et Ken Bamforth sur les mines de charbon en 1951.

Ascenseur de l’architecte (architect elevator)
Image de Gregor Hohpe qui présente l’architecte comme quelqu’un qui circule entre le penthouse d’une organisation (stratégie et dirigeants) et sa salle des machines (code et infrastructure), pour relier les deux.

ATAM (Architecture Tradeoff Analysis Method)
Méthode d’évaluation fondée sur des scénarios du SEI (Kazman, Klein et Clements, 2000). En neuf étapes réparties sur deux réunions, elle relie les facteurs métier et les scénarios priorisés aux approches architecturales et produit des risques, des non-risques, des points de sensibilité, des points de compromis et des thèmes de risque.

Branch by abstraction (branchement par abstraction)
Technique pour les modifications de grande ampleur au sein d’une même base de code : placer une abstraction devant la partie à remplacer, y rediriger les appelants, construire la nouvelle implémentation derrière elle, basculer puis supprimer l’ancienne, en gardant le système livrable du début à la fin.

Budget de latence
La part d’un objectif de temps de réponse de bout en bout attribuée à chaque étape ou à chaque service sur le chemin d’une requête, afin que chaque équipe connaisse la limite qu’elle doit respecter.

Carbone incorporé (embodied carbon)
Gaz à effet de serre émis lors de la fabrication, du transport et de l'élimination du matériel, par opposition aux émissions liées à l'électricité qu'il consomme pendant son fonctionnement.

CBAM (Cost Benefit Analysis Method)
Méthode du SEI qui prolonge l'ATAM en estimant le bénéfice de chaque stratégie architecturale à partir de l'utilité qu'elle ajoute aux scénarios priorisés, puis en le divisant par son coût pour comparer le rapport valeur-coût.

Code d'éthique de l'ACM (ACM Code of Ethics)
Code d'éthique et de conduite professionnelle de l'Association for Computing Machinery, mis à jour en 2018, dont les principes comprennent la contribution à la société, le fait d'éviter de nuire, l'honnêteté, l'équité, le respect de la vie privée et la confidentialité.

Comité de revue d'architecture
Groupe permanent d'architectes expérimentés qui examine et approuve les conceptions importantes, généralement à des étapes fixées d'un projet. Il peut apporter un jugement indépendant mais devient souvent un point de passage lent.

Consensus
Règle de décision dans laquelle le groupe poursuit la discussion jusqu'à ce que tout le monde approuve activement la proposition, ou du moins la soutienne ; inclusive mais lente, et sujette à l'impasse dans les grands groupes.

Couche anticorruption
Couche de traduction, issue de la conception pilotée par le domaine, qui convertit entre le modèle d'un autre système et le vôtre, afin que les concepts et les bizarreries de l'autre modèle ne s'infiltrent pas dans votre code.

Coût du retard (cost of delay)
Valeur perdue pour chaque unité de temps pendant laquelle un travail n'est pas livré ; Don Reinertsen l'a popularisé comme moyen de prioriser le travail selon des critères économiques.

Dark launching (lancement masqué)
Appel d'un nouveau comportement côté backend en production avec du trafic réel, tout en masquant ses résultats aux utilisateurs, afin de tester les performances et l'exactitude avant une mise en production visible.

Dark pattern (interface truquée)
Interface utilisateur conçue pour tromper les personnes ou faire pression sur elles afin qu'elles fassent des choix qu'elles ne feraient pas autrement, comme des coûts cachés ou des abonnements difficiles à résilier ; Harry Brignull a créé le terme en 2010 et parle aujourd'hui de deceptive patterns (patrons trompeurs).

Délai de récupération (payback period)
Le temps nécessaire pour que les bénéfices cumulés remboursent le coût d'un investissement ; le délai de récupération actualisé fait de même avec des flux de trésorerie actualisés.

Disagree and commit (exprimer son désaccord, puis s'engager)
Norme selon laquelle chacun exprime ouvertement son désaccord avant une décision, puis s'engage pleinement à la mettre en œuvre une fois qu'elle est prise.

Droits de décision
Énoncé explicite de qui peut prendre quelles décisions, qui doit être consulté et qui doit être informé ; les fixer avant un débat évite la plupart des conflits sur le processus.

Équipe facilitatrice (enabling team)
Équipe de spécialistes qui aide d'autres équipes à acquérir une capacité qui leur manque, comme l'automatisation des tests ou l'observabilité, pendant une durée limitée puis passe à autre chose, plutôt que de faire le travail à leur place.

Équipe orientée flux (stream-aligned team)
Équipe pérenne et pluridisciplinaire, alignée sur un flux de changement unique, comme un produit, un parcours utilisateur ou une capacité métier, capable de construire, d'exploiter et de modifier sa partie de bout en bout. La plupart des équipes d'une organisation devraient être de ce type.

Équipe plateforme
Équipe qui fournit des services, des outils et une infrastructure internes sous la forme d'un produit en libre-service, afin que les équipes orientées flux puissent livrer sans expertise approfondie de la technologie sous-jacente.

Équipe sous-système compliqué
Équipe responsable d'une partie du système qui exige des connaissances spécialisées rares, comme un algorithme d'optimisation ou un codec vidéo, afin que les équipes orientées flux puissent l'utiliser sans la maîtriser.

Évaluation de l'architecture
Examen structuré d'une architecture, avant ou après sa construction, pour déterminer si ses décisions permettront au système d'atteindre ses objectifs métier et ses exigences de qualité, et où elles mettent ces objectifs en danger.

Exécution en parallèle (parallel run)
Faire tourner l'ancienne et la nouvelle implémentation côte à côte sur les mêmes entrées et comparer leurs résultats, l'ancienne restant celle qui fait foi, afin de prouver que la nouvelle est équivalente avant de basculer.

Expérience de chaos
Test maîtrisé qui injecte une défaillance réelle, comme l'arrêt brutal d'une instance ou le ralentissement d'une dépendance, pour vérifier une hypothèse énoncée sur la manière dont le système conserve son état stable.

Facteurs métier (business drivers)
Objectifs, contraintes et contexte métier qui expliquent pourquoi un système existe et ce que signifie la réussite, comme des objectifs de croissance, la réglementation ou des contrats. Dans l'ATAM, ils sont présentés en premier afin que chaque constat puisse leur être rattaché.

Fonction d'aptitude atomique
Fonction d'aptitude qui vérifie une seule caractéristique d'architecture dans un seul contexte, comme un test qui interdit les imports entre deux paquets.

Fonction d'aptitude continue
Fonction d'aptitude qui s'exécute en permanence sur le système en production, comme la supervision des SLO ou les transactions synthétiques, plutôt que d'être déclenchée par un build ou un événement.

Fonction d'aptitude holistique
Fonction d'aptitude qui vérifie plusieurs caractéristiques ensemble dans un contexte commun, comme un test de charge qui mesure la latence pendant que la sécurité et la journalisation sont activées.

Forum consultatif d'architecture
Réunion régulière et ouverte où sont discutées les décisions proposées, les spikes et les avis recueillis. Elle apporte des conseils et de la visibilité mais ne décide pas elle-même.

Gouvernance de l'architecture
Pratiques par lesquelles une organisation s'assure que les décisions techniques importantes sont bien prises, cohérentes avec ses objectifs et réellement appliquées, y compris qui décide, comment et avec quels contrôles.

Informatique consciente du carbone (carbon-aware computing)
Exécution des logiciels de façon à en faire davantage quand l'électricité est plus propre et moins quand elle est plus polluante, en déplaçant le travail dans le temps ou dans l'espace ou en modulant la demande.

Interception d'événements
Patron de remplacement de système hérité qui intercepte les requêtes ou événements modifiant l'état sur leur chemin vers un système hérité et en transmet certains à un nouveau composant, en plus ou à la place.

Kata d’architecture
Exercice pratique créé par Ted Neward dans lequel de petites équipes conçoivent une architecture à partir d’un énoncé court et délibérément incomplet, puis la présentent et la défendent devant leurs pairs.

Les sept R (Seven Rs)
Les stratégies de migration qu’AWS recense pour faire passer des applications dans le cloud : retire, retain, rehost, relocate, repurchase, replatform, et refactor ou re-architect. Elles sont issues des cinq R proposés plus tôt par Gartner.

Mode d’interaction
Dans Team Topologies, l’une des trois manières délibérées dont deux équipes travaillent ensemble : collaboration, X en tant que service ou facilitation. Les modes sont appelés à changer à mesure que les frontières sont découvertes et se stabilisent.

Non-risque
Décision jugée saine pour un scénario donné, consignée avec les hypothèses dont elle dépend, afin de pouvoir la réexaminer si ces hypothèses changent.

Paved road (golden path, chemin balisé)
Manière par défaut, prise en charge et bien documentée, de construire et d’exécuter des logiciels, par exemple un modèle de service où la journalisation, la sécurité et le déploiement sont déjà configurés, dont les équipes peuvent s’écarter lorsqu’elles ont une bonne raison.

Plan de fracture
Ligne de séparation naturelle le long de laquelle un système logiciel peut être découpé afin que chaque partie convienne à une équipe, par exemple un contexte délimité, une frontière réglementaire, un rythme de changement différent ou un profil de risque différent.

Point de sensibilité
Propriété d’un ou de plusieurs composants ou connexions qui est déterminante pour obtenir une réponse de qualité donnée, comme le nombre de sauts synchrones sur le chemin de la commande.

Point de validation (gate)
Point de contrôle où le travail s’arrête jusqu’à ce que quelqu’un l’approuve, comme la validation d’un comité de revue avant le build ou la mise en production.

Pre-mortem
Technique de Gary Klein (2007) dans laquelle une équipe imagine que son projet a déjà échoué et où chaque personne écrit pourquoi, ce qui fait émerger des risques que l’on hésite sinon à soulever.

Principe d’architecture
Règle durable qui guide les décisions d’architecture, rédigée dans le style TOGAF avec un nom, un énoncé, une justification et ses implications.

Prise de décision par consentement
Règle de décision dans laquelle une proposition est adoptée sauf si quelqu'un soulève une objection motivée selon laquelle elle causerait un tort ou ferait reculer le groupe ; elle demande si chacun peut vivre avec une proposition, et non si chacun la préfère.

Radar technologique
Carte des technologies et des techniques, actualisée périodiquement, regroupées en quadrants et placées sur des anneaux comme Adopt, Trial, Assess et Hold (renommé Caution dans les éditions récentes de Thoughtworks), popularisée par Thoughtworks.

Réécriture big bang
Remplacement d'un système en en construisant un nouveau à partir de zéro et en basculant d'un seul coup ; largement considéré comme très risqué, car la valeur n'arrive qu'à la fin et le comportement caché se perd facilement.

Registre des risques
Liste tenue à jour des risques identifiés, avec leur probabilité, leur impact, leur responsable, leur atténuation et leur statut, afin que les risques soient suivis jusqu'à leur clôture au lieu d'être oubliés après une réunion.

Revue de conception (revue de RFC)
Revue écrite d'un document de conception proposé ou d'une demande de commentaires (request for comments), dans laquelle les personnes concernées commentent de manière asynchrone avant que la conception ne soit approuvée et ses décisions consignées.

Risk-storming
Technique collaborative de Simon Brown dans laquelle les participants marquent en silence les risques sur des diagrammes d'architecture avec des notes adhésives, notées selon la probabilité et l'impact, puis rapprochent leurs points de vue et s'accordent sur les priorités et les atténuations.

Risque (dans l'évaluation d'architecture)
Décision architecturale, ou absence de décision, susceptible d'empêcher le système de satisfaire une exigence de qualité ou un objectif métier important.

Sécurité psychologique
Conviction partagée au sein d'une équipe qu'il est sans danger de prendre des risques interpersonnels, comme reconnaître ses erreurs ou contredire un collègue plus haut placé ; étudiée par Amy Edmondson depuis 1999.

Software Carbon Intensity (SCI)
La spécification de la Green Software Foundation, publiée sous la référence ISO/IEC 21031:2024, qui évalue un logiciel par la formule ((E × I) + M) par R : l'énergie multipliée par l'intensité carbone du réseau électrique, plus les émissions incorporées, par unité fonctionnelle.

Staff-plus engineer
Contributeur individuel senior (staff, principal ou distinguished engineer) qui exerce un leadership technique à travers plusieurs équipes sans encadrer de personnes ; terme popularisé par Will Larson.

Strangler fig (figuier étrangleur)
Approche de modernisation incrémentale, nommée par Martin Fowler, dans laquelle de nouveaux composants reprennent peu à peu le comportement d'un système hérité derrière une couche d'interception, jusqu'à ce que l'ancien système puisse être arrêté.

Système hérité (legacy)
Système existant qui apporte encore de la valeur mais qui est devenu difficile, risqué ou coûteux à modifier, souvent en raison de son âge, de l'absence de tests, de compétences rares ou d'une technologie qui n'est plus prise en charge.

TARA
La Tiny Architectural Review Approach, méthode légère d'Eoin Woods pour examiner un système existant en combinant le jugement d'experts et l'analyse automatisée de son code et de ses données d'exécution.

Taux d'actualisation
Taux annuel utilisé pour convertir de l'argent futur en valeur actuelle, qui reflète le coût du capital et le risque ; généralement fixé par le service financier.

Team Topologies
Approche d'organisation des équipes logicielles pour un flux rapide, présentée par Matthew Skelton et Manuel Pais dans le livre du même nom (2019 ; deuxième édition en 2025) : quatre types d'équipes, trois modes d'interaction, et la charge cognitive de l'équipe comme contrainte principale.

Test de caractérisation
Test qui enregistre ce que fait réellement le code existant, plutôt que ce qu'il devrait faire, afin que les modifications puissent être vérifiées par rapport au comportement actuel. Le terme vient de Michael Feathers.

Thème de risque
Groupe de risques liés qui partagent une cause commune, rattaché aux facteurs métier qu'il menace. Les thèmes de risque sont généralement le résultat d'une évaluation le plus utile pour les managers.

Valeur actuelle nette (VAN, NPV)
La somme des flux de trésorerie futurs d'un investissement, chacun actualisé à sa valeur d'aujourd'hui, moins le coût initial ; une VAN positive signifie que l'investissement rapporte plus que le taux d'actualisation.

X en tant que service (X-as-a-Service)

Mode d’interaction dans lequel une équipe consomme ce qu’une autre équipe fournit, au travers d’une interface claire et documentée et avec peu de collaboration au quotidien.