

BE SOFTWARE ARCHITECTURE WISE · CHAPITRE 1

Qu’est-ce que l’architecture logicielle ?

Ce chapitre explique ce qu’est l’architecture logicielle et pourquoi elle compte. Il compare les définitions de référence, distingue l’architecture de la conception, montre à travers des incidents réels comment les décisions structurelles déterminent le coût, la qualité et les risques, et décrit ce que font les architectes. Il aborde ensuite les parties prenantes et leurs préoccupations, les décisions et les compromis, la dette technique et l’érosion, et la place de l’architecture dans la livraison itérative moderne. Il s’adresse aux développeurs qui évoluent vers des responsabilités d’architecture et à toute personne qui travaille avec des architectes, et il ne suppose que les éléments de base du chapitre 0.

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

EN UN COUP D'ŒIL

Vous apprendrez à

- Expliquer les définitions de l'architecture logicielle données par l'ISO/IEC/IEEE 42010:2022 et par le SEI, ainsi que les idées qu'elles partagent.
- Distinguer les structures de modules, de composants et connecteurs et d'allocation, et distinguer une architecture de sa description.
- Classer des décisions comme relevant de l'architecture ou de la conception d'après leur réversibilité, leur portée et leur effet sur les attributs de qualité.
- Analyser comment des décisions structurelles ont contribué à des incidents réels comme ceux de Knight Capital et de HealthCare.gov.
- Décrire les responsabilités, les compétences et les types d'architectes, et la façon dont les architectes restent dans la pratique.
- Identifier les parties prenantes et leurs préoccupations, et rendre explicites les préoccupations contradictoires.
- Évaluer les décisions architecturales comme des compromis, en tenant compte de la réversibilité et du moment opportun.
- Expliquer la dette technique, l'érosion et la dérive, et comment les détecter et les gérer tout au long du cycle de vie.

Avant de commencer

Le chapitre 0, ou une connaissance pratique des composants, des bases de données, des API, des réseaux et du cloud.

Durée

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

Termes clés

Architecture logicielle · Description d'architecture · Structure · Architecturalement significatif · Attribut de qualité · Coût du changement · Partie prenante · Préoccupation · Compromis · Réversibilité · Dette technique · Érosion de l'architecture · Fonction d'aptitude (fitness function) · Loi de Conway

Pourquoi ce chapitre vient en premier

Tout système logiciel a une architecture, que quelqu'un l'ait conçue ou non. C'est l'ensemble des structures fondamentales qui composent le système (ses parties principales, la façon dont elles sont reliées et leurs propriétés) et des décisions importantes qui les ont produites. Ces structures déterminent bien plus de choses que la plupart des gens ne l'imaginent : la rapidité et la fiabilité du système, son coût d'exploitation, le nombre d'équipes qui peuvent y travailler en même temps et, surtout, le coût de sa modification l'an prochain. Les fonctionnalités sont ce que fait un système ; l'architecture détermine en grande partie la qualité avec laquelle il le fait et ce qu'il en coûte de faire autre chose.

Le chapitre 0 a présenté les éléments de base : programmes, stockages de données, réseaux, API, serveurs et cloud. Ce chapitre prend du recul et se demande ce que signifie agencer ces éléments de façon délibérée, qui le fait et pourquoi cela en vaut la peine. Il vous donne le vocabulaire dont dépend le reste du cours, et une manière de penser applicable à n'importe quel système, d'une petite application de réservation à une plateforme nationale de paiement.

Ce qu'est l'architecture, et ce qu'elle n'est pas

Les deux premières sections portent sur les définitions, car des définitions confuses mènent à un travail confus. La section 1.1 compare les définitions qui comptent : la norme internationale ISO/IEC/IEEE 42010:2022, qui parle des concepts ou propriétés fondamentaux d'un système dans son environnement et des principes qui régissent son évolution ; la définition du Software Engineering Institute, qui parle des structures nécessaires pour raisonner sur un système, faites d'éléments, de relations et de leurs propriétés ; et les définitions des praticiens, qui insistent sur la compréhension partagée et sur les décisions coûteuses à changer. Elle montre ce qu'elles ont en commun et présente trois types de structures que possède tout système : la façon dont son code est découpé, ce qui s'exécute et communique avec quoi, et la façon dont le logiciel se répartit sur l'infrastructure et les équipes.

La section 1.2 s'attaque ensuite à la confusion la plus courante : où s'arrête l'architecture et où commence la conception. La réponse est qu'il n'y a pas de mur, seulement un continuum ordonné par l'importance. Une décision est architecturalement significative lorsqu'elle est coûteuse à annuler, large dans sa portée ou décisive pour une qualité clé ; c'est pourquoi une modeste convention, comme la façon de stocker les montants, peut être plus architecturale qu'une grande boîte sur un diagramme. La section distingue aussi les niveaux auxquels le mot est employé : architecture d'entreprise, de solution et logicielle, les architectures des données, de la sécurité et de l'infrastructure les traversant toutes.

Pourquoi elle compte et qui s'en charge

La section 1.3 explique pourquoi s'en soucier. Elle montre comment des attributs de qualité comme la disponibilité et les performances naissent de la structure plutôt que de lignes de code isolées, comment le coût du changement augmente à mesure que l'on construit sur une décision, et comment l'architecture façonne les risques, l'organisation des équipes (à travers la loi de Conway), la communication et l'estimation. Trois cas réels, racontés à partir de sources primaires, ancrent l'argumentation : la perte de trading de Knight Capital en août 2012, le lancement chaotique de HealthCare.gov en octobre 2013, et la décision prise en 2023 par Amazon Prime Video de faire passer un service de supervision d'étapes serverless distribuées à un processus unique, ce qui aurait réduit son coût de plus de 90 % et montre que l'architecture est une affaire d'adéquation, pas de mode.

La section 1.4 s'intéresse aux personnes. Elle décrit ce que font réellement les architectes, en s'appuyant sur les huit attentes formulées par Mark Richards et Neal Ford et sur les tâches du programme iSAQB Foundation ; les différents types d'architectes ; pourquoi l'architecture est un rôle avant d'être un intitulé de poste ; pourquoi les architectes accordent de la valeur à l'étendue des connaissances ; et le débat de longue date sur la question de savoir si les architectes doivent encore écrire du code.

La section 1.5 élargit l'angle aux parties prenantes (toutes les personnes qui ont un intérêt dans le système, des utilisateurs et développeurs aux équipes d'exploitation, en passant par les délégués à la protection des données et les régulateurs) et à leurs préoccupations, et montre comment les préoccupations contradictoires sont rendues explicites plutôt que tranchées par celui qui crie le plus fort. La section 1.6 passe aux décisions et aux compromis : l'anatomie d'une bonne décision, la différence entre les décisions facilement réversibles et celles qui ne le sont pas, l'idée de décider au dernier moment responsable, et les pièges cognitifs qui égarent des personnes compétentes.

L'architecture dans le temps

Les deux dernières sections portent sur le temps. La section 1.7 explique la dette technique au sens où Ward Cunningham l'entendait à l'origine en 1992, ce qui la distingue d'un simple désordre, et comment les architectures s'érodent et dérivent de la forme voulue ; elle montre comment détecter les dégâts et comment les expliquer en termes métier. La section 1.8 situe l'architecture dans le cycle de livraison : le vieux débat entre la conception complète en amont (big design up front) et la conception purement émergente, et le juste milieu pragmatique d'une architecture guidée par les risques, continue et évolutive, dans laquelle une équipe fait juste assez d'architecture, au bon moment, et vérifie en permanence qu'elle convient toujours.

L'idée à retenir

Si vous ne retenez qu'une chose de ce chapitre, que ce soit celle-ci : l'architecture est l'ensemble des décisions sur lesquelles il est coûteux de se tromper, et le travail de l'architecte consiste à identifier ces décisions, à les prendre délibérément avec les bonnes personnes, à en consigner les raisons et à les réexaminer régulièrement, tout en rendant le plus grand nombre possible d'autres décisions peu coûteuses à changer. Tout le reste de ce cours, des attributs de qualité au chapitre 2 à la conception de systèmes intégrant l'IA au chapitre 12, est une application de cette idée.

SECTION 1.1 SUR 1.8 · DÉBUTANT · 21 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Ce qu'est l'architecture logicielle : les définitions de référence et leurs points communs

L'IDÉE EN BREF

Pensez à un bâtiment hospitalier. Son architecture n'est pas le plan punaisé au mur ; c'est la façon dont le bâtiment est réellement agencé : où se trouvent les services, quels couloirs les relient, où passent les ascenseurs, quelles portes doivent rester verrouillées, et les règles que tout le monde suit quand on l'agrandit. Certains choix, comme l'emplacement des ascenseurs, coûteraient une fortune à modifier ; la couleur des rideaux, non. L'architecture logicielle, c'est la même idée appliquée à un système logiciel : ses parties principales, la façon dont elles sont reliées, leurs propriétés, et les décisions coûteuses à changer qui les sous-tendent.

CE QU'IL FAUT SAVOIR

- La norme ISO/IEC/IEEE 42010:2022 définit l'architecture comme les concepts ou propriétés fondamentaux d'une entité dans son environnement, ainsi que les principes qui régissent sa réalisation et son évolution.
- Bass, Clements et Kazman définissent l'architecture logicielle comme l'ensemble des structures nécessaires pour raisonner sur un système : les éléments logiciels, les relations entre eux et les propriétés des uns et des autres.
- Les définitions des praticiens ajoutent que l'architecture est la compréhension partagée des développeurs experts (Ralph Johnson) et l'ensemble des décisions importantes, leur importance se mesurant au coût du changement (Grady Booch).
- Les définitions partagent sept idées : structures, éléments, relations, propriétés, décisions, importance et environnement.
- Les systèmes ont trois types de structures : les structures de modules (le code), les structures de composants et connecteurs (l'exécution) et les structures d'allocation (la correspondance avec l'infrastructure, les équipes et les fichiers).
- Tout système a une architecture, qu'elle ait été conçue délibérément ou non, et qu'elle soit écrite ou non ; une description d'architecture est un produit de travail distinct qui l'exprime.

Pourquoi il faut définir le mot

Demandez à dix ingénieurs expérimentés ce qu'est l'architecture logicielle et vous entendrez dix réponses différentes : le diagramme de boîtes et de flèches affiché au mur, le choix du fournisseur de cloud, le schéma de la base de données, le découpage en services, les règles

indiquant quel code peut appeler quel autre, ou simplement « les décisions difficiles ». Chaque réponse saisit quelque chose de réel, et chacune, prise isolément, passe à côté de quelque chose. Un cours qui entend enseigner l'architecture doit donc commencer par préciser de quoi il parle.

Cela compte pour des raisons pratiques. Si une équipe pense que l'architecture, c'est « le diagramme », elle cessera d'y prêter attention une fois le diagramme dessiné, alors même que le système continue de changer en dessous. Si elle pense que l'architecture, c'est « la pile technologique », elle débattra des frameworks en ignorant la façon dont les responsabilités sont réparties et dont circulent les données. Une bonne définition vous dit quoi regarder, quoi protéger et de quoi discuter avec les autres.

Les définitions ci-dessous proviennent d'une norme internationale, du Software Engineering Institute (SEI) de l'université Carnegie Mellon, et de praticiens. Elles ont été écrites dans des buts différents, mais elles s'accordent plus qu'elles ne divergent, et leurs points d'accord forment le cœur de la discipline.

La définition normative : ISO/IEC/IEEE 42010:2022

ISO/IEC/IEEE 42010 est la norme internationale de description des architectures. Son édition actuelle, publiée en novembre 2022, a remplacé l'édition de 2011. Elle définit l'architecture, en paraphrasant de près, comme les concepts ou propriétés fondamentaux d'une entité dans son environnement, ainsi que les principes qui régissent la réalisation et l'évolution de cette entité et ses processus de cycle de vie associés.

Trois caractéristiques de cette définition méritent l'attention. D'abord, elle parle d'une *entité*, et pas seulement d'un système logiciel. L'édition 2022 a délibérément élargi le champ pour que les mêmes idées s'appliquent aux logiciels, aux systèmes de systèmes, aux produits, aux lignes de produits et aux entreprises entières. Ensuite, elle place l'entité *dans son environnement* : on ne peut pas comprendre une architecture sans les utilisateurs, les autres systèmes, l'organisation, les lois et les budgets qui l'entourent. Enfin, elle inclut des *principes directeurs* sur la façon dont l'entité est construite et dont elle change. L'architecture n'est pas seulement un instantané de la structure ; elle comprend les règles qui maintiennent la cohérence de la structure à mesure que le système évolue, comme « chaque service possède ses propres données » ou « le code du domaine ne dépend jamais du framework web ».

La norme trace aussi une frontière nette entre une architecture et une description d'architecture. L'architecture est l'ensemble des propriétés fondamentales que le système possède réellement. La description est un produit de travail (documents, modèles, diagrammes, enregistrements de décisions) qui l'exprime. ISO/IEC/IEEE 42010 est avant tout une norme sur les descriptions : elle indique ce qu'une bonne description doit contenir, comme les parties

prenantes identifiées, leurs préoccupations et les vues qui répondent à ces préoccupations. Elle ne vous dit pas comment concevoir une architecture. La section 1.5 revient sur les parties prenantes et les préoccupations, et le chapitre 4 traite des vues.

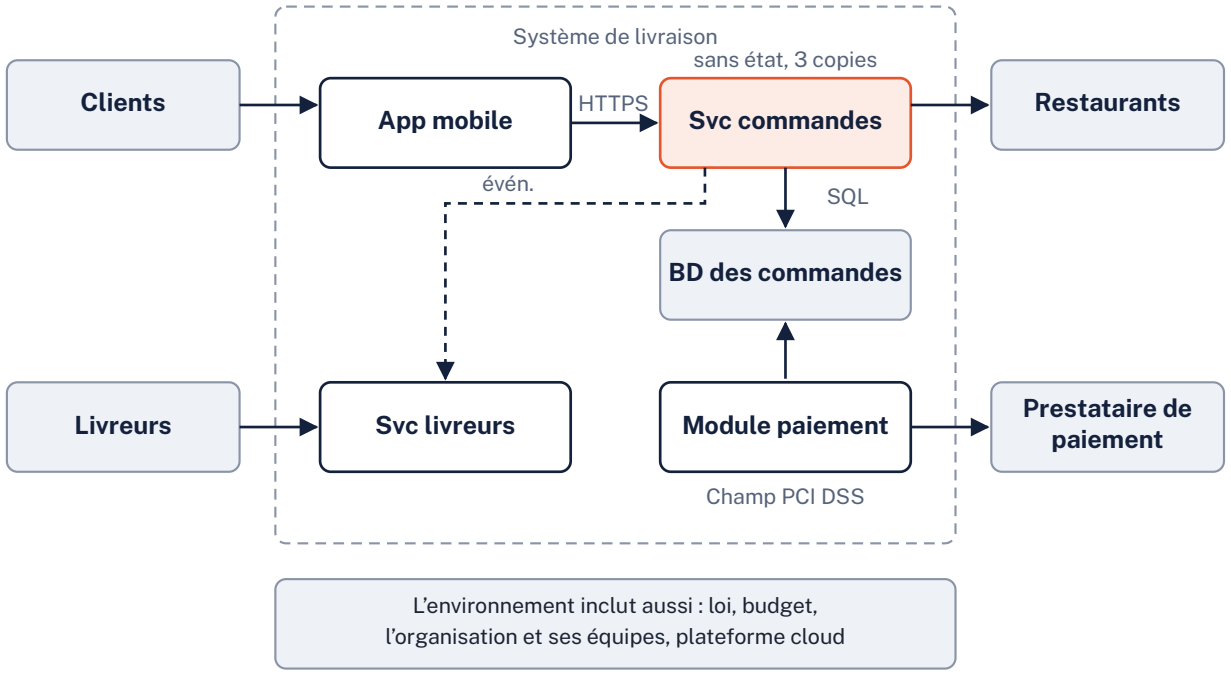
La définition du SEI : structures, éléments, relations et propriétés

La définition la plus enseignée vient de *Software Architecture in Practice* de Len Bass, Paul Clements et Rick Kazman (4e édition, Addison-Wesley, 2021), dont les auteurs ont travaillé au SEI. Selon eux, en paraphrasant : l'architecture logicielle d'un système est l'ensemble des structures nécessaires pour raisonner sur le système, et ces structures se composent d'éléments logiciels, des relations entre eux et des propriétés des uns et des autres.

Chaque partie de cette phrase a son utilité. **Structures**, au pluriel : un système possède plusieurs structures à la fois (ses modules de code, ses processus en cours d'exécution, son déploiement sur des machines) et aucune n'est à elle seule « l'architecture ». **Nécessaires pour raisonner sur le système** : pour savoir si quelque chose est architectural, on se demande si l'on en a besoin pour répondre à des questions importantes, comme « Cela résistera-t-il au pic de charge ? » ou « Deux équipes peuvent-elles y travailler indépendamment ? ». Les **éléments** sont les parties : modules, services, bases de données, files d'attente. Les **relations** sont la façon dont ils sont reliés : *utilise*, *appelle*, *publie vers*, *est déployé sur*. Les **propriétés** sont ce qui fait que les éléments et les relations se comportent comme ils le font : un service est sans état, un appel est synchrone avec un délai d'expiration de deux secondes, une base de données contient des données personnelles et doit rester dans l'UE.

La référence au raisonnement explique aussi ce qui n'est *pas* de l'architecture. Le nom d'une variable locale, l'algorithme interne d'une routine de tri ou la disposition d'un écran de paramètres n'aident généralement pas à raisonner sur le système dans son ensemble ; ils relèvent donc de la conception détaillée plutôt que de l'architecture. La section 1.2 examine où passe cette frontière.

Éléments, relations, propriétés et environnement



Des éléments (boîtes) reliés par des relations (flèches), avec leurs propriétés notées à côté, dans une frontière de système située dans un environnement plus large.

Les définitions des praticiens : compréhension partagée, ce qui est important et décisions

Les praticiens ont proposé des définitions plus courtes qui mettent l'accent sur les personnes et les décisions plutôt que sur les structures. Martin Fowler en a popularisé deux, dues à Ralph Johnson (l'un des quatre auteurs du livre *Design Patterns* de 1994). La première veut que l'architecture soit la compréhension partagée qu'ont les développeurs experts de la façon dont le système est conçu. La seconde, issue de la même discussion, est délibérément ouverte : «L'architecture, c'est ce qui est important. Quoi que ce soit.» Johnson a aussi suggéré que l'architecture est l'ensemble des décisions que l'on aimerait réussir tôt dans un projet, non parce qu'on a plus de chances de les réussir tôt, mais parce qu'il est si coûteux de se tromper.

Grady Booch, l'un des créateurs d'UML, l'a formulé en termes économiques : l'architecture est constituée des décisions de conception importantes qui façonnent un système, leur importance se mesurant au coût du changement. Cela transforme une idée abstraite en critère utilisable. Le choix de la couleur d'un bouton est facile à annuler. Le choix entre une base de données relationnelle et une base orientée documents, ou entre une seule unité déployable et quarante,

est coûteux à annuler une fois que les données, le code et les équipes se sont développés autour. Booch a aussi fait remarquer que toute architecture est conception, mais que toute conception n'est pas architecture.

Ces définitions ajoutent ce que les définitions structurelles laissent implicite. L'architecture vit en partie dans la tête des gens ; un système dont les ingénieurs seniors partagent une image claire de son fonctionnement a une architecture plus cohérente qu'un système où chacun a une image différente, même si aucun des deux n'a de diagramme. Et l'architecture est façonnée par des décisions, dont beaucoup impliquent un compromis : choisir une option, c'est renoncer aux avantages d'une autre. La section 1.6 examine en détail les décisions et les compromis.

Les définitions comparées

Source	Idée centrale (paraphrasée)	Ce qu'elle met en avant	Surtout utile pour
ISO/IEC/IEEE 42010:2022	Les concepts ou propriétés fondamentaux d'une entité dans son environnement, ainsi que les principes qui régissent sa réalisation et son évolution	Environnement, principes, évolution ; l'architecture distincte de sa description	Les contrats, le travail de normalisation, la description cohérente des architectures dans toute une organisation
Bass, Clements et Kazman (2021)	L'ensemble des structures nécessaires pour raisonner sur le système : éléments, relations et propriétés des uns et des autres	Structures multiples ; raisonnement sur les qualités	L'analyse et la conception : se demander si une structure atteindra ses objectifs
Ralph Johnson, via Martin Fowler	La compréhension partagée des développeurs experts ; ce qui est important, quoi que ce soit	Les personnes, la communication, le jugement sur ce qui est important	La pratique d'équipe : entretenir un modèle mental commun
Grady Booch	Les décisions de conception importantes qui façonnent un système, leur importance se mesurant au coût du changement	Les décisions et leur dimension économique	Décider où consacrer l'effort d'analyse

Ce que les définitions ont en commun

Lues côte à côte, les définitions convergent vers sept idées, qui forment ensemble une définition de travail pour la suite de ce cours.

Sept idées partagées

1. **Structures**
   Un système possède plusieurs structures à la fois, chacune le montrant sous un angle différent. Aucun diagramme ne représente à lui seul toute l'architecture.

2. **Éléments**
   Les parties qui composent ces structures : modules, composants, services, stockages de données, files d'attente, serveurs.

3. **Relations**
   La façon dont les éléments sont reliés et dépendent les uns des autres : appels, messages, données partagées, déploiement, propriété.

4. **Propriétés**
   Les caractéristiques visibles de l'extérieur des éléments et des relations, comme le fait d'être sans état, chiffré, asynchrone ou répliqué, qui déterminent ensemble les attributs de qualité du système.

5. **Décisions**
   Les choix qui ont produit les structures, avec les raisons et les alternatives écartées qui les sous-tendent. Sans les raisons, d'autres personnes défont plus tard de bonnes décisions par accident.

6. **Importance**
   Seules certaines choses sont architecturales : celles qui sont coûteuses à changer, qui touchent de nombreuses parties ou qui déterminent une qualité clé. L'architecture est sélective par définition.

7. **Environnement**
   Le système se situe parmi des utilisateurs, d'autres systèmes, une organisation, des lois et des budgets. Ceux-ci façonnent l'architecture et sont façonnés par elle.

Trois types de structures

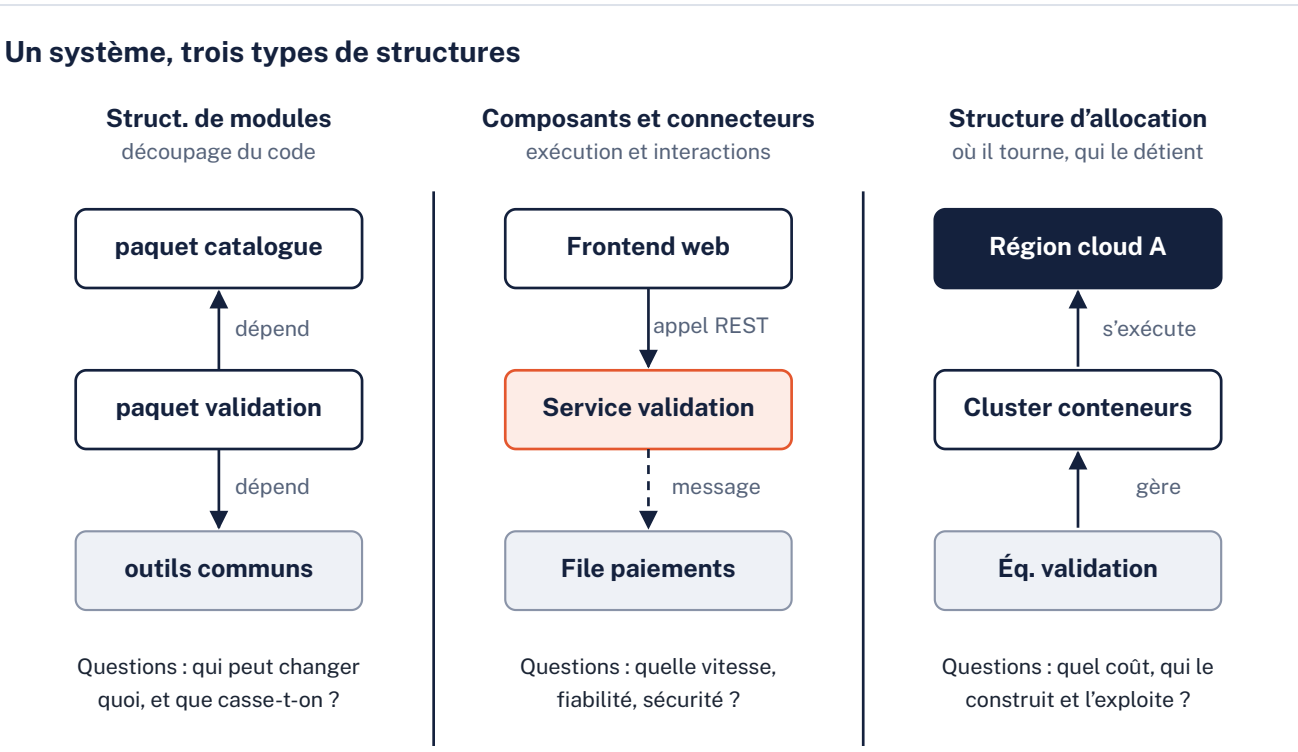
Les travaux du SEI sur la documentation, publiés sous le titre *Documenting Software Architectures : Views and Beyond* (Clements et ses collègues, 2e édition, 2010), regroupent les structures en trois types, et *Software Architecture in Practice* utilise le même regroupement. Connaître ces trois types vous protège de l'erreur la plus courante dans les discussions d'architecture : parler d'un type de structure alors que votre interlocuteur pense à un autre.

Les **structures de modules** montrent comment le code est découpé en unités d'implémentation, comme des paquets, des bibliothèques et des espaces de noms, et comment ces modules dépendent les uns des autres. Elles existent au moment du développement. Elles répondent à des questions comme : quelle équipe peut modifier quoi sans casser le travail des autres, et quel code peut être réutilisé ?

Les **structures de composants et connecteurs** montrent ce qui existe quand le système s'exécute : processus, services, clients, bases de données, et les connecteurs entre eux, comme les appels HTTP, les files de messages et les flux d'événements. Elles répondent à des questions d'exécution : quelle est la rapidité d'une requête, que se passe-t-il quand une partie tombe en panne, où les données sont-elles conservées ?

Les **structures d'allocation** font correspondre le logiciel à des choses extérieures au logiciel : au matériel et à l'infrastructure cloud (déploiement), aux équipes (affectation du travail), et aux fichiers et dépôts (implémentation). Elles répondent à des questions de coût, d'exploitation et d'organisation : où cela s'exécute-t-il, qui le construit, qui est réveillé la nuit quand il tombe en panne ?

Un même élément peut apparaître dans les trois. Une fonctionnalité de validation de commande est un *paquet* dans la structure de modules, un *service* en cours d'exécution dans la structure de composants et connecteurs, et un ensemble de *conteneurs dans une région cloud dont l'équipe validation est propriétaire* dans les structures d'allocation. La correspondance entre les structures n'est pas toujours biunivoque : un même module peut être déployé sous forme de plusieurs services, ou plusieurs modules peuvent s'exécuter dans un même processus. C'est précisément dans cet écart que se cachent bien des surprises.



Les structures de modules, de composants et connecteurs, et d'allocation montrent la même boutique en ligne sous trois angles et répondent à des questions différentes.

Tout système a une architecture

Comme l'architecture est faite de structures et des propriétés qu'elles ont réellement, tout système logiciel a une architecture, que quelqu'un l'ait conçue volontairement ou non, et qu'elle soit écrite ou non. Une macro de tableur devenue un outil critique pour l'entreprise a une architecture. Le code d'une start-up écrit à la hâte par trois personnes aussi. La question n'est jamais de savoir si un système a une architecture, mais si cette architecture est adaptée à son usage, comprise par les personnes qui y travaillent et façonnée délibérément.

Cela conduit à une distinction utile. L'architecture *voulue* (ou prescriptive) est ce que l'équipe croit ou a décidé que la structure devrait être : le diagramme du wiki, les règles convenues. L'architecture *réelle* (ou descriptive) est ce que font vraiment le code et l'infrastructure. Les deux s'éloignent avec le temps à moins que quelqu'un ne les maintienne alignées, et cet écart est l'une des principales sources de dette technique, traitée à la section 1.7. Une architecture non documentée n'est pas absente ; elle est simplement plus difficile à voir, à analyser et à modifier sans risque.

D'OÙ CELA VIENT

D'où viennent les définitions

L'architecture logicielle est apparue comme discipline à part entière au début des années 1990. L'article de Dewayne Perry et Alexander Wolf de 1992, *Foundations for the Study of Software Architecture*, proposait qu'une architecture se compose d'éléments, de leur forme et de la justification qui les sous-tend, et les travaux de David Garlan et Mary Shaw, qui ont abouti à leur livre de 1996 *Software Architecture: Perspectives on an Emerging Discipline*, ont catalogué des styles architecturaux récurrents. La première édition de *Software Architecture in Practice* est parue en 1998, et IEEE 1471, l'ancêtre d'ISO/IEC/IEEE 42010, a été publiée en 2000.

Les idées sont plus anciennes encore. Le système d'exploitation THE d'Edsger Dijkstra, organisé en couches (1968), et l'article de David Parnas de 1972 sur le découpage des systèmes en modules par le masquage des décisions de conception relèvent tous deux de la pensée architecturale avant même que le mot ne soit employé.

EXEMPLE COMMENTÉ

Exemple commenté : appliquer les définitions à une application de livraison de repas

Une entreprise de livraison de repas demande à un nouvel architecte de « nous dire quelle est notre architecture ». Il existe un seul diagramme, obsolète. En utilisant les idées partagées comme liste de contrôle, l'architecte produit une première description en une semaine.

Éléments	Application mobile, application web pour les restaurants, service des commandes, service d'attribution des livreurs, module de paiement, une base de données PostgreSQL partagée, un courtier de messages (message broker), un prestataire externe de paiement par carte.
Relations	Les applications appellent le service des commandes en HTTPS ; le service des commandes écrit dans la base de données partagée et publie des événements « commande passée » ; l'attribution des livreurs consomme ces événements ; le module de paiement appelle l'API du prestataire.
Propriétés	Le service des commandes est sans état et tourne en trois copies ; l'attribution des livreurs conserve son état en mémoire et tourne en une seule copie ; la base de données contient des données personnelles et des jetons de carte.
Environnement	Deux équipes de développement, une autorité de sécurité sanitaire des aliments, la législation sur la protection des données, un budget cloud fixe, des restaurants aux tablettes peu fiables.
Décisions importantes relevées	Une seule base de données partagée par tous les services ; une seule copie de l'attribution des livreurs ; des événements pour l'affectation des livreurs mais des appels directs pour les paiements.
Voulue ou réelle	L'ancien diagramme montre une base de données distincte par service. En réalité, le service des livreurs lit directement les tables des commandes, si bien que les deux services ne peuvent pas être modifiés indépendamment.

L'exercice fait apparaître deux risques que personne n'avait formulés : la copie unique de l'attribution des livreurs est un point de défaillance unique, et la base de données partagée couple deux équipes entre elles.

Aucun de ces risques n'était visible dans le code d'un service pris isolément. Tous deux sont des propriétés des structures et des relations, et c'est pourquoi ils relèvent de l'architecture.

La description a pris une semaine parce que personne n'avait consigné les décisions ni leurs raisons. Les consigner est le sujet du chapitre 4.

ATTENTION

Idées reçues courantes

- **« L'architecture, c'est le diagramme. »** Un diagramme est une description partielle d'une seule structure. L'architecture est l'ensemble des propriétés que le système possède réellement.
- **« L'architecture, c'est la pile technologique. »** Les choix technologiques peuvent être architecturaux, mais la répartition des responsabilités, des données et des dépendances compte généralement plus que le framework utilisé.
- **« L'architecture, c'est seulement pour les grands systèmes. »** Les petits systèmes ont aussi une architecture ; ils ont simplement moins de décisions importantes, si bien qu'il faut moins d'efforts pour les réussir.
- **« L'architecture se décide une fois pour toutes, au début. »** Toutes les définitions incluent l'évolution. L'architecture se façonne tout au long de la vie d'un système, comme l'explique la section 1.8.

Comment les professionnels utilisent les définitions

En pratique, les architectes expérimentés récitent rarement une définition, mais ils se servent de chacune comme d'une grille de lecture. La vision de l'ISO leur rappelle de regarder vers l'extérieur, vers l'environnement, et de distinguer ce que le système *est* de ce que les documents *disent*. La vision du SEI leur donne un vocabulaire de structures, d'éléments, de relations et de propriétés pour l'analyse, ainsi qu'un critère (« avons-nous besoin de cela pour raisonner sur le système ? »). La vision de Johnson leur rappelle qu'un modèle mental partagé par l'équipe est en soi un actif à entretenir. La vision de Booch leur indique où consacrer leur temps limité : aux décisions qu'il serait coûteux d'annuler.

Il en découle une règle empirique pratique. Quand vous ne savez pas si quelque chose est architectural, posez trois questions. Serait-ce coûteux ou risqué de le changer plus tard ? Cela touche-t-il plusieurs parties du système ou plusieurs équipes ? Cela détermine-t-il si le système atteint un objectif de qualité important, comme la rapidité, la disponibilité ou la sécurité ? Si la réponse à l'une de ces questions est oui, traitez-le comme de l'architecture : décidez-le délibérément, consignez les raisons, et assurez-vous que les personnes concernées le comprennent.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Quelle est la différence entre une architecture et une description d'architecture ?

L'architecture est l'ensemble des propriétés et structures fondamentales que le système possède réellement. La description est un produit de travail, comme des diagrammes et des documents, qui l'exprime, et elle peut être incomplète ou obsolète.

▼ Quel type de structure examineriez-vous pour savoir quelle équipe est réveillée quand un service tombe en panne ?

Une structure d'allocation, plus précisément la correspondance d'affectation du travail ou de propriété entre le logiciel et les équipes.

▼ Selon le critère de Booch, le choix d'une bibliothèque de journalisation est-il architectural ?

Généralement non, s'il est masqué derrière une petite interface et facile à remplacer. Il devient architectural si chaque module l'appelle directement, ce qui le rend coûteux à changer.

▼ Pourquoi la définition de l'ISO mentionne-t-elle l'environnement ?

Parce qu'une architecture ne peut être jugée qu'au regard des utilisateurs, des systèmes, de l'organisation, des lois et des budgets qui l'entourent ; la même structure peut être bonne dans un environnement et médiocre dans un autre.

▼ À vous : Une responsable de la livraison de repas dit : « Nous n'avons pas d'architecture, juste du code et un vieux diagramme. » Répondez-lui, puis classez ces éléments par types de structures : le paquet orders importe payments ; les commandes sont publiées vers un courtier ; l'affectation tourne sur trois serveurs gérés par l'équipe B.

Tout système a une architecture, conçue ou non ; ce qui lui manque, c'est une description à jour de celle-ci. L'import de package est une structure de modules, car il concerne le code. La publication vers un courtier est une structure composants-connecteurs, car elle concerne le système en exécution. L'affectation sur les serveurs gérés par l'équipe B est une structure d'allocation, qui associe le logiciel à l'infrastructure et aux personnes. Aucun diagramme ne montre les trois à la fois : une bonne description en exige donc plusieurs.

SOURCES

- ISO/IEC/IEEE 42010:2022 Software, systems and enterprise: Architecture description (ISO): https://www.iso.org/standard/74393.html

- Software Architecture in Practice, Fourth Edition (SEI, Carnegie Mellon University): https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/
- Software Architecture Guide (Martin Fowler): https://martinfowler.com/architecture/
- Documenting Software Architectures: Views and Beyond, Second Edition (SEI): https://www.sei.cmu.edu/library/documenting-software-architectures-views-and-beyond-second-edition/

SECTION 1.2 SUR 1.8 · DÉBUTANT · 20 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Architecture, conception et code : importance, portée et niveaux

L'IDÉE EN BREF

Quand on conçoit une maison, certains choix déterminent tout le reste : le nombre d'étages, l'emplacement de l'escalier, la position des murs porteurs. D'autres peuvent être modifiés en un week-end : la peinture, les étagères, les luminaires. Les deux relèvent de la conception, mais seul le premier type relève de l'architecture, car le changer plus tard oblige à éventrer la maison. Le logiciel, c'est pareil. Choisir comment un système est découpé en parties, ou comment les montants sont stockés partout, relève de l'architecture, car c'est coûteux à défaire. Choisir comment une fonction est écrite à l'intérieur d'une partie relève de la conception.

CE QU'IL FAUT SAVOIR

- L'architecture et la conception forment un continuum : toute architecture est conception, mais seules les décisions de conception importantes relèvent de l'architecture.
- Une décision est architecturalement significative lorsqu'elle est difficile ou coûteuse à annuler, qu'elle a une large portée, qu'elle détermine un attribut de qualité clé, qu'elle impose des contraintes aux autres ou qu'elle implique des compromis importants.
- Le niveau d'abstraction seul est un mauvais critère : des conventions de bas niveau, comme les formats d'identifiants ou de montants, peuvent être architecturales, et certaines boîtes de haut niveau sont anodines.
- Les décisions stratégiques fixent une direction à long terme pour de nombreuses personnes ; les décisions tactiques résolvent des problèmes immédiats et circonscrits, mais des choix tactiques répétés peuvent devenir architecturaux.
- L'architecture d'entreprise couvre toute une organisation (TOGAF est le cadre le plus connu), l'architecture de solution un problème métier à travers plusieurs systèmes, et l'architecture logicielle un seul système ; les architectures des données, de la sécurité et de l'infrastructure traversent tous les niveaux.
- Les décisions devraient être prises au niveau le plus bas qui dispose d'assez de contexte et de responsabilité, avec une consultation plus large pour celles qui touchent de nombreuses équipes.

Un continuum, pas un mur

Si l'architecture est l'ensemble des décisions et des structures importantes (section 1.1), alors la conception est tout le reste : les nombreuses décisions plus modestes sur le fonctionnement interne de chaque partie. La difficulté est qu'il n'existe pas de frontière nette entre les deux. Choisir de découper une plateforme de livraison de repas en un système de commandes et un système de livreurs est clairement architectural. Nommer une variable ne l'est clairement pas. Entre ces extrêmes s'étend une vaste zone intermédiaire de décisions (le format d'erreur d'une API, une règle de cache, le découpage d'un module en paquets) dont le statut dépend du contexte.

Grady Booch l'a joliment résumé : toute architecture est conception, mais toute conception n'est pas architecture. L'architecture et la conception détaillée sont des points d'un même continuum de décisions de conception, ordonnées selon leur importance. C'est pourquoi la question la plus utile n'est pas, dans l'absolu, « est-ce de l'architecture ou de la conception ? », mais « quelle est l'importance de cette décision ici, et quel soin mérite-t-elle ? ».

Une deuxième façon répandue de tracer la frontière est le niveau d'abstraction : l'architecture serait la vue de haut niveau (« des boîtes et des flèches »), la conception la vue de bas niveau (« des classes et des méthodes »). C'est une première approximation raisonnable, mais elle échoue dans les deux sens. Un détail de bas niveau peut être architectural : le format d'un identifiant partagé par vingt services, ou le choix d'une convention de fuseau horaire pour les dates stockées, peut coûter des mois à changer. Et une boîte de haut niveau peut être anodine : qu'un outil d'administration interne soit ou non un service séparé peut n'avoir aucune importance.

Ce qui rend une décision architecturalement significative

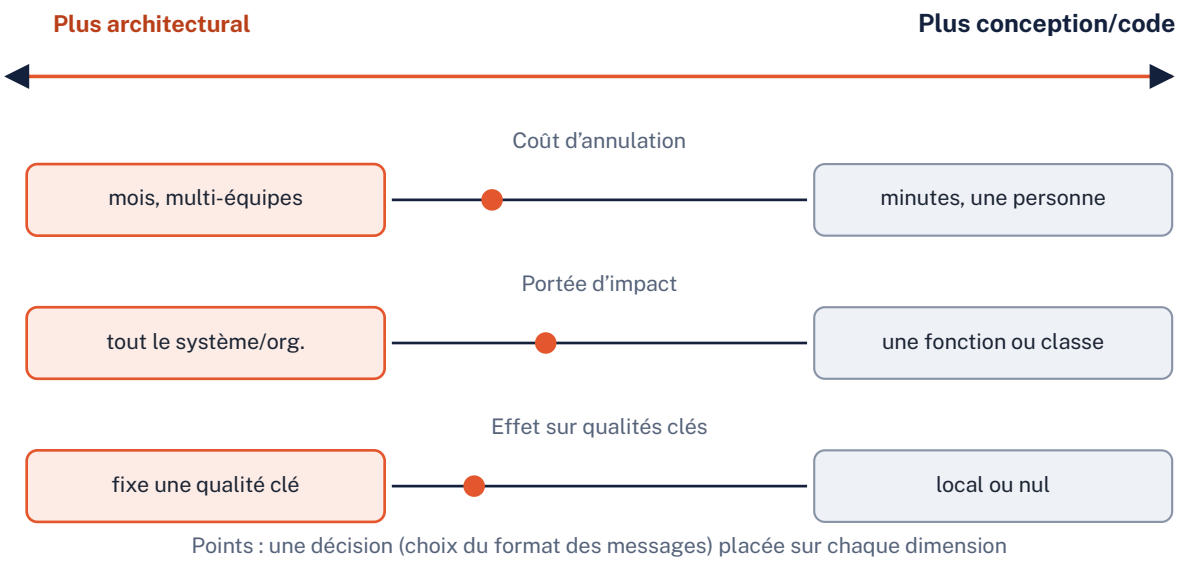
Une décision est architecturalement significative lorsqu'elle présente une ou plusieurs des caractéristiques suivantes. Ce ne sont pas des règles formelles issues d'une norme unique ; elles résument ce que disent, en d'autres termes, le SEI, Martin Fowler, Mark Richards et Neal Ford, entre autres.

Signes qu'une décision est architecturalement significative

Caractéristique	Question à se poser	Exemple
Difficile ou coûteuse à annuler	Si elle s'avère mauvaise dans un an, que coûtera son changement ?	Choisir un modèle de données en event sourcing plutôt que de stocker l'état courant
Large portée	Combien de composants, d'équipes ou de systèmes touche-t-elle ?	Définir le format des identifiants clients utilisés par tous les services
Détermine une qualité clé	Décide-t-elle si nous atteignons un objectif de performances, de disponibilité, de sécurité ou de coût ?	Placer un cache devant le catalogue de produits pour tenir un objectif de temps de réponse de 200 ms
Impose une contrainte aux autres	Limite-t-elle ce que d'autres équipes peuvent choisir ?	Imposer que tous les services publient leurs événements vers un seul courtier partagé
Implique des compromis importants	Renonçons-nous à quelque chose d'important pour obtenir autre chose ?	Accepter une cohérence à terme entre les commandes et le stock pour que l'étape de paiement reste disponible
Forte nouveauté ou risque élevé	Quelqu'un ici l'a-t-il déjà fait ? Et si la technologie nous lâche ?	Adopter une base de données que l'organisation n'a jamais exploitée en production

Mark Richards et Neal Ford, dans *Fundamentals of Software Architecture*, décrivent le spectre architecture/conception en des termes semblables : une décision penche d'autant plus vers l'architecture qu'elle est stratégique (long terme, nombreuses personnes impliquées), qu'elle demanderait d'efforts pour être changée et que ses compromis sont importants. Martin Fowler en propose une version condensée : l'architecture, c'est ce qui est difficile à changer, ce qui signifie aussi que l'un des objectifs d'un architecte est de rendre davantage de choses faciles à changer. Rendre une décision peu coûteuse à annuler, par exemple en la plaçant derrière une interface, la fait de fait glisser de l'architecture vers la conception. Cette idée, la réversibilité, est approfondie à la section 1.6.

Placer une décision sur le continuum



Points : une décision (choix du format des messages) placée sur chaque dimension

Trois dimensions placent une décision entre architecture et conception ; une décision peut obtenir un score différent sur chacune.

Décisions stratégiques et tactiques

Une distinction voisine oppose décisions stratégiques et décisions tactiques. Les décisions **stratégiques** fixent une direction pour longtemps et pour de nombreuses personnes : adopter un fournisseur de cloud, choisir entre un monolithe et des services distribués, décider que les données clients auront un système maître unique. Elles demandent généralement l'avis de plusieurs parties prenantes et méritent d'être consignées avec leurs raisons. Les décisions **tactiques** résolvent un problème immédiat dans un périmètre circonscrit : quelle bibliothèque utiliser pour analyser les dates dans un service, comment structurer une requête de base de données particulière, comment nommer les endpoints d'une API interne.

Stratégique ne veut pas dire lent. Beaucoup de décisions stratégiques peuvent et doivent être prises rapidement une fois les options claires, et certaines sont réversibles si elles sont bien structurées. Tactique ne veut pas dire sans importance : une mauvaise décision tactique répétée à cinquante endroits (par exemple, chaque service qui écrit sa propre logique de nouvelle tentative sans backoff) peut finir par constituer un problème architectural. Les architectes surveillent les décisions tactiques qui sont sur le point de devenir stratégiques à force de répétition, et les transforment en principe commun ou en capacité de plateforme.

Architecture ou conception ? Exemples de décisions et raisonnement

Décision (pour une boutique en ligne)	Verdict	Raisonnement
Découper le système en services déployables séparément pour le catalogue, le paiement et la livraison	Architecturale	Touche chaque équipe, le déploiement, la propriété des données et le comportement en cas de défaillance ; très coûteuse à annuler.
Utiliser PostgreSQL plutôt qu'une base de données orientée documents comme stockage principal des commandes	Architecturale	Façonne le modèle de données, les transactions et les compétences d'exploitation ; une migration ultérieure est coûteuse et risquée.
Utiliser une bibliothèque JSON précise dans le service de paiement	Conception	Locale à un service et remplaçable en une journée si elle est masquée derrière le code propre au service.
Exiger que chaque service émette des logs structurés avec un identifiant de corrélation commun	Architecturale	Une préoccupation transversale : elle ne fonctionne que si tout le monde la respecte, et elle détermine la rapidité avec laquelle les incidents peuvent être diagnostiqués.
Nommer la méthode qui calcule les frais de livraison	Conception (code)	Purement locale ; le renommage est automatisé par les outils.
Choisir le format des numéros de commande affichés aux clients et partagés avec le partenaire logistique	Architecturale	De bas niveau, mais elle franchit les frontières entre systèmes et entre entreprises, et le numéro est imprimé sur les colis ; le changer exige une coordination avec une partie externe.
Utiliser le patron Strategy pour prendre en charge plusieurs règles de frais de livraison	Conception	Façonne l'intérieur d'un seul composant ; les autres composants ne sont pas concernés.
Mettre en cache les prix des produits pendant 60 secondes	Cela dépend	Conception si les prix changent rarement et qu'un prix périmé est sans conséquence ; architecture si un prix périmé crée un problème juridique ou financier, car la décision détermine alors une qualité clé (l'exactitude).

Les niveaux d'architecture

Le mot architecture s'emploie aussi à plusieurs niveaux de portée, et les intitulés de poste suivent ces niveaux. Les confondre provoque bien des malentendus : un architecte d'entreprise et un architecte logiciel peuvent tous deux être « l'architecte », alors que leurs travaux se recoupent à peine.

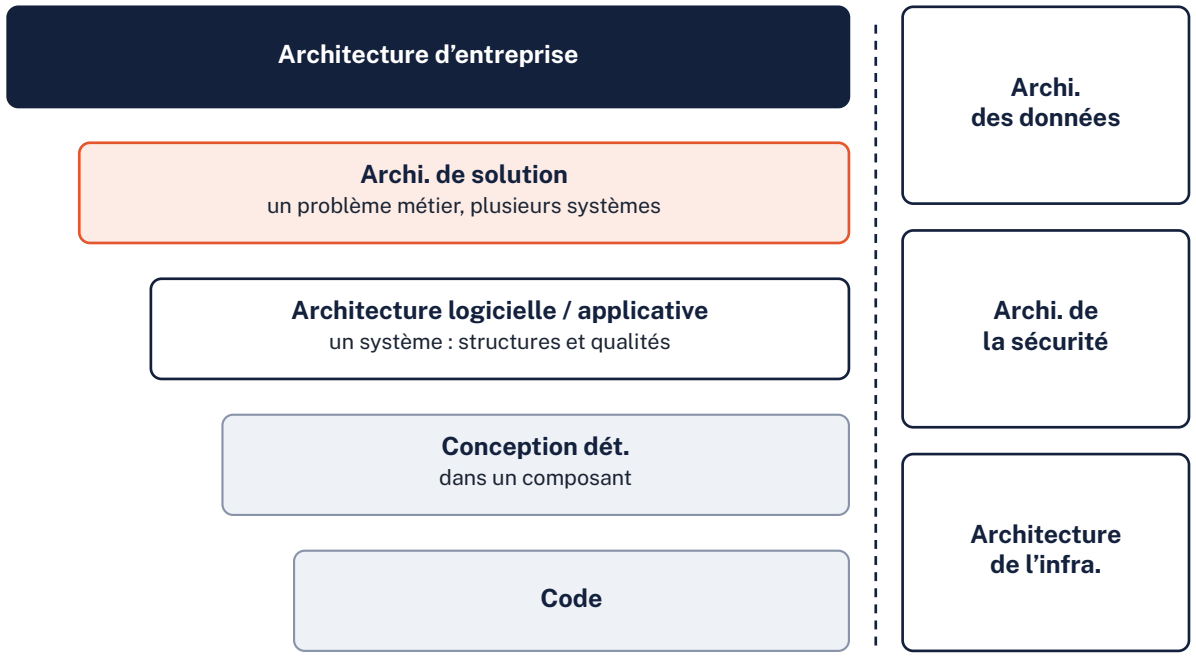
L'**architecture d'entreprise** porte sur toute une organisation : ses capacités métier (ce qu'elle doit être capable de faire, comme « instruire les sinistres » ou « intégrer de nouveaux clients »), les informations qu'elle détient, son portefeuille d'applications et ses plateformes technologiques, et la façon dont tout cela doit évoluer pour servir la stratégie. Les livrables typiques sont des cartes de capacités, des feuilles de route vers l'état cible, des normes technologiques et des décisions sur les systèmes à retirer. Le cadre le plus utilisé dans ce domaine est TOGAF, publié par The Open Group, dont la version actuelle est le TOGAF Standard, 10e édition (2022). Il fournit une méthode (l'Architecture Development Method) et un ensemble de contenus pour planifier et gouverner les transformations de l'entreprise. Les organisations l'utilisent à des degrés divers ; beaucoup en adoptent certaines parties et d'autres le trouvent lourd, et ce cours ne le décrit que pour que vous puissiez le reconnaître.

L'**architecture de solution** traite un problème métier ou un programme, qui s'étend généralement sur plusieurs applications, équipes et fournisseurs : par exemple, un hôpital qui met en place la prise de rendez-vous en ligne et doit relier un nouveau frontend web, le système existant de gestion administrative des patients, un prestataire de SMS et le service d'identité. L'architecte de solution fait en sorte que les pièces s'emboîtent, souvent sous des contraintes commerciales et contractuelles.

L'**architecture logicielle ou applicative**, sujet principal de ce cours, concerne les structures et les qualités d'un seul système : sa décomposition, ses interactions à l'exécution, son déploiement, ses données et son évolution.

Trois spécialités traversent tous ces niveaux. L'**architecture des données** porte sur la façon dont les données sont modélisées, possédées, stockées, déplacées, gouvernées et utilisées pour l'analytique (chapitre 7). L'**architecture de sécurité** porte sur l'identité, l'accès, la protection des données, les modèles de menaces et le placement des contrôles de sécurité (chapitre 8). L'**architecture d'infrastructure** (souvent appelée aujourd'hui architecture cloud ou de plateforme) porte sur le calcul, les réseaux, le stockage, les environnements et les plateformes de déploiement (chapitre 10).

Niveaux et spécialités transversales



Chaque niveau contraint le suivant ; les spécialités traversent tous les niveaux

La portée se resserre de l'entreprise jusqu'au code ; les architectures des données, de la sécurité et de l'infrastructure s'appliquent à tous les niveaux.

Qui prend quelles décisions

Dans une organisation saine, les décisions sont prises au niveau le plus bas qui dispose d'assez de contexte et qui répond des conséquences. L'architecture n'est pas la chasse gardée des personnes dont l'intitulé de poste contient le mot « architecte ». Une répartition par défaut raisonnable ressemble à ceci, même si chaque organisation l'ajuste.

Type de décision	Généralement prise par	Généralement consultés
Normes et plateformes technologiques de l'organisation (p. ex. fournisseur de cloud approuvé, plateforme d'identité)	Architectes d'entreprise, CTO, comité d'architecture	Architectes de solution et architectes logiciels, sécurité, finance
La façon dont plusieurs systèmes et fournisseurs se combinent pour résoudre un problème métier	Architecte de solution	Product owners, architectes d'entreprise, architectes des fournisseurs
La structure principale d'un système : décomposition, style de communication, propriété des données	Architecte logiciel ou responsable technique de l'équipe, avec l'équipe	Exploitation, sécurité, autres équipes concernées
La conception à l'intérieur d'un composant : classes, patrons, algorithmes	L'équipe de développement	L'architecte, seulement si un principe ou une qualité est en jeu
Les choix au niveau du code : nommage, refactoring local	Chaque développeur, via la revue de code	Personne d'autre, normalement

Deux signaux d'alerte montrent que la répartition est mauvaise. Si les architectes décident des structures de classes internes, ils deviennent un goulot d'étranglement et l'équipe cesse de réfléchir. Si les équipes prennent chacune de leur côté des décisions qui ont des effets sur tout le système (chacune choisissant son propre courtier de messages, par exemple), l'organisation paie plus tard le prix de l'incohérence. La façon de consigner quelles décisions exigent une consultation plus large, et de faciliter leur enregistrement, est traitée dans décisions et compromis et, en profondeur, dans la partie du chapitre 4 consacrée aux enregistrements de décision d'architecture (ADR).

EXEMPLE COMMENTÉ

Exemple commenté : classer les décisions d'une application de VTC

Une start-up de VTC dresse la liste de cinq décisions en attente. Le responsable technique utilise les trois signes (coût d'annulation, portée, effet sur les qualités clés) pour déterminer lesquelles exigent une trace écrite et une revue plus large.

1. Stocker les positions des trajets dans une base de données de séries temporelles	Annulation : coût élevé (migration de données). Portée : trajets, facturation, analytique. Qualité : performances du suivi en temps réel. Verdict : architecturale ; la rédiger et la revoir avec les équipes données et exploitation.
2. Utiliser une bibliothèque cliente HTTP précise dans le service de tarification	Annulation : coût faible. Portée : un service. Qualité : aucune de notable. Verdict : conception ; l'équipe tarification décide.
3. Calculer les tarifs sur le serveur, jamais sur le téléphone	Annulation : coût moyen. Portée : applications et backend. Qualité : sécurité et intégrité de la tarification. Verdict : architecturale ; c'est un principe directeur.
4. Représenter partout l'argent en unités monétaires mineures entières (pence, centimes)	Annulation : coût très élevé dès que les données et les API des partenaires l'utilisent. Portée : chaque service. Qualité : exactitude. Verdict : architecturale, bien que de bas niveau.
5. Utiliser les couleurs du mode sombre dans l'app du chauffeur	Annulation : coût négligeable. Portée : un ensemble d'écrans. Verdict : conception.

Deux des trois décisions architecturales (3 et 4) ne concernent pas du tout des boîtes et des flèches ; ce sont des principes et des conventions. Le niveau d'abstraction seul les aurait mal classées.

La décision 4 est le type le plus souvent manqué : elle ressemble à un détail de code, mais devient très coûteuse à changer dès que d'autres systèmes en dépendent.

Questions fréquentes

Le choix d'un langage de programmation est-il une décision architecturale ?
Souvent, oui. Il influe sur le recrutement, les bibliothèques disponibles, les caractéristiques de performances et la facilité avec laquelle le code peut être partagé entre composants. Dans un système fait de nombreux services indépendants, le langage d'un petit service peut être un choix local ; le langage par défaut de toute l'organisation est architectural.

Un schéma de base de données relève-t-il de l'architecture ou de la conception ?
Cela dépend de qui l'utilise. Une table utilisée uniquement dans un service, derrière sa propre API, relève de la conception. Un schéma lu directement par plusieurs services, des rapports et des exports vers des partenaires relève de l'architecture, car chaque modification doit être coordonnée avec tous. Beaucoup d'équipes rendent les schémas privés à un seul service précisément pour les ramener du côté de la conception.

Le développement agile signifie-t-il qu'il n'y a pas d'architecture ?
Non. Tout système a une architecture ; les approches agiles changent le moment et la manière dont elle est décidée, en privilégiant des étapes petites et réversibles et en reportant les décisions irréversibles jusqu'à ce que l'on en sache assez. La section 1.8 traite de ce sujet.

Qui décide si une décision est importante ?
Généralement l'équipe et son architecte ou son responsable technique, ensemble, à l'aide de critères convenus comme les signes présentés dans cette section. Mettre ces critères par écrit évite deux écueils : débattre de chaque choix, et découvrir après coup qu'une équipe a pris seule une décision qui touche tout le système.

Comment les professionnels appliquent la distinction

Les équipes expérimentées transforment le continuum en une routine légère. Elles conviennent d'une courte liste des types de décisions qui exigent une trace écrite et une consultation plus large, par exemple tout ce qui modifie une API publique, introduit un nouveau stockage de données ou une nouvelle technologie, franchit les frontières entre équipes ou affecte un objectif de qualité déclaré. Tout le reste est décidé par les personnes qui font le travail, au travers de la revue de code habituelle. Elles revoient la liste quand quelque chose passe entre les mailles du filet ou quand les revues ressemblent à de la bureaucratie.

Elles cherchent aussi activement des moyens de réduire l'ensemble des décisions architecturales, car chaque décision qui devient peu coûteuse à annuler est une décision de moins à réussir du premier coup. Masquer un fournisseur derrière une interface, rendre une base de données privée à un seul service, utiliser des feature flags pour les comportements risqués et déployer par petits incréments déplacent tous des décisions de l'extrémité architecturale du continuum vers l'extrémité conception.

POUR ALLER PLUS LOIN

L'idée intension/localité

Amnon Eden et Rick Kazman ont proposé en 2003 une distinction plus formelle, fondée sur deux critères. L'*intension* demande si un énoncé est général (il décrit toute une classe d'implémentations possibles) ou spécifique (il décrit une seule implémentation) ; la *localité* demande s'il peut être vérifié dans une seule partie du système ou s'il contraint l'ensemble. Les énoncés architecturaux sont généraux et *non locaux* : ils contraignent tout le système (« tous les composants communiquent par le bus de messages »), si bien qu'ils peuvent être enfreints n'importe où. Les énoncés de conception sont généraux mais locaux (« utiliser le patron Observer dans ce module »). Les énoncés d'implémentation sont spécifiques et locaux (« cette classe a ces méthodes »). Ce critère est utile pour rédiger des règles : un énoncé qui doit être vrai partout est candidat à devenir un principe architectural et une vérification automatisée, une fonction d'aptitude.

ATTENTION

Erreurs à éviter

- Tout traiter comme de l'architecture, ce qui produit des comités de revue qui ralentissent chaque changement et des équipes qui cessent de s'approprier leur travail.
- Ne rien traiter comme de l'architecture, ce qui laisse des propriétés de l'ensemble du système émerger par accident de centaines de choix locaux.
- Supposer qu'une décision relève de la conception parce qu'elle est de bas niveau. Les formats de données, les identifiants, les représentations du temps et de l'argent et les contrats d'API sont souvent architecturaux.
- Supposer qu'une décision est architecturale parce qu'elle implique un gros composant. Si elle est facilement réversible et d'effet local, elle n'a pas besoin de cérémonie.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi le choix du format d'un identifiant peut-il être architectural ?

Parce qu'une fois que de nombreux services, bases de données et partenaires externes en dépendent, le changer exige un travail coordonné entre tous, si bien que son coût du changement et sa portée sont élevés.

▼ Quelle est la différence entre l'architecture d'entreprise et l'architecture de solution ?

L'architecture d'entreprise couvre les capacités, les informations, les applications et la technologie de toute l'organisation ; l'architecture de solution traite un problème métier qui s'étend généralement sur plusieurs systèmes et équipes.

▼ Comment un architecte peut-il faire passer une décision de l'architecture vers la conception ?

En la rendant moins coûteuse à annuler, par exemple en masquant une technologie derrière une interface pour qu'elle puisse être remplacée localement.

▼ À vous : Une équipe de billetterie dresse la liste de trois décisions : nommer une fonction utilitaire ; stocker tous les montants en pence entiers dans tous les services ; donner à chaque service sa propre base de données ou en partager une. Lesquelles sont architecturales ? Servez-vous du coût de retour en arrière, de la portée et de l'effet sur les qualités.

Le nom de la fonction utilitaire relève de la conception locale : peu coûteux à inverser, portée minime, aucun effet sur les qualités. Le format monétaire semble de bas niveau, mais il est architectural, car il se répand dans chaque service et chaque enregistrement stocké, coûte cher à changer plus tard et détermine l'exactitude. Bases de données partagées ou séparées est clairement architectural : très difficile à inverser, portée large, et cela façonne la déployabilité et la cohérence. Ces deux derniers méritent une trace écrite et une revue plus large ; pas le premier.

SOURCES

- Who Needs an Architect? Martin Fowler, IEEE Software, 2003 (PDF): https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf
- Software Architecture Guide (Martin Fowler): https://martinfowler.com/architecture/
- The TOGAF Standard (The Open Group): https://www.opengroup.org/togaf
- iSAQB CPSA-Foundation curriculum: https://public.isaqb.org/curriculum-foundation/curriculum-foundation-en.html

SECTION 1.3 SUR 1.8 · DÉBUTANT · 19 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Pourquoi l'architecture compte : coût du changement, qualité, risques et échecs réels

L'IDÉE EN BREF

Imaginez deux cuisines qui préparent la même carte. Dans l'une, chaque poste est disposé de sorte que les cuisiniers se croisent rarement ; dans l'autre, le seul réfrigérateur se trouve derrière le lave-vaisselle. Les deux servent les mêmes plats lors d'une soirée calme. Un samedi de grande affluence, ou quand la carte change, la seconde cuisine se bloque. Le logiciel, c'est pareil : les fonctionnalités peuvent sembler identiques, mais l'agencement des parties détermine la rapidité, la fiabilité et le coût du système, et à quel point il est pénible de le modifier. Certains des échecs logiciels les plus coûteux de l'histoire étaient des problèmes d'agencement, et non de lignes de code isolées.

CE QU'IL FAUT SAVOIR

- L'architecture détermine en grande partie les attributs de qualité tels que les performances, la disponibilité, la sécurité et le coût, car ils naissent de la façon dont les parties sont agencées et interagissent.
- Le coût de la remise en cause d'une décision augmente à mesure que du code, des données, des intégrations et des équipes s'appuient sur elle : les premiers choix structurels déterminent donc le coût et la rapidité de tout le travail ultérieur.
- L'architecture régit les risques, la façon dont le travail peut être réparti entre les équipes (loi de Conway), la communication et la capacité à estimer les changements.
- Knight Capital a perdu plus de 460 millions de dollars US le 1er août 2012 après qu'un déploiement manuel eut laissé l'un de ses huit serveurs exécuter un ancien code qu'un flag réutilisé a réactivé.
- Le GAO a constaté que HealthCare.gov avait été lancé en octobre 2013 sans vérification du respect de ses exigences de performances, après des exigences non arrêtées, un contrôle insuffisant et des tests incomplets.
- En 2023, le passage chez Prime Video d'un service de supervision d'étapes serverless distribuées à un processus unique aurait réduit ses coûts d'infrastructure de plus de 90 %, ce qui montre que l'architecture est une question d'adéquation, et non de mode.

Les décisions qu'il coûte cher de rater

La section 1.1 a défini l'architecture comme l'ensemble des structures et des décisions significatives qui façonnent un système, l'importance d'une décision se mesurant au coût du changement. Cette section pose la question pratique qui s'ensuit : et alors ? Pourquoi une équipe, un product owner ou un directeur financier devraient-ils se soucier de l'architecture, plutôt que simplement des fonctionnalités ?

La réponse courte est que l'architecture détermine, plus que tout autre facteur technique, ce à quoi le système *ressemblera* : à quel point il sera rapide, fiable, sécurisé, bon marché à exploiter et surtout facile à modifier. Les fonctionnalités sont ce que fait un système. L'architecture détermine à quel point il le fait bien et ce qu'il en coûtera de faire autre chose l'an prochain. Beaucoup de grands échecs logiciels et beaucoup de succès coûteux remontent à une poignée de décisions structurelles.

L'architecture détermine les attributs de qualité

Un attribut de qualité est une propriété mesurable de la façon dont un système fait son travail : performances, disponibilité, sécurité, modifiabilité, scalabilité, utilisabilité, efficacité des coûts, et d'autres encore. Le chapitre 2 les précise. Le point essentiel ici est que la plupart des attributs de qualité sont *émergents* : ils naissent de la façon dont les parties sont agencées et interagissent, et non d'un morceau de code en particulier.

Prenons la disponibilité. Si l'étape de paiement d'une boutique en ligne doit faire cinq appels synchrones successifs à des services disponibles chacun 99,9 % du temps, la chaîne dans son ensemble ne peut pas être disponible à plus de 99,5 % environ, aussi bien écrit que soit chaque service, car les probabilités se multiplient. Ajouter une file d'attente pour que certaines de ces étapes aient lieu plus tard, ou mettre des résultats en cache pour que certains appels deviennent inutiles, change le calcul. Aucun de ces changements n'est visible à l'intérieur d'une fonction ; tous deux sont architecturaux. Les performances, la sécurité et le coût se comportent de la même façon. Une page de connexion sécurisée ne rend pas un système sécurisé si un service interne sans authentification est accessible depuis internet.

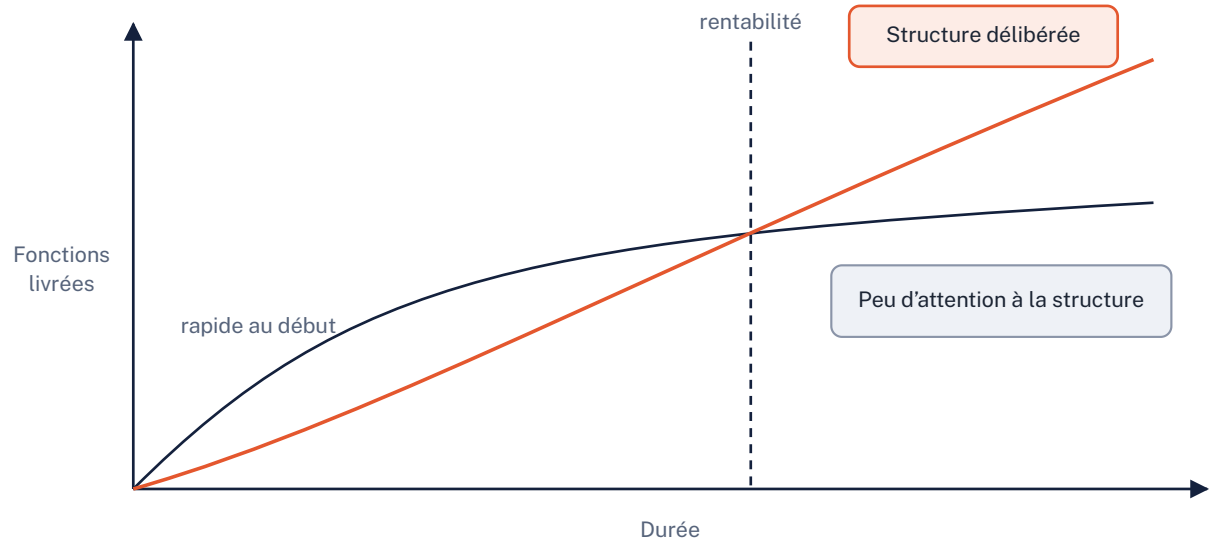
Les fonctionnalités, en revanche, sont généralement assez indépendantes de la structure. Vous pouvez mettre en œuvre les mêmes fonctionnalités d'une application bancaire sous forme d'une seule unité déployable ou de cinquante services ; les utilisateurs voient les mêmes boutons. Ce qui change, c'est tout le reste : la rapidité avec laquelle un correctif peut être mis en production, ce qui se passe quand une partie tombe en panne, le nombre d'ingénieurs que le système peut absorber, et le montant de la facture cloud.

Coût et rapidité du changement

L'essentiel du coût d'un système qui réussit intervient après sa première mise en production, à mesure qu'il est modifié pour répondre à de nouveaux besoins. La structure compte donc surtout parce qu'elle régit le coût de chaque changement futur. Dans un système bien structuré, une modification des règles fiscales touche un seul module. Dans un système enchevêtré, elle en touche vingt, qui demandent chacun des tests et de la coordination.

L'*hypothèse de l'endurance de la conception* (design stamina hypothesis) de Martin Fowler (2007) décrit cet effet. Une équipe qui prête peu d'attention à la structure interne livre souvent plus vite au début. Avec le temps, l'enchevêtrement accumulé ralentit chaque changement, et une équipe qui a investi dans la structure finit par la dépasser. Fowler parle d'hypothèse parce qu'elle est difficile à mesurer rigoureusement, et il suggère que le croisement intervient généralement en quelques semaines plutôt qu'en quelques mois. Ce ralentissement correspond aux intérêts de la dette technique, que traite la section 1.7.

L'hypothèse de l'endurance de la conception



Une esquisse conceptuelle d'après Martin Fowler : prêter moins d'attention à la structure va plus vite au début, mais une structure délibérée maintient le rythme de livraison.

EXEMPLE COMMENTÉ

Exemple commenté : le coût d'une décision dans le temps

Une boutique en ligne stocke les montants sous forme de nombres à virgule flottante. L'arithmétique à virgule flottante ne peut pas représenter exactement de nombreuses valeurs décimales (0,1 + 0,2 ne fait pas exactement 0,3), si bien que les totaux diffèrent parfois d'un penny de ce qu'attendent les comptables. Les chiffres ci-dessous sont des hypothèses illustratives, et non des données du secteur, choisies pour montrer comment le coût du changement augmente.

Mois 1 : décider autrement	Deux développeurs conviennent de stocker plutôt des unités monétaires mineures entières (des pence). Effort : environ une demi-journée. Coût, à un tarif supposé de £600 par jour-personne : environ £300.
Année 3 : ce qui a été construit sur la décision	35 services manipulent des montants ; 120 colonnes de base de données les stockent ; 200 millions de lignes de commandes historiques ; 4 intégrations partenaires (prestataire de paiement, entrepôt, comptabilité, place de marché) les échangent.
Année 3 : modifications du code	35 services à environ 3 jours-personne chacun pour modifier, tester et mettre en production = 105 jours-personne.
Année 3 : migration des données	Convertir les colonnes, migrer l'historique, rapprocher les totaux : environ 20 jours-personne, plus une fenêtre de maintenance planifiée.
Année 3 : coordination avec les partenaires	4 partenaires à environ 10 jours-personne chacun, y compris leurs tests et les modifications de contrat = 40 jours-personne.
Année 3 : tests de bout en bout et rapprochement avec le service financier	Environ 30 jours-personne.
Année 3 : total	Environ 195 jours-personne, soit à peu près £117 000, plus le risque d'erreurs pendant la migration et des mois de délai. C'est plusieurs centaines de fois le coût du mois 1.

La décision elle-même n'a jamais changé de taille. Ce qui a grandi, c'est tout ce qui a été construit par-dessus : code, données, partenaires et habitudes.

C'est pourquoi Ralph Johnson décrivait l'architecture comme les décisions que l'on aimerait pouvoir prendre correctement dès le début, et pourquoi les architectes recherchent les conventions de bas niveau à large portée.

Une autre stratégie consiste à rendre ces décisions moins coûteuses à inverser, par exemple en plaçant dès le départ toute la gestion des montants derrière un type ou une bibliothèque partagés, pour qu'un changement ultérieur reste local.

Risques, équipes, communication et estimation

Risques. C'est dans l'architecture que la plupart des risques techniques sont soit contenus, soit amplifiés. Une conception comportant un point de défaillance unique, une intégration non testée entre des parties construites indépendamment, ou une technologie non éprouvée en son cœur porte un risque qu'aucun codage soigneux ne supprime. Une grande partie du travail architectural relève de la gestion des risques : identifier ce qui pourrait très mal tourner et façonner la structure pour que cela ne puisse pas arriver, ou ne fasse que des dégâts limités si cela arrive.

Répartir le travail entre les équipes. Melvin Conway a observé en 1968 que les organisations qui conçoivent des systèmes tendent à produire des conceptions qui reflètent leurs propres structures de communication. C'est ce qu'on appelle aujourd'hui la loi de Conway, et elle joue dans les deux sens. Un système dont les parties ont des frontières claires et des interfaces étroites permet à plusieurs équipes de travailler en parallèle avec peu de coordination. Un système où tout dépend de tout oblige chaque équipe à assister à chaque réunion. L'architecture et la conception de l'organisation sont donc étroitement liées, comme l'explique la section du chapitre 3 sur la loi de Conway.

Communication. Une architecture donne aux personnes un vocabulaire commun et une carte commune. Les nouveaux ingénieurs apprennent le système plus vite, les chefs de produit comprennent pourquoi certaines demandes sont bon marché et d'autres coûteuses, et les équipes d'exploitation savent ce qui dépend de quoi pendant un incident.

Estimation et planification. La décomposition en parties connues rend l'estimation possible : un changement peut être rattaché aux composants qu'il touche, et chacun peut être estimé par l'équipe qui le connaît. Sans structure claire, les estimations sont des suppositions, car personne ne peut dire avec certitude jusqu'où un changement se propagera.

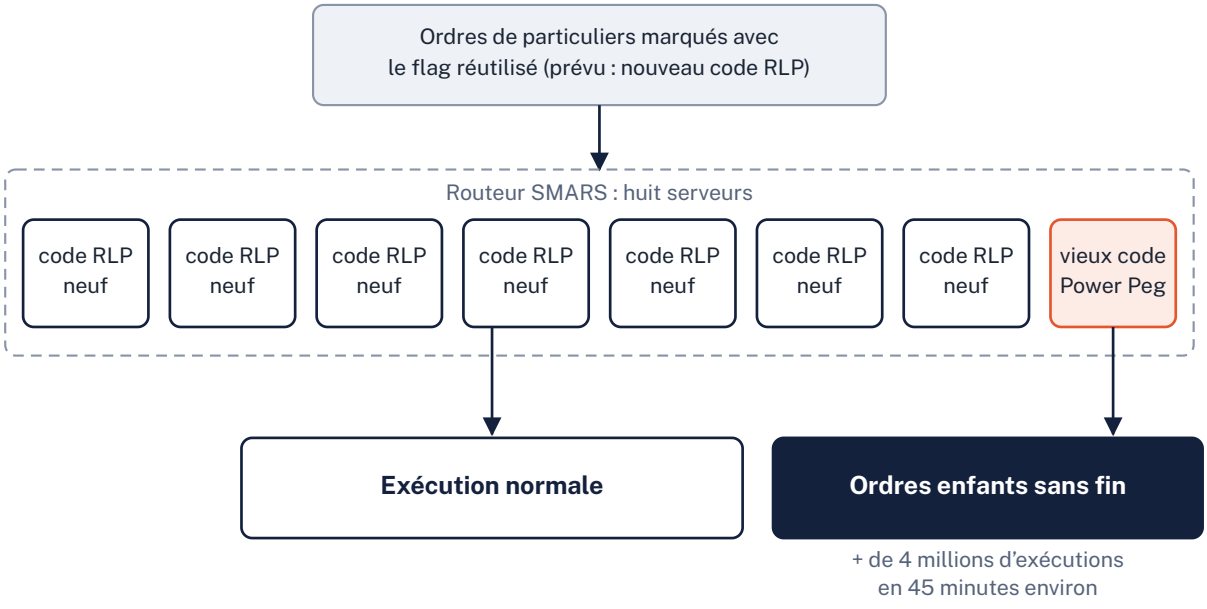
Étude de cas : Knight Capital, 1er août 2012

Knight Capital était l'un des plus grands teneurs de marché sur les actions américaines. Selon l'ordonnance de la Securities and Exchange Commission des États-Unis du 16 octobre 2013, au cours des 45 premières minutes de négociation du 1er août 2012, le routeur d'ordres automatisé de Knight (appelé SMARS) a envoyé des millions d'ordres non voulus sur le marché en traitant 212 petits ordres de particuliers. Il a reçu plus de 4 millions d'exécutions sur 154

titres, portant sur plus de 397 millions d'actions, et s'est retrouvé avec d'importantes positions non désirées. Knight a perdu plus de 460 millions de dollars US, a eu besoin d'un investissement extérieur d'urgence en quelques jours, et la SEC lui a ensuite infligé une amende de 12 millions de dollars US pour violation des règles d'accès au marché.

L'ordonnance de la SEC décrit le mécanisme. Des années plus tôt, Knight avait cessé d'utiliser une ancienne fonction appelée Power Peg, mais son code était resté dans le routeur et pouvait encore être déclenché : c'était du code mort qui n'était pas mort. En 2012, Knight a écrit du nouveau code pour un nouveau programme du NYSE, le Retail Liquidity Program, et ce nouveau code réutilisait un flag (indicateur) qui servait autrefois à activer Power Peg. Le nouveau code a été déployé manuellement sur les huit serveurs du routeur. Un technicien ne l'a pas copié sur l'un d'eux, aucun second technicien n'a vérifié le déploiement, et Knight n'avait aucune procédure écrite exigeant une telle vérification. Quand le flag a été activé, sept serveurs ont exécuté le nouveau code ; le huitième a interprété le flag comme « exécuter Power Peg » et a envoyé des ordres sans reconnaître qu'ils avaient déjà été exécutés. Des e-mails automatiques signalant un problème ont été envoyés avant l'ouverture du marché, mais personne n'y a donné suite.

Comment un seul serveur pas à jour a causé les pertes de Knight Capital



Pas de second technicien pour revoir le déploiement ; aucun contrôle auto de la concordance des serveurs

Un flag réutilisé, un serveur pas à jour et aucune vérification du déploiement : la défaillance tenait à la structure et au processus, pas à une ligne de code isolée.

Il serait injuste de qualifier cela de simple bug de codage, et tout aussi injuste d'y voir une pure erreur d'exploitation. Les leçons architecturales portent sur la structure et la déployabilité : une fonctionnalité obsolète laissée accessible dans un système en production ; un même flag doté

de deux significations dans différentes versions du code ; un processus de déploiement sans vérification automatique que tous les serveurs exécutaient la même version ; et, comme l'a souligné la SEC, des contrôles insuffisants pour limiter les dégâts une fois que quelque chose avait mal tourné. Chacune est une décision sur la façon dont le système est construit et mis en production, et non sur une ligne de code isolée.

Étude de cas : le lancement de HealthCare.gov, octobre 2013

HealthCare.gov, la place de marché fédérale américaine de l'assurance santé, a ouvert les inscriptions le 1er octobre 2013. De nombreuses personnes n'ont pas pu créer de compte ni terminer leur demande, et le site a subi des pannes pendant des semaines. Le Government Accountability Office (GAO) des États-Unis et l'Office of Inspector General (OIG) du Department of Health and Human Services (ministère de la Santé et des Services sociaux) ont ensuite enquêté, et leurs conclusions sont la meilleure source pour comprendre ce qui n'a pas fonctionné.

Le rapport du GAO de juillet 2014 (GAO-14-694) a constaté que les Centers for Medicare and Medicaid Services (CMS) avaient attribué des contrats avant que les exigences clés ne soient arrêtées, que les exigences n'avaient cessé de changer, que le contrôle était insuffisant et que les CMS avaient lancé le système sans vérifier qu'il satisfaisait à ses exigences de performances. Les engagements au titre du principal contrat de la place de marché sont passés de 56 millions de dollars US à plus de 209 millions de dollars US entre 2011 et début 2014. Un second rapport du GAO, en mars 2015 (GAO-15-238), a constaté que les systèmes n'avaient pas été entièrement testés avant le lancement, et que les problèmes au lancement comprenaient une planification de capacité insuffisante, des erreurs de codage et des fonctions prévues qui n'avaient pas été mises en œuvre. L'étude de cas de l'OIG de 2016 a désigné l'absence de leadership clair et la mauvaise coordination entre le travail sur les politiques publiques et le travail technique comme causes centrales. Après d'intenses travaux correctifs, le site a pu répondre à une forte demande des consommateurs en deux mois environ.

L'histoire de HealthCare.gov relève surtout de la gestion de programme, mais sa leçon architecturale est directe. La place de marché a été construite par plusieurs prestataires sous forme de parties distinctes qui devaient fonctionner ensemble, dont un hub de données connecté à d'autres agences fédérales, et personne n'a vérifié de bout en bout, sous une charge réaliste, que l'ensemble assemblé atteignait ses objectifs de capacité et de performances. Les attributs de qualité comme la capacité sont des propriétés du système entier : ils ne peuvent donc être garantis que par quelqu'un qui est responsable de la structure du système entier et qui le teste comme un tout.

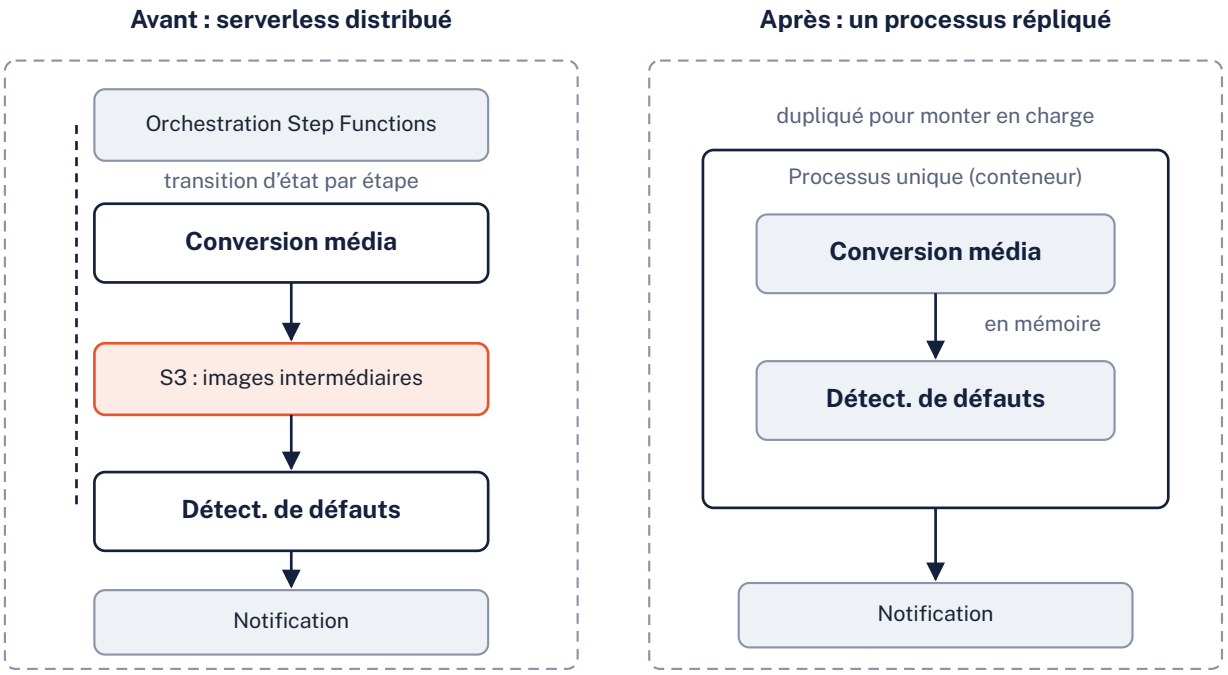
-

Étude de cas : Prime Video, ou l'architecture comme adéquation et non comme mode

En mars 2023, un ingénieur de l'équipe Video Quality Analysis d'Amazon Prime Video a publié sur le blog Prime Video Tech un billet consacré à un service qui surveille les flux vidéo pour y détecter des défauts de qualité. La première version avait été construite à partir de composants serverless distribués : AWS Step Functions orchestrait les étapes et des fonctions AWS Lambda effectuaient le travail, les images vidéo intermédiaires passant d'une étape à l'autre par le stockage Amazon S3. L'équipe a indiqué que cette conception se heurtait à une limite de scalabilité stricte à environ 5 % de la charge prévue et aurait été trop coûteuse à pleine échelle, en grande partie à cause des frais d'orchestration par étape et des nombreuses requêtes de stockage.

L'équipe a regroupé les étapes de conversion des médias et de détection des défauts dans un processus unique s'exécutant dans des conteneurs sur Amazon ECS, en faisant passer les images en mémoire plutôt que par le stockage, et a assuré la montée en charge en exécutant davantage de copies de ce processus. Le billet indiquait que les coûts d'infrastructure avaient baissé de plus de 90 % et que le service pouvait monter davantage en charge. Le billet a été largement partagé sous le titre « Amazon abandonne les microservices », ce qu'il ne disait pas : le changement concernait un seul service, et l'auteur notait que les microservices et les composants serverless fonctionnent bien à grande échelle, mais que le choix entre eux et un monolithe doit se faire au cas par cas.

Le service de supervision de Prime Video, avant et après



Les mêmes étapes, regroupées : faire passer les données en mémoire au sein d'un seul processus a supprimé les coûts d'orchestration par étape et de stockage.

La leçon n'est pas que les monolithes l'emportent sur le serverless. C'est que l'architecture est une question d'adéquation. La conception serverless était rapide à construire et appropriée pour une charge de travail initiale plus petite. Quand la caractéristique déterminante de cette charge (de très gros volumes de données transmis entre les étapes pour chaque seconde de chaque flux) est devenue prépondérante, une structure qui faisait transiter ces données sur le réseau à de nombreuses reprises n'était plus adaptée, et une autre structure convenait mieux. Un bon architecte se demande ce dont la charge de travail, l'équipe et le budget ont besoin, et non ce qui est à la mode.

ATTENTION

Lire les études de cas avec soin

- Les récits publics simplifient. Utilisez les sources primaires (ordonnances des régulateurs, rapports d'audit, billets des ingénieurs eux-mêmes) plutôt que les récits de seconde main, qui exagèrent souvent ou ajoutent des détails.
- La plupart des échecs réels ont plusieurs causes : structure technique, processus, organisation et incitations. Accuser un ingénieur ou une ligne de code est presque toujours une erreur.
- Une décision qui a échoué dans un contexte peut être juste dans un autre. Demandez-vous quel était le contexte avant d'en tirer une règle générale.

POUR ALLER PLUS LOIN

Pourquoi l'architecture est difficile à justifier, et comment y parvenir

Les bénéfices d'une bonne architecture sont surtout des choses qui n'arrivent *pas* : la panne évitée, la réécriture inutile, l'équipe non bloquée. Il est donc facile de les sous-estimer. Les architectes efficaces traduisent les choix structurels en termes métier : délai de mise en production des changements, coût par transaction, heures d'indisponibilité par an, nombre d'équipes pouvant travailler en parallèle, ampleur du risque encouru. Le chapitre 11 revient sur la construction d'analyses de rentabilité (business cases) pour le travail architectural.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi un service bien écrit peut-il quand même faire partie d'un système peu fiable ?

Parce que la disponibilité naît de la façon dont les services dépendent les uns des autres. Une chaîne d'appels synchrones multiplie les probabilités de défaillance, si bien que l'ensemble peut être moins disponible que n'importe laquelle de ses parties.

▼ Qu'est-ce qui faisait de la défaillance de Knight Capital un problème architectural plutôt qu'un simple bug ?

Du code obsolète accessible, un flag ayant deux significations selon les versions, et un déploiement manuel sans contrôle automatique de la concordance de tous les serveurs : autant de décisions de structure et de mise en production, et non une seule ligne fautive.

▼ **Quelle est la principale leçon du cas Prime Video ?**

Qu'une architecture doit être adaptée à sa charge de travail et à son contexte. Une conception adaptée à une certaine échelle et à un certain profil de données peut devenir mal adaptée, et changer de structure peut transformer les coûts.

▼ **Pourquoi le coût de la remise en cause d'une décision augmente-t-il avec le temps ?**

Parce que de plus en plus de code, de données stockées, d'intégrations, de partenaires et d'habitudes en viennent à en dépendre : la changer signifie donc les changer tous.

▼ **À vous : Un nouveau service de billetterie stocke les prix en nombres à virgule flottante. Passer aux pence entiers aujourd'hui demande une journée à deux développeurs. Le responsable technique dit : « On le fera l'an prochain, ce n'est qu'un problème d'arrondi. » Quel est votre argument, et à qui l'adressez-vous ?**

Corrigez-le maintenant. Aujourd'hui, rien ne dépend du format ; dans un an, il se trouvera dans les réservations stockées, les rapports, les intégrations et le code d'autres équipes, et le même changement exigera donc une migration de données, des mises en production coordonnées et un rapprochement. Les premiers choix structurels fixent le coût de tout le travail ultérieur. Présentez-le au responsable technique et au responsable financier comme une question métier : une journée maintenant contre des semaines plus tard, alors que des totaux faux d'un penny continuent entre-temps d'agacer les comptables.

SOURCES

- SEC press release 2013-222: SEC charges Knight Capital with violations of market access rule: https://www.sec.gov/news/press-release/2013-222
- GAO-14-694: Healthcare.gov: Ineffective Planning and Oversight Practices Underscore the Need for Improved Contract Management: https://www.gao.gov/products/gao-14-694
- GAO-15-238: Healthcare.gov: CMS Has Taken Steps to Address Problems, but Needs to Further Implement Systems Development Best Practices: https://www.gao.gov/products/GAO-15-238
- Marcin Kolny: Scaling up the Prime Video audio/video monitoring service and reducing costs by 90% (Prime Video Tech, 2023; the original post, which may no longer be online): https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90
- InfoQ: Prime Video moves from serverless to ECS, cutting costs (report on the original post, 2023): https://infoq.com/news/2023/05/prime-ec2-ecs-saves-costs
- Design Stamina Hypothesis (Martin Fowler): https://martinfowler.com/bliki/DesignStaminaHypothesis.html

SECTION 1.4 SUR 1.8 · DÉBUTANT · 19 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Ce que fait un architecte : responsabilités, compétences et types d'architectes

L'IDÉE EN BREF

Sur un chantier, l'architecte ne pose pas chaque brique, mais il décide des choses qu'il serait ruineux de rater, les explique aux constructeurs et aux clients, vérifie sur place que les plans sont respectés, et s'adapte quand le terrain se révèle plus meuble que prévu. Un architecte logiciel fait de même pour un système logiciel : il prend ou oriente les grandes décisions, les explique, vérifie en permanence qu'elles sont respectées et qu'elles ont toujours du sens, et doit en savoir un peu sur beaucoup de technologies et beaucoup sur les personnes et le métier.

CE QU'IL FAUT SAVOIR

- Les architectes veillent à ce que les décisions significatives soient bien prises, comprises et respectées à mesure que le système évolue ; l'architecture est un rôle avant d'être un intitulé de poste.
- Richards et Ford décrivent huit attentes : prendre des décisions, analyser en continu, rester à jour, garantir la conformité, avoir une expérience variée, connaître le domaine métier, avoir des compétences relationnelles et savoir naviguer dans les jeux politiques.
- Le programme de l'iSAQB décrit ainsi les tâches des architectes : clarifier les exigences, concevoir les structures et les concepts transversaux, communiquer, accompagner la mise en œuvre et évaluer, de manière itérative.
- Parmi les types d'architectes figurent l'architecte d'entreprise, de solution, logiciel ou applicatif, des données, de la sécurité et du cloud ou de l'infrastructure ; les staff engineers et les principal engineers assument souvent les mêmes responsabilités.
- Les architectes privilégient l'étendue technique (connaître de nombreuses options et leurs compromis) plutôt qu'une expertise approfondie dans quelques domaines, comme le montre la pyramide de la connaissance.
- Les architectes devraient généralement rester impliqués dans le code sans être responsables de code sur le chemin critique : preuves de concept, fonctions d'aptitude, corrections de bugs et revues de code.

-

À quoi servent les architectes

Si l'architecture est l'ensemble des décisions et des structures significatives d'un système (section 1.1), l'architecte est quiconque veille à ce que ces décisions soient bien prises, comprises par les autres et respectées à mesure que le système évolue. Cela paraît simple, mais c'est l'un des rôles les moins uniformément définis du logiciel. Dans une entreprise, l'architecte est un ingénieur senior qui écrit encore du code tous les jours ; dans une autre, un membre d'une équipe centrale qui revoit les conceptions et tient à jour les normes ; dans une troisième, une personne qui parle surtout avec les dirigeants métier et les fournisseurs.

Derrière cette diversité se trouve un socle commun. Les architectes prennent ou orientent les décisions coûteuses à inverser, veillent aux qualités du système entier plutôt qu'à celles d'une seule partie, et relient les choix techniques aux objectifs métier. La suite de cette section décrit ce que cela implique, les compétences requises et la façon dont le rôle est organisé.

Huit attentes envers un architecte

Mark Richards et Neal Ford, dans *Fundamentals of Software Architecture* (O'Reilly, 1re édition 2020, 2e édition 2025), décrivent huit attentes essentielles envers un architecte, quels que soient l'intitulé du poste ou l'entreprise. Elles sont paraphrasées ci-dessous, avec pour chacune un exemple tiré d'une entreprise de livraison de repas.

Les huit attentes (d'après Richards et Ford)

Attente	Ce que cela signifie	Exemple
Prendre des décisions d'architecture	Définir les décisions et les principes qui guident les choix technologiques, généralement en orientant l'équipe plutôt qu'en imposant des produits précis.	Décider que les intégrations avec les restaurants passent par une seule couche d'adaptateurs, plutôt que de choisir la bibliothèque de chaque restaurant.
Analyser l'architecture en continu	Réévaluer si l'architecture convient toujours à mesure que la technologie et l'activité évoluent, et rechercher les signes de dégradation.	Remarquer que le service d'affectation des livreurs concentre désormais 40 % de tous les échecs de déploiement, et chercher pourquoi.
Rester à jour des tendances	Suivre les évolutions des technologies et des pratiques, car les décisions architecturales durent longtemps.	Comprendre les nouvelles offres de bases de données managées avant la prochaine revue de capacité.
Garantir la conformité aux décisions	Vérifier que les décisions et principes convenus sont réellement suivis, idéalement à l'aide de contrôles automatisés.	Ajouter au build un contrôle qui échoue lorsque l'app appelle directement le prestataire de paiement.
Avoir une exposition et une expérience variées	Connaître un large éventail de technologies, de frameworks et de plateformes suffisamment bien pour les comparer.	Comparer des courtiers de messages (message brokers) sur la garantie d'ordre, la durabilité et le coût d'exploitation.
Connaître le domaine métier	Comprendre suffisamment bien l'activité pour parler aux parties prenantes dans leurs propres termes.	Savoir comment fonctionnent les reversements aux restaurants et les remboursements avant de repenser les paiements.
Avoir des compétences relationnelles	Diriger, collaborer, encadrer et résoudre les désaccords ; l'essentiel du travail d'architecture passe par d'autres personnes.	Animer une séance où deux équipes se mettent d'accord sur un contrat d'événement.
Comprendre les jeux politiques et savoir y naviguer	Obtenir le soutien aux décisions dans une organisation où les budgets, les priorités et les responsabilités sont disputés.	Obtenir l'accord de la finance et de l'exploitation avant de retirer un ancien système de reporting.

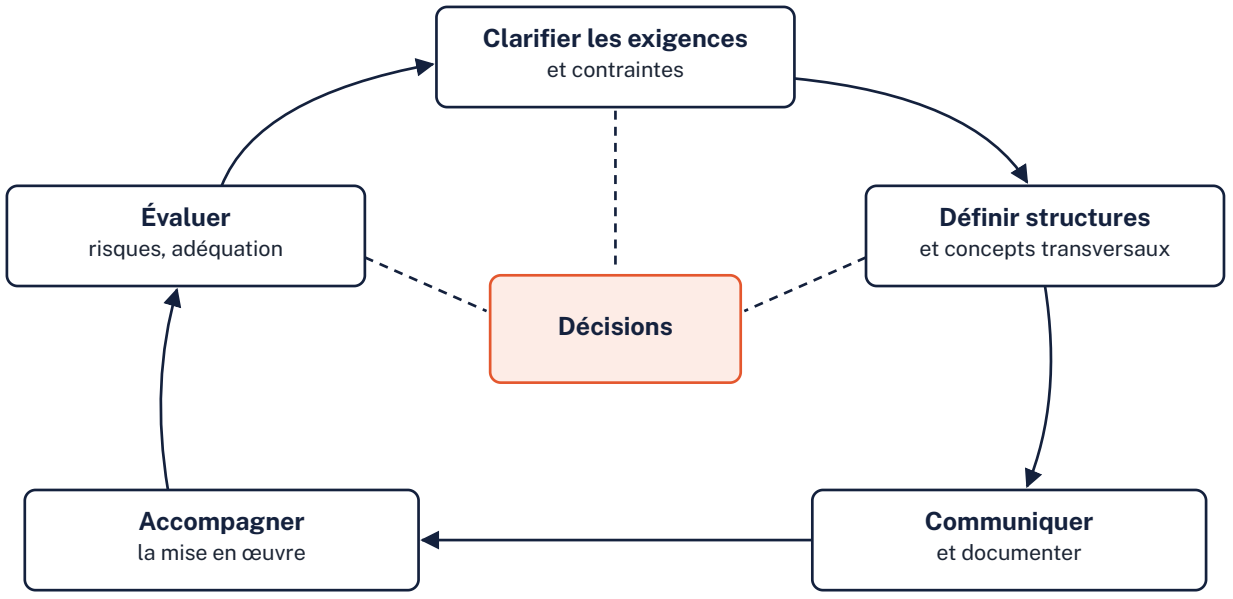
Remarquez combien peu de ces huit attentes sont purement techniques. Richards et Ford soulignent que presque chaque décision d'un architecte sera contestée par quelqu'un (développeurs, autres architectes, product owners, chefs de projet) : la négociation et la persuasion sont donc des compétences essentielles, pas des extras. Une architecture correcte sur le papier mais ni comprise ni acceptée ne sera pas construite comme prévu.

La vision des tâches de l'architecte selon l'iSAQB

L'International Software Architecture Qualification Board (iSAQB), dont le programme de niveau Foundation (CPSA-F) est largement utilisé en Europe, décrit les tâches et responsabilités des architectes de façon similaire. En résumé, les architectes : clarifient et examinent attentivement les exigences et les contraintes ; décident comment décomposer le système en blocs de construction avec leurs dépendances et leurs interfaces ; décident des préoccupations transversales (cross-cutting concerns) comme la persistance, la communication et l'interface utilisateur ; communiquent et documentent l'architecture ; accompagnent sa réalisation et sa mise en œuvre ; l'analysent et l'évaluent, notamment au regard des risques de ne pas satisfaire les exigences ; et expliquent les conséquences des décisions architecturales aux autres parties prenantes.

Le programme souligne aussi que ces tâches sont itératives : les architectes travaillent par cycles, en revenant sur les exigences et les décisions à mesure que les retours arrivent, plutôt que de concevoir une fois pour toutes puis de passer la main. La section 1.8 examine comment cela s'intègre à une livraison itérative et continue.

Le travail de l'architecte comme un cycle



Chaque tâche alimente et réexamine les décisions ; le cycle se répète toute la vie du système

D'après le programme de l'iSAQB : clarifier, concevoir, communiquer, accompagner et évaluer se répètent, et toutes ces tâches gravitent autour des décisions.

Types d'architectes

Les intitulés suivent les niveaux d'architecture décrits dans la section 1.2. Les responsabilités exactes varient fortement d'une organisation à l'autre : considérez le tableau ci-dessous comme typique plutôt que définitif.

Type	Périmètre	Livrables typiques
Architecte d'entreprise	L'organisation entière : capacités, portefeuille d'applications, normes technologiques, alignement sur la stratégie	Cartes des capacités, feuilles de route vers l'état cible, normes, conseils d'investissement
Architecte de solution	Un problème métier ou un programme couvrant plusieurs systèmes, fournisseurs et équipes	Conceptions de solution, conceptions d'intégration, évaluations de fournisseurs
Architecte logiciel ou applicatif	La structure et les qualités d'un système ou d'un groupe de systèmes étroitement liés	Structures de composants, enregistrements de décision, scénarios de qualité, orientations techniques pour les équipes
Architecte des données	La façon dont les données sont modélisées, détenues, stockées, déplacées, gouvernées et analysées	Modèles de données, cartes de responsabilité des données, conceptions de pipelines et de plateformes
Architecte sécurité	Identité, accès, protection des données et emplacement des contrôles de sécurité	Modèles de menaces, patrons de sécurité, exigences de contrôle
Architecte cloud ou infrastructure	Calcul, réseau, stockage, environnements et plateformes	Landing zones, conceptions réseau, normes de plateforme, modèles de coûts

Un rôle ou un intitulé de poste ?

L'architecture est un ensemble de responsabilités avant d'être un intitulé de poste. Dans de nombreuses organisations, en particulier les plus petites et celles qui sont organisées autour d'équipes produit autonomes, personne ne porte le titre d'« architecte ». Le travail architectural est pourtant fait, par les responsables techniques, les ingénieurs seniors ou l'équipe collectivement. Ce qui compte, c'est que quelqu'un soit clairement responsable des décisions significatives et des qualités à l'échelle du système. Quand personne ne l'est, une architecture est quand même produite, mais par accident.

De nombreuses entreprises technologiques proposent une filière de carrière technique pour les contributeurs individuels seniors, avec des niveaux comme staff engineer et principal engineer. Le livre *Staff Engineer* (2021) de Will Larson décrit quatre archétypes courants de ces rôles, dont l'un est explicitement appelé l'Architect : une personne responsable de l'orientation et de la qualité d'un domaine technique critique. Les autres (le Tech Lead, le Solver et le Right Hand) assument aussi une part importante de responsabilité architecturale. Dans ces organisations, le travail de l'architecte est fait par des personnes qui sont aussi ingénieurs.

L'article de Martin Fowler de 2003, *Who Needs an Architect?*, oppose deux caricatures : un architecte qui prend seul toutes les décisions importantes, et un autre qui travaille étroitement avec l'équipe, encadre les développeurs et les aide à prendre eux-mêmes de bonnes décisions. Fowler plaidait résolument pour le second. La plupart des praticiens s'accordent aujourd'hui à dire que les architectes qui dictent à distance tendent à produire des conceptions que les équipes contournent discrètement.

L'intégrité conceptuelle : pourquoi quelqu'un doit tenir l'ensemble

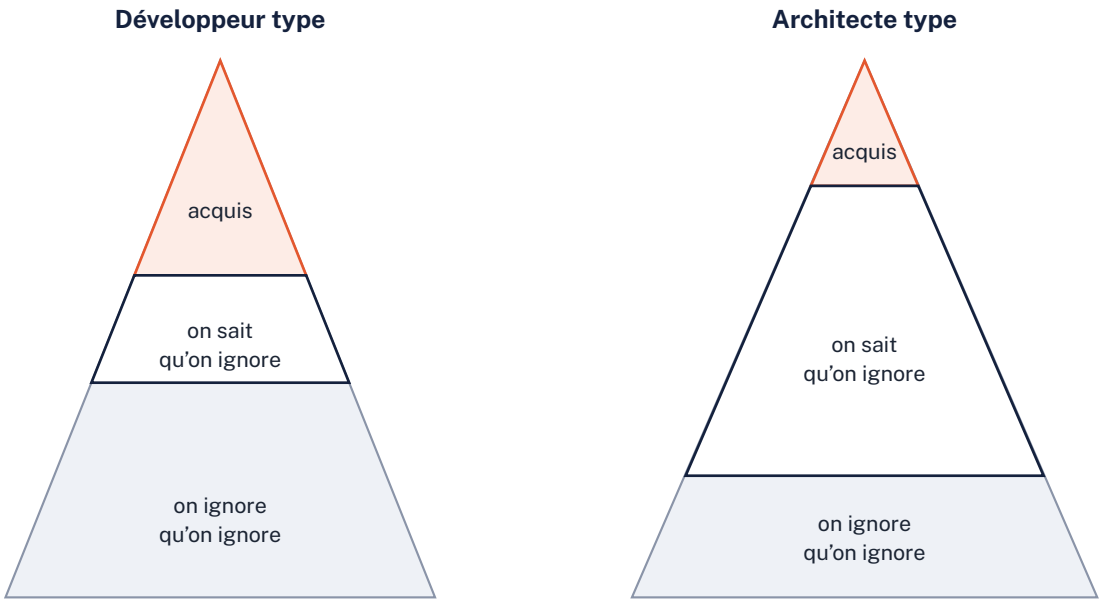
Fred Brooks a donné au rôle sa justification classique dans *The Mythical Man-Month* (1975) : « Je soutiendrai que l'intégrité conceptuelle est la considération la plus importante dans la conception d'un système. » Un système devrait donner l'impression d'avoir été conçu par un seul esprit, avec un seul ensemble d'idées appliqué de façon cohérente, même si cela oblige à laisser de côté de bonnes fonctionnalités sans rapport avec lui. Il le comparait à l'unité de la cathédrale de Reims, dont les bâtisseurs ont renoncé à certaines de leurs propres idées au profit de l'ensemble. Son remède consistait à séparer l'architecture de l'implémentation, et il acceptait l'accusation d'aristocratie. Les équipes modernes partagent ce travail plus largement, mais le besoin demeure : si la plateforme de livraison de repas a trois façons de traiter un remboursement, quelqu'un devrait le remarquer et s'en soucier.

L'étendue technique face à la profondeur

Les développeurs sont généralement récompensés pour leur profondeur : connaître extrêmement bien un langage, un framework ou une plateforme. Les architectes ont besoin d'un profil différent. Richards et Ford décrivent une pyramide de la connaissance à trois niveaux : ce que vous savez (là où vous avez une réelle expertise), ce que vous savez ne pas savoir (vous savez que cela existe et à peu près à quoi cela sert, mais vous ne pourriez pas vous en servir pour construire quelque chose aujourd'hui), et ce que vous ne savez pas que vous ne savez pas. L'expertise du niveau supérieur doit être entretenue, ce qui prend du temps. Le niveau intermédiaire est peu coûteux à développer, et c'est lui qui vous permet de reconnaître les options.

La valeur d'un architecte réside en grande partie dans son étendue technique (technical breadth) : savoir qu'il existe cinq façons de résoudre un problème, ce en quoi chacune est bonne ou mauvaise, et laquelle mérite un examen plus approfondi. Un architecte qui ne connaît qu'une seule base de données la recommandera à chaque fois. L'étendue a un prix, car le temps passé à élargir le niveau intermédiaire n'est pas passé à approfondir le niveau supérieur ; les architectes laissent donc délibérément s'estomper certaines expertises anciennes et investissent dans la connaissance d'un plus grand nombre de choses. Richards et Ford mettent en garde contre l'échec inverse : l'architecte dont les opinions se sont forgées sur une mauvaise expérience vécue il y a des années et qui rejette toute option qui lui ressemble.

La pyramide de la connaissance



La profondeur (haut) compte encore, mais c'est l'étendue (milieu) qui permet de comparer les options

D'après Richards et Ford : un architecte élargit délibérément le niveau intermédiaire, celui des choses dont on sait qu'elles existent sans les maîtriser.

Les architectes doivent-ils écrire du code ?

Les praticiens sont réellement en désaccord sur ce point, et ce désaccord est utile, car chaque camp a raison sur quelque chose.

Les arguments pour. Les architectes qui codent comprennent le coût réel de leurs décisions, remarquent quand un principe est pénible en pratique, gardent leur jugement technique à jour et conservent le respect des développeurs qui doivent mettre en œuvre leurs conceptions. Les architectes qui cessent de coder pendant des années risquent de concevoir des systèmes élégants sur un tableau blanc et malcommodes dans un éditeur.

Les arguments contre. Le temps d'un architecte est réparti entre de nombreuses équipes, réunions et décisions. Un architecte responsable de fonctionnalités sur le chemin critique devient un goulot d'étranglement : l'équipe attend du code retardé par ses autres obligations. Son attention est aussi nécessaire sur les préoccupations inter-équipes dont aucune équipe ne s'occupera seule.

La solution courante. La plupart des voix expérimentées, dont Richards et Ford, recommandent que les architectes restent impliqués dans le code d'une manière qui ne les place pas sur le chemin critique. Par exemple : construire une preuve de concept (proof of concept) pour tester une option risquée avant que l'équipe ne s'y engage ; écrire des fonctions d'aptitude (fitness functions), des contrôles automatisés qui vérifient des règles architecturales comme « le module de domaine ne doit pas dépendre du framework web » ; corriger des bugs et traiter de petits éléments de dette technique ; construire des outils et de l'automatisation pour l'équipe ; et relire régulièrement du code, ce qui montre comment les décisions sont réellement mises en œuvre.

```
import pathlib, re

FORBIDDEN = re.compile(r"^\s*(from|import)\s+flask\b", re.M)

def test_domain_does_not_depend_on_web_framework():
    offenders = []
    for path in pathlib.Path("shop/domain").rglob("*.py"):
        if FORBIDDEN.search(path.read_text(encoding="utf-8")):
            offenders.append(str(path))
    assert not offenders, f"Domain code imports Flask: {offenders}"
```

Une fonction d'aptitude simple qu'un architecte pourrait écrire : faire échouer le build si un module de domaine importe le framework web.

Une semaine dans la vie d’un architecte logiciel (exemple illustratif)

Jour	Activités principales	Attente mise en œuvre
Lundi	Planification avec deux équipes produit ; constat qu’une nouvelle fonctionnalité de fidélité a besoin de données que le service des commandes n’expose pas ; ébauche d’options	Prendre des décisions ; connaître le domaine métier
Mardi	Atelier avec le délégué à la protection des données et l’équipe sécurité sur le stockage des adresses des clients ; rédaction d’un enregistrement de décision	Compétences relationnelles ; garantir la conformité
Mercredi	Preuve de concept de deux jours d’un service managé de streaming d’événements pour tester les garanties d’ordre sous charge	Expérience variée ; rester à jour
Jeudi	Revues de code dans l’équipe des paiements ; ajout d’une fonction d’aptitude après avoir repéré une dépendance interdite ; mentorat individuel	Garantir la conformité ; compétences relationnelles
Vendredi	Analyse des rapports d’incident et des métriques de délai de livraison pour repérer des signes de dégradation architecturale ; réunion avec la finance sur les coûts du cloud ; lecture	Analyser en continu ; naviguer dans les jeux politiques ; rester à jour

EXEMPLE COMMENTÉ

Exemple commenté : une décision, cinq tâches

Le système de prise de rendez-vous d'un hôpital doit envoyer des rappels par SMS. L'architecte logiciel passe en revue les tâches de l'iSAQB pour ce seul besoin.

Clarifier les exigences	Demande combien de rappels par jour (environ 20 000), dans quel délai ils doivent partir, ce qui se passe si le fournisseur de SMS est en panne, et si les messages peuvent contenir des détails cliniques (ils ne le peuvent pas, pour des raisons de confidentialité).
Conception	Propose que le service de réservation publie des événements « rendez-vous réservé » et qu'un composant de notification distinct envoie les messages à partir d'une file d'attente, pour qu'une panne de SMS ne bloque jamais les réservations.
Communiquer	Rédige un enregistrement de décision d'une page avec les options envisagées (appel direct ou file d'attente) et les raisons ; le présente à l'équipe et au responsable de la gouvernance de l'information.
Accompagner la mise en œuvre	Travaille en binôme avec un développeur sur la première version, relit la logique de nouvelle tentative et ajoute une fonction d'aptitude qui empêche les champs cliniques d'apparaître dans le modèle de message.
Évaluer	Trois mois plus tard, vérifie les délais d'acheminement et les taux d'échec ; constate que les rappels restent 30 minutes en file d'attente pendant une panne du fournisseur et confirme avec les cliniciens que c'est acceptable.

L'essentiel de l'effort de l'architecte a porté sur la clarification, la communication et la vérification, pas sur le dessin de la conception.

La décision a été consignée avec ses raisons, pour qu'une future équipe puisse la réexaminer si le contexte change.

Questions fréquentes

Comment devenir architecte ?

Généralement en faisant du travail d'architecture avant d'en avoir le titre : prendre en charge les décisions transversales de votre équipe, rédiger des enregistrements de décision, élargir délibérément votre étendue technique, apprendre le domaine métier et vous exercer à expliquer les compromis à des personnes non techniques. Des certifications comme le CPSA-F de l'iSAQB peuvent structurer cet apprentissage, mais ne remplacent pas l'expérience.

Chaque équipe a-t-elle besoin d'un architecte ?
Chaque équipe a besoin que la responsabilité architecturale soit clairement attribuée, mais pas nécessairement à une personne dédiée. Les petites équipes la confient souvent à un responsable technique ; les grandes organisations peuvent avoir des architectes qui accompagnent chacun plusieurs équipes.

Quelle est la différence entre un responsable technique et un architecte ?
Un responsable technique est généralement responsable de la livraison technique d'une équipe et passe une grande partie de son temps à coder ; un architecte est généralement responsable des décisions et des qualités qui couvrent plusieurs équipes ou systèmes. Les rôles se recoupent fortement, et dans de nombreuses organisations le responsable technique est l'architecte du système de son équipe.

Combien de temps un architecte doit-il passer en réunion ?
Il n'existe pas de chiffre juste, mais un bon critère consiste à se demander si les réunions produisent des décisions, une compréhension partagée ou la levée d'obstacles. L'image de Gregor Hohpe, celle de l'architecte qui prend l'ascenseur entre la salle du conseil et la salle des machines, rappelle que les deux extrémités doivent être visitées régulièrement.

ATTENTION

Les anti-patrons du rôle d'architecte

- **La tour d'ivoire :** des conceptions imposées d'en haut sans implication dans la mise en œuvre, que les équipes ignorent ou contournent.
- **Le goulot d'étranglement :** chaque décision, même minime, doit passer par l'architecte, si bien que la livraison ralentit et que les équipes cessent de réfléchir.
- **L'expert figé :** un jugement ancré dans des technologies ou de mauvaises expériences datant de plusieurs années.
- **L'architecture pilotée par le CV :** choisir des technologies parce qu'elles intéressent l'architecte plutôt que parce qu'elles conviennent au problème.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi Richards et Ford incluent-ils les jeux politiques parmi les attentes envers un architecte ?

Parce que presque chaque décision architecturale sera contestée par quelqu'un, et qu'une décision incapable d'obtenir un soutien face aux enjeux de budgets, de priorités et d'équipes ne sera pas mise en œuvre comme prévu.

▼ Quel niveau de la pyramide de la connaissance un architecte doit-il délibérément développer, et pourquoi ?

Le niveau intermédiaire, ce que vous savez ne pas savoir, car savoir que des options existent et à quoi elles servent permet à un architecte de comparer des solutions.

▼ Citez deux façons dont un architecte peut rester impliqué dans le code sans devenir un goulot d'étranglement.

Deux au choix parmi : construire des preuves de concept, écrire des fonctions d'aptitude, corriger des bugs ou de petits éléments de dette, construire des outils et de l'automatisation, et relire du code.

▼ À vous : Une architecte d'une plateforme de livraison de repas, la meilleure programmeuse présente, propose d'écrire elle-même le module central de paiement parce qu'il est trop risqué pour quiconque d'autre. Est-ce un bon usage du rôle ? Que lui conseilleriez-vous de faire à la place ?

Non. Être propriétaire du code du chemin critique fait d'elle un goulet d'étranglement, et lorsque le travail d'architecture l'accapare, l'équipe reste bloquée sur le module le plus important. Elle doit rester impliquée dans le code d'autres manières : construire une preuve de concept de la partie risquée, relire le code, ajouter des fonctions d'aptitude qui protègent les règles et corriger quelques bugs. Un développeur solide prend alors le module en charge, ce qui diffuse les connaissances et la laisse en mesure d'assumer la responsabilité des décisions plus larges.

SOURCES

- iSAQB CPSA-Foundation curriculum (LG 01-04: tasks and responsibilities of software architects): https://public.isaqb.org/curriculum-foundation/curriculum-foundation-en.html
- Who Needs an Architect? Martin Fowler, IEEE Software, 2003 (PDF): https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf
- Staff archetypes (Will Larson, StaffEng): https://staffeng.com/guides/staff-archetypes/
- The Architect Elevator (Gregor Hohpe): https://architectelevator.com/

- Brooks: The Mythical Man-Month, chapter 4 (sample chapter): https://ptgmedia.pearsoncmg.com/images/0201835959/samplechapter/chap4.html

SECTION 1.5 SUR 1.8 · DÉBUTANT · 20 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Les parties prenantes, leurs préoccupations et comment l'architecture y répond

L'IDÉE EN BREF

Pensez à la construction d'une maison. La famille qui va y vivre veut de l'espace et de la lumière. La banque qui prête l'argent veut que la maison vaille le prêt. Le service d'urbanisme veut qu'elle s'intègre à la rue. Le constructeur veut des plans réellement réalisables, et les voisins ne veulent pas de bruit la nuit. Tous sont des parties prenantes, et ce qui inquiète chacun d'eux est une préoccupation. Un système logiciel a lui aussi ses parties prenantes : les utilisateurs, ceux qui paient, les développeurs, les personnes qui le font fonctionner, les autorités de régulation. L'architecture est l'art de les écouter toutes et de trancher ouvertement là où leurs souhaits se heurtent.

CE QU'IL FAUT SAVOIR

- Une partie prenante est tout rôle, toute personne, toute organisation ou toute catégorie de ceux-ci ayant un intérêt, un droit, une part ou une revendication dans un système, y compris des personnes qui ne l'utilisent jamais directement, comme les autorités de régulation et les exploitants.
- Une préoccupation, au sens de l'ISO/IEC/IEEE 42010, est toute question pertinente ou importante pour une partie prenante : qualités, coûts, risques, contraintes, faisabilité et questions d'organisation, tout compte.
- Les préoccupations ne sont pas encore des exigences : consignez-les dans les mots de la partie prenante, puis affinez-les en scénarios d'attributs de qualité mesurables, en contraintes ou en exigences fonctionnelles.
- La matrice pouvoir/intérêt aide à planifier l'implication de chaque partie prenante, mais elle ne dit rien de l'importance relative de leurs préoccupations ; des groupes qui ont peu de pouvoir peuvent avoir le plus à perdre.
- Les préoccupations contradictoires sont normales ; l'architecte les rend explicites, chiffre les options, désigne qui décide et consigne la justification.
- Les préoccupations structurent la documentation d'architecture : chaque point de vue encadre certaines préoccupations, et chaque vue montre comment le système y répond.

Qui sont les parties prenantes

Une partie prenante est toute personne qui a un intérêt légitime dans un système : celles qui l'utilisent, le paient, le construisent, l'exploitent, le réglementent ou sont touchées par lui. L'ISO/IEC/IEEE 42010 :2022, la norme internationale de description d'architecture, définit largement ce terme comme un rôle, une fonction, un individu, une organisation ou une catégorie de ceux-ci ayant un intérêt, un droit, une part ou une revendication dans la chose décrite. Deux caractéristiques de cette définition comptent en pratique. D'abord, une partie prenante est souvent une *catégorie* plutôt qu'une personne : « les patients » ou « les ingénieurs d'astreinte » plutôt qu'un individu nommé. Ensuite, une revendication ou un droit suffit : une autorité de régulation qui ne se connecte jamais à votre système reste une partie prenante, parce qu'elle peut vous empêcher d'exercer.

La liste est plus longue que la plupart des équipes ne le pensent. Pour un système commercial typique, elle comprend :

Les utilisateurs (les personnes qui se servent du logiciel au quotidien, souvent plusieurs groupes distincts) ; **les clients** (ceux qui paient, qui ne sont pas forcément utilisateurs) ; le **product owner** ou chef de produit, qui décide de ce qui est construit ; le **commanditaire**, qui finance le travail et répond de son succès ; **les développeurs**, qui doivent comprendre et modifier le code ; **les testeurs**, qui doivent pouvoir préparer des données et observer le comportement ; **les équipes d'exploitation et les ingénieurs SRE (site reliability engineering)**, qui le déploient, le supervisent et le rétablissent à trois heures du matin ; **l'équipe sécurité** ; le **délégué à la protection des données (DPD)** lorsque des données personnelles sont traitées (en vertu du RGPD britannique et du RGPD de l'UE, certains organismes doivent en désigner un) ; **les services juridique et achats**, attentifs aux licences, aux contrats et à la responsabilité ; **la finance**, attentive aux coûts d'exploitation autant qu'aux coûts de construction ; **les autorités de régulation et les auditeurs** ; et **les intégrateurs tiers**, les autres équipes et entreprises dont le logiciel appelle le vôtre ou est appelé par lui.

Certaines parties prenantes sont absentes de toutes les réunions : les développeurs qui maintiendront le système dans cinq ans, le service d'assistance qui reçoit les plaintes, les personnes en situation de handicap qui utiliseront l'interface et les membres du public touchés par ses décisions. Une partie du travail de l'architecte consiste à parler au nom de ceux qui ne sont pas dans la salle.

Les préoccupations : ce qui compte pour les parties prenantes

Dans la 42010, une préoccupation est toute question pertinente ou importante pour une partie prenante. Le mot est volontairement large. Une préoccupation peut être une qualité (« la page de réservation restera-t-elle rapide le lundi matin ? »), un coût (« combien cela coûtera-t-il à

exploiter par mois ? »), un risque (« que se passe-t-il si le fournisseur de SMS tombe en panne ? »), une contrainte (« les données des patients doivent-elles rester au Royaume-Uni ? »), une question de faisabilité (« notre équipe peut-elle construire cela avec les compétences dont elle dispose ? ») ou une question d'organisation (« quelle équipe est responsable de l'intégration avec le système des médecins généralistes ? »).

Les préoccupations ne sont pas encore des exigences. Elles sont généralement formulées de façon vague, dans le vocabulaire propre de la partie prenante, et ce sont souvent des inquiétudes plutôt que des spécifications. « Je ne veux pas d'une autre panne comme l'hiver dernier » est une préoccupation ; « le service de réservation est disponible 99,9 % de chaque mois civil, mesuré au niveau de l'équilibreur de charge » est une exigence qui en découle. Consigner d'abord la préoccupation dans les mots de la partie prenante, et seulement ensuite l'affiner, vous oblige à rester honnête sur l'origine d'une exigence et sur la raison de son importance.

Une même préoccupation est souvent partagée par plusieurs parties prenantes, et une partie prenante a généralement de nombreuses préoccupations. La sécurité est une préoccupation de l'équipe sécurité, du délégué à la protection des données, du commanditaire (qui craint les gros titres) et des patients (qui craignent une fuite de leur dossier). C'est précisément à cause de cette relation plusieurs-à-plusieurs que la 42010 fait des préoccupations la charnière entre les personnes et l'architecture : les décisions répondent aux préoccupations, et les vues s'organisent autour des préoccupations qu'elles traitent.

Parties prenantes typiques et leurs préoccupations

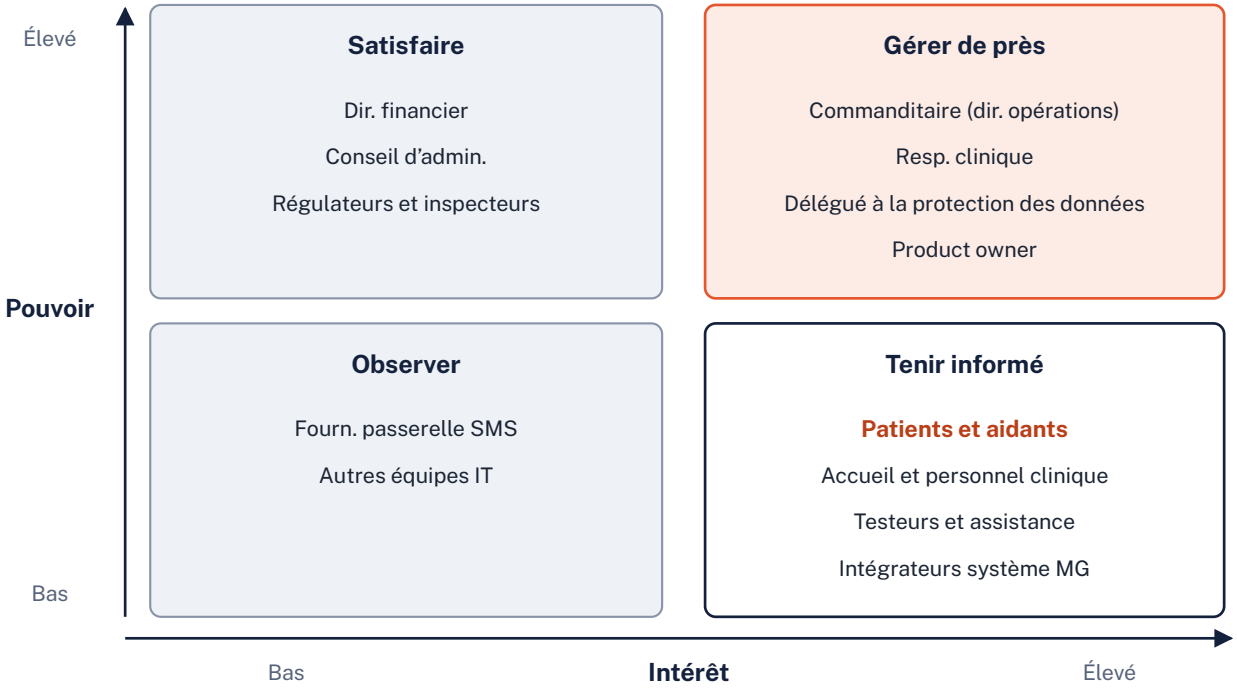
Partie prenante	Préoccupations typiques	Ce que l'architecture doit leur apporter
Usagers	Rapidité, facilité d'utilisation, accessibilité, un système disponible quand on en a besoin	Objectifs de temps de réponse et de disponibilité ; une couche d'interface capable de respecter les normes d'accessibilité
Clients et commanditaires	Valeur métier, délai de mise sur le marché, coût total, risque pour la réputation	Un plan de livraison crédible, un modèle de coûts et une liste des principaux risques
Product owner	Vitesse d'ajout de nouvelles fonctionnalités, maintien des options ouvertes	Une structure modulaire alignée sur le domaine métier ; peu de décisions difficiles à inverser
Développeurs	Compréhensibilité, rapidité du build et des tests, responsabilités claires	Des frontières de modules nettes, un build local rapide, des conventions documentées
Testeurs	Testabilité, données de test maîtrisables, comportement observable	Des points de jonction où remplacer les dépendances ; des environnements de test proches de la production
Exploitation et SRE	Déployabilité, supervision, rétablissement, capacité, charge d'astreinte	Contrôles de santé, journaux, métriques et traces ; retour arrière sûr ; runbooks
Équipe sécurité	Surface d'attaque, identité, secrets, réponse aux vulnérabilités	Frontières de confiance, conception de l'authentification, stratégie de correctifs et de dépendances
Délégué à la protection des données	Base légale, minimisation des données, conservation, droits des personnes concernées, transferts	Un inventaire des données et une vue des flux de données ; des mécanismes d'effacement et d'export
Juridique et achats	Licences, contrats, responsabilité, dépendance au fournisseur	Une liste des composants et services tiers avec leurs conditions
Finance	Coût de construction, coût d'exploitation, prévisibilité des coûts	Un modèle de coûts par environnement et par unité d'utilisation
Autorités de régulation et auditeurs	Conformité aux règles du secteur, preuves de contrôle	Pistes d'audit, traçabilité de l'exigence au test, décisions documentées
Intégrateurs tiers	Interfaces stables, versionnage, documentation, environnements de test (sandbox)	Des contrats d'API publiés et une politique de versionnage

Cartographier les parties prenantes : la matrice pouvoir/intérêt

Avec vingt groupes de parties prenantes ou plus, vous ne pouvez pas accorder à chacun la même attention. Un outil très utilisé est la matrice pouvoir/intérêt, généralement attribuée à Aubrey Mendelow, dont l'article sur les parties prenantes et la veille environnementale date de 1981 (il est souvent cité comme datant de 1991). Chaque partie prenante est placée selon le pouvoir qu'elle a sur le système (le financer, l'approuver, le bloquer ou le remodeler) et l'intérêt qu'elle lui porte. Les quadrants suggèrent une stratégie d'implication : **gérer de près** (pouvoir élevé, intérêt élevé), **satisfaire** (pouvoir élevé, intérêt faible), **tenir informé** (pouvoir faible, intérêt élevé) et **observer** (pouvoir faible, intérêt faible).

La matrice porte sur l'implication, pas sur la question de savoir quelles préoccupations comptent. Les patients d'un hôpital ont peu de pouvoir formel sur la conception d'un système de réservation, mais ce sont eux qui ont le plus à perdre, et certaines obligations légales (accessibilité, protection des données) existent précisément pour protéger les personnes qui ont peu de pouvoir. Voyez la matrice comme un rappel des personnes à consulter et de la fréquence, et formez-vous un jugement distinct sur les préoccupations critiques. Les positions bougent aussi : une autorité de régulation peu intéressée devient la partie prenante la plus intéressée de toutes la semaine qui suit une violation de données.

Une matrice pouvoir/intérêt pour un système de réservation hospitalier



Les patients se trouvent dans le quadrant du pouvoir faible, mais ce sont eux qui ont le plus à perdre : c'est pourquoi la matrice guide l'implication et non les priorités.

Une analyse légère des parties prenantes

1. **Identifier**
   Dressez la liste des catégories de parties prenantes avec le commanditaire et l'équipe, puis vérifiez-la par rapport à un ensemble type (utilisateurs, payeurs, bâtisseurs, exploitants, protecteurs, autorités de régulation, intégrateurs). Demandez à chaque groupe identifié : à qui d'autre devrions-nous parler ?

2. **Recueillir les préoccupations**
   Menez des entretiens ou des ateliers avec chaque groupe important. Consignez les préoccupations dans leurs propres mots, puis notez la qualité, la contrainte ou le risque qui se cache derrière chacune.

3. **Cartographier et hiérarchiser**
   Placez les parties prenantes sur la matrice pouvoir/intérêt pour planifier leur implication. Séparément, classez les préoccupations selon leur impact métier et selon la force avec laquelle elles contraignent l'architecture.

4. **Rendre les conflits explicites**
   Lorsque deux préoccupations tirent dans des directions opposées, mettez le conflit par écrit, nommez les personnes qui doivent se mettre d'accord et convenez de qui décide.

5. **Tracer et revisiter**
   Reliez chaque décision importante aux préoccupations qu'elle traite, et refaites l'analyse aux grands jalons, car les parties prenantes et leurs préoccupations changent.

Les préoccupations contradictoires et comment les rendre explicites

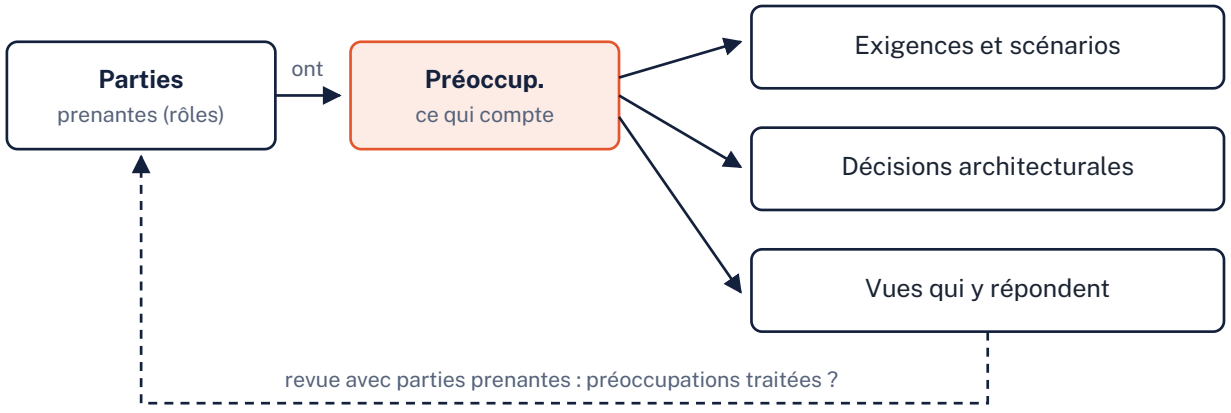
Les préoccupations entrent sans cesse en conflit, et une architecture qui plaît à tout le monde signifie généralement qu'un conflit a été caché plutôt que résolu. L'équipe sécurité veut que chaque connexion soit protégée par une authentification multifacteur ; la représentante des patients fait remarquer que de nombreux patients âgés n'ont pas de smartphone. La finance veut un seul petit serveur de base de données ; l'exploitation veut une réplique de secours dans un second centre de données. Le product owner veut une nouvelle fonctionnalité chaque semaine ; les testeurs veulent du temps pour faire les tests de non-régression de l'intégration avec le système d'administration des patients de l'hôpital.

La contribution de l'architecte n'est pas de désigner discrètement un gagnant, mais de rendre le conflit visible, chiffré et assumé par quelqu'un. En pratique, cela signifie quatre choses. **Formuler précisément les deux préoccupations**, idéalement sous forme de scénarios testables, pour que le débat porte sur des chiffres plutôt que sur des adjectifs. **Présenter côte à côte les options et leurs conséquences** : par exemple, la MFA pour le personnel et pour les

patients qui la choisissent, avec en solution de repli un code à usage unique envoyé par SMS ou par appel sur une ligne fixe. **Nommer la personne qui décide** : certains compromis relèvent du commanditaire ou d'un comité des risques, pas de l'architecte. **Consigner la décision et sa justification**, pour que, lorsque quelqu'un demandera dans deux ans pourquoi les patients peuvent se connecter sans MFA, la réponse existe. La section suivante, Décisions et compromis, explique comment peser de telles options.

Certains conflits se dissipent quand on les examine de près. L'équipe sécurité n'a peut-être pas besoin de la MFA pour toutes les actions des patients, seulement pour la consultation des résultats d'examens ; le groupe de patients peut très bien l'accepter à cet endroit. Demander à chaque camp *pourquoi* il a cette préoccupation, et quel résultat il craint vraiment, révèle souvent une conception qui satisfait les deux.

Où vont les préoccupations



Les préoccupations font le lien entre les personnes et l'architecture : elles deviennent des exigences, orientent les décisions et déterminent les vues que vous dessinez.

Comment les préoccupations alimentent les exigences et la documentation

Les préoccupations sont la matière première de deux activités ultérieures. La première est l'ingénierie des exigences. Une préoccupation de rapidité devient une exigence d'attribut de qualité, que le chapitre 2 vous apprend à rédiger sous forme de scénario mesurable (source, stimulus, artefact, environnement, réponse et mesure de la réponse) : voir Rédiger des scénarios d'attributs de qualité. Une préoccupation comme « les données des patients doivent rester au Royaume-Uni » devient une *contrainte*, une décision déjà prise à votre place. Les préoccupations qui s'avèrent sans effet sur la structure du système restent de simples exigences fonctionnelles ou exigences de projet.

La seconde est la documentation. La 42010 organise une description d'architecture autour des préoccupations : un point de vue nomme les préoccupations qu'il encadre et les parties prenantes qu'il sert, et une vue applique ce point de vue à votre système. Les préoccupations de l'équipe d'exploitation appellent une vue de déploiement montrant les serveurs, les régions et le basculement ; celles du délégué à la protection des données appellent une vue des flux de données montrant où les données personnelles circulent et où elles sont stockées ; les développeurs ont besoin d'une vue des modules montrant ce qui dépend de quoi. Un diagramme qui ne répond à la préoccupation d'aucune partie prenante n'est que de la décoration. Le chapitre 4 traite ce sujet en détail dans Vues et points de vue.

La traçabilité relie les deux. Si vous pouvez suivre un fil depuis une partie prenante, en passant par une préoccupation, jusqu'à une exigence, une décision et une vue, vous pouvez répondre aux deux questions que tout relecteur finit par poser : pourquoi est-ce construit ainsi, et avons-nous oublié quelqu'un ?

EXEMPLE COMMENTÉ

Exemple commenté : les parties prenantes d'un système hospitalier de prise de rendez-vous

Un groupement hospitalier régional veut un système en ligne permettant aux patients de prendre, modifier et annuler leurs rendez-vous de consultation externe, en remplacement de la prise de rendez-vous par téléphone. Il doit s'intégrer au système d'administration des patients (PAS) existant et envoyer des rappels par SMS. Voici la première analyse des parties prenantes qu'un architecte pourrait produire au cours des deux premières semaines.

Patients et aidants	Préoccupations : réserver facilement sur un téléphone, accessibilité (lecteurs d'écran, grands caractères), rappels, confidentialité. Impact sur l'architecture : une interface web adaptative (responsive) conforme au niveau AA des WCAG 2.2 ; aucune app à installer ; un minimum de données affichées à l'écran.
Accueil et personnel clinique	Préoccupations : pas de doubles réservations, des listes de consultations conformes à la réalité, moins d'appels téléphoniques. Impact : le PAS reste la source unique de vérité pour les créneaux ; le système de réservation ne doit pas en détenir sa propre copie, qui pourrait diverger.
Resp. clinique	Préoccupations : les demandes urgentes ne doivent pas être évincées par les rendez-vous de routine ; sécurité clinique. Impact : des règles de créneaux gérées par les cliniciens et configurables sans nouvelle mise en production ; un dossier de sécurité clinique (en Angleterre, selon la norme DCB0129) couvrant le système.
Délégué à la protection des données	Préoccupations : les données de santé relèvent des catégories particulières de données au sens du RGPD britannique (UK GDPR) ; minimisation ; conservation ; audit. Impact : une vue des flux de données, le chiffrement en transit et au repos, des journaux d'audit indiquant qui a consulté quoi, une politique de conservation.
Équipe d'exploitation	Préoccupations : les pics du lundi matin, la capacité limitée du PAS, le support en dehors des heures ouvrées. Impact : une file d'attente ou un limiteur de débit devant le PAS ; une mise en cache de la disponibilité des créneaux avec une expiration courte ; une supervision des temps de réponse du PAS.
Dir. financier	Préoccupations : budget fixe, coût d'exploitation prévisible, économies sur le personnel du centre d'appels. Impact : des services cloud managés dimensionnés pour le pic ; un modèle de coûts pour mille réservations.
Fournisseur du PAS (intégrateur)	Préoccupations : aucune charge non prise en charge sur son système ; utilisation de sa seule interface publiée. Impact : une intégration via l'interface du fournisseur avec des limites de débit convenues ; une suite de tests de contrat.

Principal conflit identifié	L'exploitation veut mettre en cache la disponibilité des créneaux pour protéger le PAS ; le personnel d'accueil craint des doubles réservations dues à des données périmées. Solution proposée : un cache pour l'affichage seulement (30 secondes), mais une confirmation de chaque réservation auprès du PAS en temps réel, avec un message clair si le créneau vient d'être pris.

Sept groupes de parties prenantes ont produit au moins cinq décisions aux conséquences structurelles (source unique de vérité, limitation de débit, journalisation d'audit, règles configurables, interface accessible) avant qu'aucune fonctionnalité n'ait été conçue en détail.

Le conflit le plus lourd de conséquences opposait deux groupes discrets, l'exploitation et le personnel d'accueil, dont aucun n'aurait dominé un comité de pilotage.

Chaque ligne peut être reliée, en aval, à un scénario d'attribut de qualité, à un enregistrement de décision et à une vue, et en amont à une catégorie de personnes nommée.

ATTENTION

Erreurs courantes

Voici les façons dont le travail sur les parties prenantes dérape le plus souvent.

- Traiter « le métier » comme une seule partie prenante. Les ventes, la finance, la conformité et le service client veulent des choses différentes, souvent contradictoires.
- Oublier les exploitants. Les systèmes conçus uniquement autour des utilisateurs et des développeurs sont régulièrement difficiles à déployer, à superviser et à rétablir.
- Confondre bruit et importance. La personne présente à toutes les réunions n'est pas forcément celle dont la préoccupation fera couler le projet.
- Se fier à des intermédiaires. Un product owner peut représenter les utilisateurs, mais ne peut pas toujours vous dire ce que vit réellement un utilisateur ayant des besoins d'accessibilité ou une infirmière de nuit. Parlez à quelques personnes réelles.
- Ne faire l'analyse qu'une fois. Des parties prenantes arrivent et partent, les réglementations changent, et un nouveau partenaire d'intégration apporte de nouvelles préoccupations. Revoyez la liste à chaque grand jalon.

Questions fréquentes

L'équipe de développement est-elle vraiment une partie prenante ? C'est nous qui le construisons.

Oui. Les développeurs vivent avec l'architecture plus longtemps que presque tout le monde, et leurs préoccupations (temps de build, testabilité, responsabilités claires) se traduisent directement en vitesse de livraison. La définition de la 42010 ne fait aucune distinction entre ceux qui construisent et ceux qui utilisent.

Quelle est la différence entre une préoccupation d'une partie prenante et une exigence ?

Une préoccupation est l'intérêt sous-jacent, souvent vague (« je veux que ce soit fiable »). Une exigence est un énoncé précis et vérifiable, dérivé d'une ou de plusieurs préoccupations (« 99,9 % de disponibilité mensuelle »). Plusieurs exigences peuvent servir une même préoccupation, et certaines préoccupations deviennent plutôt des contraintes.

Qui décide quand deux parties prenantes puissantes ne sont pas d'accord ?

En général, pas l'architecte seul. L'architecte formule les options et leurs conséquences ; la décision revient à la personne responsable du résultat, typiquement le commanditaire ou une instance de gouvernance. Convenir à l'avance de qui tranche quel type de compromis épargne des semaines de discussions.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Citez trois parties prenantes d'un système commercial typique qui assistent rarement aux réunions de conception.

Par exemple : les développeurs qui le maintiendront dans les années à venir, le service d'assistance, les utilisateurs en situation de handicap, les équipes d'exploitation d'astreinte la nuit, et les membres du public touchés par ses décisions.

▼ Un commanditaire dit : « Il faut que ce soit sécurisé. » Est-ce une préoccupation ou une exigence ?

Une préoccupation. Elle doit être affinée en exigences précises et testables, par exemple sur l'authentification, le chiffrement ou la journalisation d'audit, avant de pouvoir guider les décisions.

▼ **Pourquoi une même préoccupation, comme la sécurité, peut-elle appartenir à plusieurs parties prenantes à la fois ?**

Parce que des personnes différentes craignent des conséquences différentes d'une même qualité : l'équipe sécurité craint les attaques, le délégué à la protection des données craint les violations de données personnelles, le commanditaire craint l'atteinte à la réputation et les utilisateurs craignent la fuite de leurs données.

▼ **Qu'est-ce qui relie une préoccupation d'une partie prenante à une vue d'architecture ?**

Un point de vue : il indique quelles préoccupations et quelles parties prenantes il sert et comment les modéliser, et la vue l'applique au système.

▼ **À vous : Le marketing, puissant et motivé, veut un plan de salle animé pour le jour de la mise en vente. Un petit groupe de personnes handicapées indique que les lecteurs d'écran ne peuvent pas l'utiliser. Un responsable place les deux sur la matrice pouvoir/intérêt et déclare que le marketing l'emporte. Critiquez ce raisonnement et dites ce qu'il faut faire.**

La grille aide seulement à prévoir comment associer les personnes ; elle ne dit rien de la préoccupation qui compte le plus, et les groupes peu influents peuvent avoir le plus à perdre. Ici, certains clients ne pouvaient plus acheter du tout. Consignez les deux préoccupations dans les mots de leurs auteurs, puis rendez le conflit explicite : par exemple une liste utilisable au clavier et au lecteur d'écran à côté du plan, avec son coût. Chiffrez les options, nommez qui décide et consignez le raisonnement, y compris toute obligation d'accessibilité.

SOURCES

- ISO/IEC/IEEE 42010:2022 Software, systems and enterprise: Architecture description (ISO): https://www.iso.org/standard/74393.html
- ISO/IEC/IEEE 42010 summary (arc42 quality model): https://quality.arc42.org/standards/iso-42010
- Stakeholder analysis for UX projects (Nielsen Norman Group): https://www.nngroup.com/articles/stakeholder-analysis/
- arc42 section 3: Context and scope: https://docs.arc42.org/section-3/

Décisions architecturales et compromis : réversibilité et moment opportun

AVANT DE LIRE

Cette section s'appuie sur :

- 1.2 Architecture ou conception
- 1.5 Parties prenantes et préoccupations

Termes déjà vus qu'elle utilise : Partie prenante, Préoccupation, Architecturalement significatif

L'IDÉE EN BREF

Choisir une voiture est un compromis : un moteur plus gros consomme plus de carburant, une voiture plus petite se gare plus facilement mais transporte moins. Personne n'obtient tout. Certains choix sont faciles à défaire, comme une couleur de peinture que vous pourrez changer plus tard ; d'autres, comme acheter la maison qui donne sur une autoroute, sont très difficiles à défaire. L'architecture logicielle est pleine de choix de ce genre. Les bons architectes font rapidement les choix faciles à défaire, prennent soin de ceux qui sont difficiles à défaire, décident au bon moment plutôt que trop tôt ou trop tard, et notent pourquoi ils ont choisi, pour que ceux qui viendront après eux comprennent.

CE QU'IL FAUT SAVOIR

- Tout choix architectural important est un compromis : il améliore certaines qualités au détriment d'autres, et ses inconvénients doivent être identifiés et énoncés.
- Une décision bien formée consigne son contexte, les options envisagées, les critères, la décision avec sa justification, les conséquences et son statut.
- La réversibilité est la propriété clé d'une décision : les portes à double sens doivent être tranchées vite et localement, les portes à sens unique de façon réfléchie et avec les personnes qui en subiront les conséquences.
- Le dernier moment responsable est le point à partir duquel attendre supprimerait une option importante ; décider à ce moment-là permet de disposer du maximum de connaissances sans laisser la décision se prendre par défaut.
- Les abstractions aux frontières, les feature flags, la migration incrémentale et les spikes rendent les décisions plus réversibles, à un coût qui ne vaut la peine que lorsque l'incertitude et le coût d'un retour en arrière sont tous deux élevés.

> - Les matrices de décision pondérées et les enregistrements de décision d'architecture rendent le raisonnement visible et contestable ; convenez des critères et des pondérations avant de noter, et testez la sensibilité du résultat à ces choix.

Tout est compromis

Tout choix architectural important achète quelque chose en renonçant à autre chose. Découper un système en services déployables indépendamment achète l'autonomie des équipes et une mise à l'échelle indépendante, et se paie en latence réseau, en complexité opérationnelle et en cohérence des données plus difficile. Le cache achète de la vitesse et se paie en données périmées et en logique d'invalidation. Une base de données managée dans le cloud libère des correctifs et des sauvegardes, et se paie en coût et en dépendance envers un seul fournisseur. Mark Richards et Neal Ford placent cette idée en tête de *Fundamentals of Software Architecture* (2020 ; deuxième édition 2025), comme leur première loi : «Tout, en architecture logicielle, est un compromis.» Ils en tirent un corollaire : si vous pensez avoir trouvé quelque chose qui n'est pas un compromis, c'est probablement que vous n'avez pas encore identifié le compromis.

Leur deuxième loi dit que le *pourquoi* est plus important que le *comment*. N'importe qui peut regarder un système en fonctionnement et voir comment il est construit : trois services, un courtier de messages (message broker), une base de données orientée documents. Ce que le code ne permet pas de retrouver, c'est le pourquoi : quelles alternatives ont été envisagées, quelles préoccupations l'ont emporté, quels risques ont été acceptés. Sans le pourquoi, les ingénieurs suivants soit conservent des choix qui n'ont plus de sens, soit défont des choix dont les raisons restent valables. La deuxième édition ajoute une troisième loi : la plupart des décisions architecturales ne sont pas binaires, mais se situent sur un continuum entre des extrêmes. Vous choisissez rarement entre « monolithe » et « microservices » ; vous choisissez combien d'unités déployables, découpées selon quelles lignes, partageant quoi.

Penser en compromis est l'habitude qui distingue le plus nettement la réflexion architecturale de l'enthousiasme. Quand quelqu'un propose une technologie, les questions utiles ne sont pas « est-ce bien ? » mais « que coûte-t-elle, que rend-elle plus difficile, et ces coûts sont-ils acceptables au regard de nos préoccupations ? ». La section précédente a montré d'où viennent ces préoccupations.

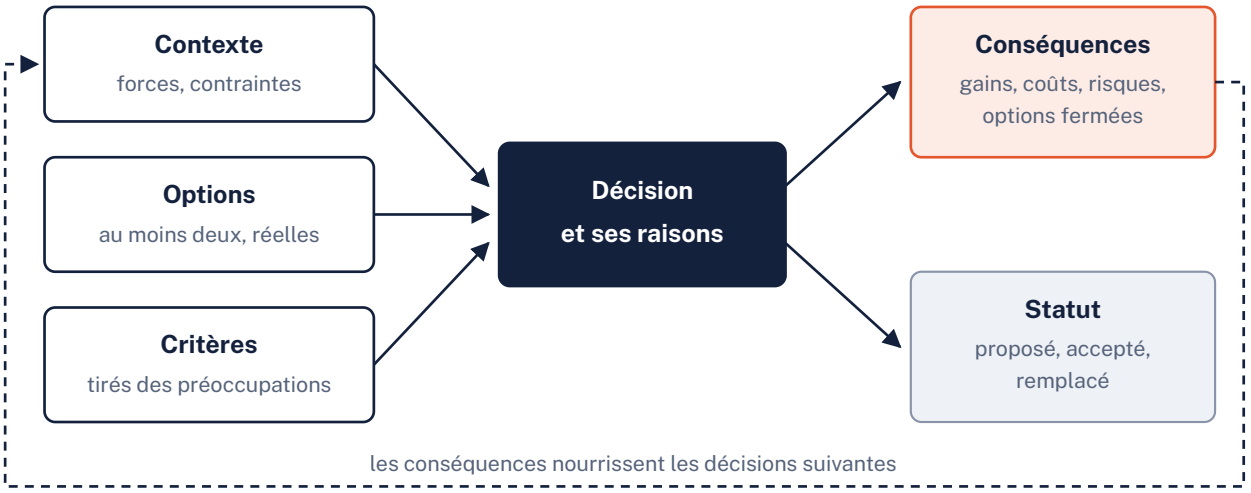
L'anatomie d'une décision

Une décision architecturale est plus que son résultat. Une décision que l'on peut expliquer, examiner et réexaminer comporte six parties, et les nommer est la première étape pour bien décider.

Contexte : les forces en présence. Le problème à résoudre, les exigences d'attributs de qualité qui s'y rapportent, les contraintes (budget, échéances, réglementations, systèmes existants, compétences de l'équipe) et les hypothèses retenues. **Options** : au moins deux alternatives réelles, y compris souvent « ne rien faire » ou « reporter ». Une décision pour laquelle une seule option a été envisagée est une annonce, pas une décision. **Critères** : la façon dont les options seront jugées, tirée des préoccupations des parties prenantes et pondérée selon leur importance. **Décision et justification** : l'option retenue et, surtout, la raison pour laquelle elle l'a emporté sur les autres au regard des critères. **Conséquences** : ce qui devient plus facile, ce qui devient plus difficile, quels nouveaux risques apparaissent et quelles options futures sont désormais fermées. **Statut** : proposée, acceptée, obsolète ou remplacée, avec qui a pris la décision et quand.

Les conséquences méritent une attention particulière, car c'est là que l'honnêteté est mise à l'épreuve. Toute option a des inconvénients ; les mettre par écrit protège l'équipe des surprises et donne à la personne qui prendra la décision suivante un compte rendu loyal. Les conséquences d'une décision font aussi partie du contexte de la suivante : une fois que vous avez choisi la messagerie asynchrone, vous devez décider comment traiter les messages en double et les messages qui arrivent dans le désordre.

Anatomie d'une décision architecturale



Le contexte, les options et les critères alimentent une décision ; ses conséquences et son statut sont consignés et deviennent le contexte des décisions suivantes.

Portes à sens unique et portes à double sens

La propriété la plus utile d'une décision, à elle seule, est sa réversibilité : le coût et la sûreté avec lesquels on peut la défaire. Dans sa lettre de 2015 aux actionnaires d'Amazon, Jeff Bezos distinguait les décisions de **type 1**, lourdes de conséquences et irréversibles ou presque (« portes à sens unique »), des décisions de **type 2**, modifiables et réversibles (« portes à double

sens »). Les décisions de type 1 méritent une délibération lente et soigneuse, avec consultation. Les décisions de type 2 doivent être prises rapidement par des individus ou de petits groupes dotés d'un bon jugement, car si elles s'avèrent mauvaises, on peut repasser la porte dans l'autre sens. Il avertissait que les grandes organisations ont tendance à appliquer à tout le processus lourd du type 1, ce qui les rend lentes et frileuses.

L'idée se transpose directement à l'architecture. En logiciel, une porte à sens unique est une décision qui, pour être défaite, exigerait de réécrire de grandes parties du système, de migrer de grandes quantités de données, de renégocier des contrats ou de reformer de nombreuses personnes : le langage de programmation principal d'une vaste base de code, le modèle de données d'un système de référence, l'API publique avec laquelle des centaines de partenaires s'intègrent, un engagement cloud sur plusieurs années. Parmi les portes à double sens figurent le choix d'une bibliothèque de journalisation placée derrière votre propre interface, la structure interne d'un module, le réglage d'un cache, ou le choix entre deux formats de message équivalents à l'intérieur d'un service.

Deux mises en garde. D'abord, la réversibilité est un continuum, pas une alternative binaire : la plupart des décisions se situent quelque part entre « trivial à défaire » et « impossible à défaire », et le coût d'un retour en arrière augmente généralement à mesure que davantage de code, de données et de personnes dépendent du choix. Une décision qui est une porte à double sens le premier mois peut devenir une porte à sens unique au douzième. Ensuite, vous pouvez souvent changer le type de porte face auquel vous vous trouvez, par la conception, comme le montre plus loin la section sur la réversibilité.

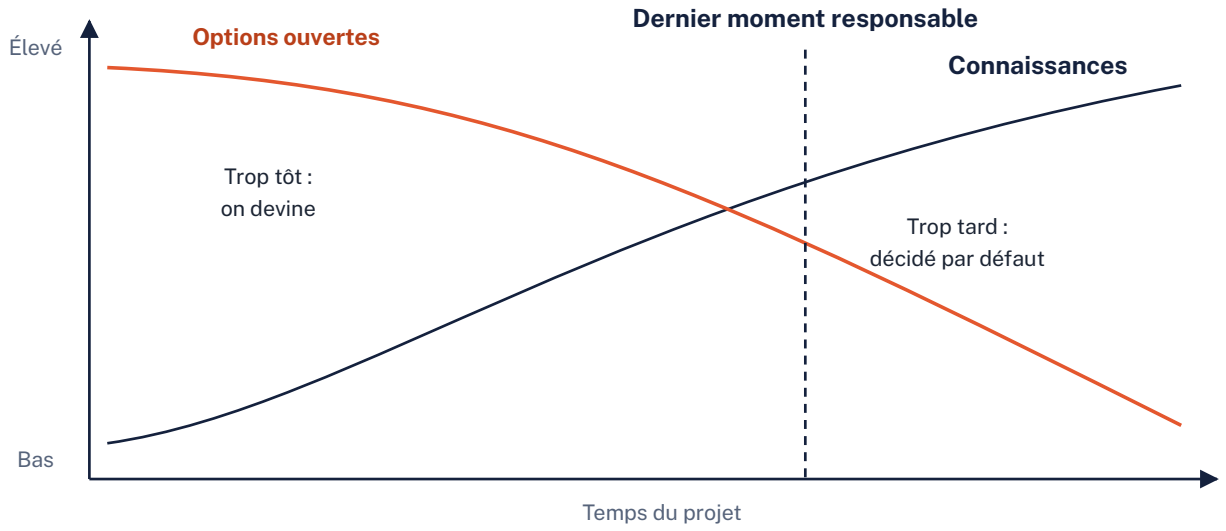
Le moment : le dernier moment responsable

Quand faut-il prendre une décision ? Mary et Tom Poppendieck, dans *Lean Software Development: An Agile Toolkit* (2003), ont popularisé l'idée de décider au dernier moment responsable : le moment où ne pas décider éliminerait une alternative importante. Avant ce moment, vous décidez avec moins d'informations que vous ne pourriez en avoir ; après, la décision est prise à votre place, par défaut, et généralement mal.

Le principe vaut dans les deux sens. Il plaide contre une décision trop précoce : figer la technologie de base de données la première semaine, avant de comprendre la forme des données et les schémas d'accès, revient à deviner. Il plaide tout autant contre une décision trop tardive. Si l'équipe a commencé à écrire du code de persistance en s'appuyant sur une hypothèse jamais formulée, la décision a déjà été prise, simplement pas consciemment. Si le service achats a besoin de six semaines pour approuver un fournisseur cloud, le dernier moment responsable pour ce choix se situe au moins six semaines avant que vous ayez besoin de l'environnement.

Les praticiens ne s'accordent pas sur la limite à ne pas franchir. Les critiques font valoir que le « dernier moment responsable » sert facilement de prétexte à la procrastination, que certaines décisions (structure des équipes, langage de programmation) façonnent tout ce qui suit et doivent donc être prises tôt, et que l'indécision a son propre coût : les équipes ne peuvent pas travailler en parallèle sans interfaces convenues. Les partisans répondent que le principe n'a jamais voulu dire « le plus tard possible » mais « aussi tard que c'est responsable », et qu'identifier ce moment est en soi un acte de planification. Les deux positions s'accordent sur le noyau pratique : sachez quelles décisions sont en attente, sachez quand chacune doit être prise, et utilisez le temps qui reste pour apprendre.

Décider trop tôt, trop tard et au bon moment



Les connaissances augmentent à mesure que le projet avance, tandis que les options ouvertes se ferment ; le dernier moment responsable est le dernier point avant qu'une option importante ne disparaisse.

SI VOUS DÉBUTEZ

Pas à pas : quand choisir la base de données du service des commandes

L'idée la plus difficile de cette section est le dernier moment responsable : une décision ne doit être prise ni le plus tôt ni le plus tard possible, mais au dernier point avant qu'attendre ne supprime une option importante. Suivez une décision dans le temps : la base de données du nouveau service des commandes de l'entreprise de livraison de repas, évaluée plus loin dans l'exemple commenté de cette section.

- Première semaine. L'équipe pourrait choisir une base de données tout de suite, mais elle ne sait pas encore comment les commandes seront enregistrées et lues. Choisir maintenant serait deviner : décider trop tôt.
- L'équipe attend donc, et met ce temps à profit pour apprendre. Elle découvre par exemple que chaque commande et son paiement doivent être enregistrés ensemble, en tout ou rien, en une seule transaction : exactement le genre de fait sur lequel doit reposer le choix d'une base de données.
- L'attente a cependant une échéance. Supposons que toute nouvelle base de stockage exige une revue de sécurité de trois semaines, et que la base de données doive fonctionner pour les tests en semaine neuf. Le dernier moment responsable pour ce choix est donc, au plus tard, la semaine six.
- Il existe aussi une échéance plus discrète. Si les développeurs commencent à écrire le code qui enregistre les commandes, et que ce code suppose discrètement une base de données particulière, le choix a été fait par défaut : personne ne l'a décidé, mais le changer maintenant oblige à réécrire ce code.
- L'équipe nomme donc la décision, lui attribue un responsable et une date antérieure aux deux échéances, décide à ce moment-là avec ce qu'elle a appris, et consigne les hypothèses sur lesquelles elle repose.
- **La leçon :** le dernier moment responsable ne signifie pas « le plus tard possible ». Sachez quelles décisions sont en attente et quand chacune doit être prise, utilisez le temps qui reste pour apprendre, et décidez avant que le choix ne soit fait à votre place.

Concevoir pour la réversibilité

La façon la moins coûteuse de prendre une décision difficile est souvent de la transformer en une décision plus facile. Plusieurs techniques réduisent le coût d'un changement d'avis ultérieur, au prix d'un certain effort aujourd'hui.

L'abstraction à la frontière. Placez une interface étroite qui vous appartient entre votre logique centrale et une technologie que vous pourriez remplacer, comme un prestataire de paiement, un courtier de messages ou l'API d'un grand modèle de langage. Remplacer la technologie plus

tard ne touchera alors qu'un adaptateur, et non toute la base de code. L'idée remonte au masquage de l'information (information hiding) de David Parnas (1972) et réapparaît dans l'architecture hexagonale (chapitre 5). **Les feature flags (indicateurs de fonctionnalité).** Livrez le nouveau comportement désactivé, activez-le pour une petite part des utilisateurs, et désactivez-le de nouveau en quelques secondes s'il se comporte mal ; les mises en production deviennent ainsi des portes à double sens. **La migration incrémentale.** Faites tourner l'ancien et le nouveau côte à côte et basculez le trafic progressivement plutôt qu'en une seule étape. **Les spikes et les preuves de concept.** Passez quelques jours à construire la partie la plus risquée avec une technologie candidate avant de vous y engager. **Les sorties contractuelles.** Préférez des engagements plus courts, des formats de données standard et la possibilité d'exporter vos données.

C'est le *raisonnement en options*, emprunté à la finance : vous payez une petite prime aujourd'hui (un adaptateur, un flag, un spike) pour acheter le droit, mais pas l'obligation, de changer de cap plus tard. Comme toute option, elle ne vaut la peine d'être achetée que si la prime est faible par rapport à la valeur de la flexibilité. Envelopper chaque bibliothèque dans une abstraction « au cas où nous la changerions » est une façon classique d'ajouter du coût et de l'indirection pour une flexibilité dont personne ne se sert. Achetez des options sur les décisions à la fois incertaines et coûteuses à inverser ; pour tout le reste, décidez et passez à la suite.

GUIDE DE DÉCISION

Guide de décision : décider maintenant ou reporter ?

Utilisez-le pour toute décision susceptible d'être architecturalement significative. Parcourez les questions dans l'ordre.

Q1 **D'autres travaux sont-ils bloqués, ou une option importante va-t-elle bientôt disparaître si personne ne décide (par exemple un délai d'achat, ou une équipe qui commence à construire sur une hypothèse) ?**

- **Oui** Vous êtes au dernier moment responsable. Décidez maintenant avec les informations dont vous disposez, consignez la décision et ses hypothèses, et fixez une date pour la réexaminer.
- **Non** Passez à la question suivante.

Q2 **La décision est-elle peu coûteuse et sûre à inverser (une porte à double sens) ?**

- **Oui** Décidez rapidement, au niveau pertinent le plus bas (souvent l'équipe responsable du code), consignez-la brièvement et passez à la suite. N'y consacrez pas une semaine de débat.
- **Non** C'est une porte à sens unique. Passez à la question suivante.

Q3 **Pouvez-vous la rendre plus réversible à peu de frais, par exemple avec une interface autour de la technologie, un feature flag ou un chemin de migration incrémentale ?**

- **Oui** Achetez d'abord cette option, puis traitez la décision restante comme une porte à double sens.
- **Non** Passez à la question suivante.

Q4 **Des informations que vous pouvez raisonnablement obtenir avant l'échéance (un spike, un test de charge, une conversation avec une partie prenante) changeraient-elles le choix ?**

- **Oui** Reportez délibérément : nommez la décision, son responsable, la date à laquelle elle doit être prise et l'expérience qui l'éclairera.
- **Non** Attendre davantage n'aidera pas. Lancez dès maintenant une comparaison structurée (critères, pondérations, options), impliquez les parties prenantes qui assument les conséquences, et rédigez un enregistrement de décision d'architecture (ADR).

Les pièges cognitifs

Les architectes sont humains, et les décisions architecturales se prennent dans l'incertitude, sous la pression du temps et sous la pression sociale : des conditions idéales pour les biais cognitifs. Connaître les plus courants par leur nom permet de mieux les repérer chez soi et de les contester avec bienveillance chez les autres.

Les pièges qui faussent les décisions architecturales

Piège	À quoi il ressemble	Parade
Ancrage	La première option mentionnée, ou la première estimation donnée, devient le point de référence auquel toutes les autres options sont comparées.	Générez plusieurs options indépendamment avant d'en discuter aucune ; demandez à chacun de noter ses estimations avant de les partager.
Coût irrécupérable	« Nous avons passé huit mois sur ce framework, nous ne pouvons pas l'abandonner maintenant », alors que la dépense passée est perdue quoi que vous décidiez.	Jugez les options uniquement sur leurs coûts et bénéfices futurs ; demandez-vous ce que vous choisiriez si vous partiez de zéro aujourd'hui.
Pensée de groupe	Une équipe soudée converge vite, le désaccord paraît déloyal, et personne n'exprime l'objection évidente.	Chargez quelqu'un de défendre la position contraire ; recueillez les avis par écrit avant la réunion ; invitez un relecteur extérieur.
Développement piloté par le CV	Une technologie est choisie parce qu'elle est passionnante à apprendre ou fait bien sur un CV, et non parce qu'elle convient au problème.	Exigez que chaque choix soit justifié au regard des critères convenus ; prévoyez un budget distinct pour le temps d'apprentissage.
Effet de mode et suivisme	« Tout le monde passe à X » tient lieu d'analyse ; des conférences de très grandes entreprises sont appliquées à de petites équipes.	Demandez quel problème X a résolu pour ceux qui l'ont adopté, et si vous avez ce problème à cette échelle.
Biais de confirmation	Les éléments en faveur de l'option préférée sont recherchés et retenus ; les éléments contraires sont écartés par des explications.	Notez à l'avance quel résultat vous ferait changer d'avis, puis lancez le spike ou le test.

EXEMPLE COMMENTÉ

Exemple commenté : une matrice de décision pondérée

Une entreprise de livraison de repas construit un nouveau service des commandes. Les commandes impliquent un paiement et ne doivent jamais être perdues ni débitées deux fois. L'équipe de six personnes connaît bien PostgreSQL et n'a pas d'administrateur de bases de données dédié. Trois options sont sur la table : (A) un service PostgreSQL managé de leur fournisseur cloud, (B) la base de données orientée documents managée et propriétaire du fournisseur, et (C) PostgreSQL auto-hébergé sur des machines virtuelles. L'équipe construit une matrice de décision pondérée : elle convient de critères et de pondérations à partir des préoccupations des parties prenantes (5 = le plus important), puis note chaque option de 1 (médiocre) à 5 (excellent).

Intégrité transactionnelle (poids 5)	A : 5, B : 3, C : 5. Les transactions sur plusieurs lignes sont au cœur des commandes et des paiements ; la base orientée documents les prend en charge avec davantage de restrictions.
Effort opérationnel (poids 4)	A : 4, B : 5, C : 2. L'auto-hébergement implique correctifs, sauvegardes et basculement assurés par une équipe sans spécialiste des bases de données.
Familiarité de l'équipe (poids 3)	A: 5, B: 2, C: 4.
Coût à la charge prévue (poids 3)	A: 3, B: 4, C: 4.
Marge de capacité (poids 2)	A : 3, B : 5, C : 3. La croissance attendue est d'environ trois fois en deux ans, largement à la portée d'une seule grande instance PostgreSQL.
Réversibilité et dépendance au fournisseur (poids 2)	A : 4, B : 2, C : 5. Un PostgreSQL standard peut passer d'un fournisseur à l'autre ; l'API propriétaire, non.
Totaux pondérés (maximum 95)	A: 25 + 16 + 15 + 9 + 6 + 8 = **79**. B: 15 + 20 + 6 + 12 + 10 + 4 = **67**. C: 25 + 8 + 12 + 12 + 6 + 10 = **73**.
Test de sensibilité	Portez la marge de capacité au poids 5 (comme si l'on attendait une croissance d'un facteur cinquante) : A 88, B 82, C 82. Abaissez l'effort opérationnel au poids 1 (comme si une équipe plateforme gérait les bases de données pour eux) : A 67, B 52, C 67, soit une égalité entre A et C.
Décision	Option A, PostgreSQL managé. Elle l'emporte avec les pondérations convenues et reste première ou ex aequo sous des variations plausibles. Conséquences consignées : une facture mensuelle plus élevée qu'en auto-hébergement ; une dépendance aux fenêtres de maintenance du fournisseur ; un plan pour garder le schéma et les requêtes standard afin que l'option C reste une solution de repli.

Les chiffres ne prennent pas la décision ; ils rendent le raisonnement visible. Quiconque n'est pas d'accord peut désigner une pondération ou une note précise au lieu de contester l'ensemble.

Le test de sensibilité est l'étape la plus instructive : il montre de quelles hypothèses dépend le résultat. Ici, l'issue tient surtout au manque de capacité de l'équipe à exploiter des bases de données.

Les matrices peuvent être manipulées en choisissant les critères et les pondérations une fois que l'option favorite est connue. Convenez des critères et des pondérations avant de noter, idéalement avec les parties prenantes dont ils représentent les préoccupations.

Consigner les décisions : un aperçu des ADR

La deuxième loi dit que le pourquoi compte plus que le comment ; le pourquoi doit donc être mis par écrit. Le format léger le plus utilisé est l'enregistrement de décision d'architecture (ADR), proposé par Michael Nygard dans un billet de blog de 2011. Un ADR est un court fichier texte, conservé avec le code dans la gestion de versions, qui consigne une décision selon l'anatomie décrite plus haut : titre, statut, contexte, décision et conséquences. Les ADR ne sont jamais modifiés pour réécrire l'histoire ; une décision ultérieure remplace une décision antérieure et y renvoie, de sorte que le journal raconte comment l'architecture a évolué.

Le chapitre 4 présente les modèles, les outils et les habitudes qui gardent les ADR utiles, dans Enregistrements de décision d'architecture (ADR). L'exemple ci-dessous consigne la décision de l'exemple commenté.

```
ADR 7: Use managed PostgreSQL for the order service

Status: Accepted (2026-10-12)

Context
Orders and payments need multi-row transactions and must never be lost.
The team knows PostgreSQL well and has no database administrator.
Expected growth is about 3x in two years.

Decision
We will use the cloud provider's managed PostgreSQL service.

Consequences
+ Backups, patching and failover are handled by the provider.
+ Standard SQL keeps self-hosting or another provider open as a fallback.
- Monthly cost is about 30% higher than self-hosting at current load.
- Maintenance windows are set by the provider.
Revisit if write volume exceeds what one primary instance can handle.
```

Un ADR court pour la décision sur la base de données du service des commandes, selon la structure de Nygard

ATTENTION

Erreurs courantes

- Débattre longuement des portes à double sens tout en franchissant des portes à sens unique au passage (« on va juste utiliser la table clients partagée pour l'instant »).
- N'envisager qu'une seule option. Si vous ne pouvez pas citer une alternative crédible, vous n'avez pas compris la décision.
- Cacher les inconvénients. Des conséquences qui ne listent que des avantages finiront par être démasquées, et la confiance dans l'architecte disparaîtra avec elles.
- Décider selon l'ancienneté ou selon qui parle le plus fort, plutôt que selon des critères convenus à l'avance.
- Ne jamais réexaminer. Les décisions reposent sur des hypothèses ; consignez-les, et réexaminez la décision quand une hypothèse change.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Quelle est la différence entre la première et la deuxième loi de Richards et Ford ?**

La première dit que tout choix architectural est un compromis. La deuxième dit que le raisonnement derrière un choix (le pourquoi) compte plus que sa mise en œuvre (le comment), parce que le pourquoi ne peut pas être retrouvé à partir du code.

▼ **Donnez un exemple de porte à double sens et un exemple de porte à sens unique dans une application web typique.**

À double sens : la bibliothèque de journalisation utilisée derrière votre propre interface, ou le réglage d'expiration d'un cache. À sens unique : le modèle de données d'un système de référence, ou une API publique dont dépendent déjà de nombreux partenaires.

▼ **Pourquoi décider trop tôt peut-il être aussi nuisible que décider trop tard ?**

Trop tôt signifie décider avec moins d'informations que vous n'auriez pu en avoir, souvent en devinant. Trop tard signifie que la décision est en fait prise par défaut, à travers des hypothèses déjà intégrées au code ou aux plans, ou qu'une option a expiré.

▼ **Qu'apporte un test de sensibilité à une matrice de décision pondérée ?**

Il montre si l'option gagnante gagne encore quand les pondérations ou les notes varient dans des plages plausibles, ce qui révèle de quelles hypothèses dépend la décision.

▼ **À vous : Pondérations : fiabilité 5, compétences de l'équipe 3, coût 2. La base de données A obtient 4, 5, 3 ; la base de données B obtient 5, 2, 4. Laquelle gagne ? Prenez maintenant les pondérations 5, 2, 3. Qu'est-ce que cela vous apprend sur la décision ?**

Avec 5, 3, 2 : A = 20 + 15 + 6 = 41 et B = 25 + 6 + 8 = 39, donc A l'emporte de peu. Avec 5, 2, 3 : A = 20 + 10 + 9 = 39 et B = 25 + 4 + 12 = 41, donc B l'emporte. Le résultat est sensible aux pondérations : la matrice ne tranche donc pas seule. Convenez des pondérations avant de noter, puis discutez de l'importance réelle des compétences de l'équipe, et consignez ce jugement.

SOURCES

- Documenting Architecture Decisions, Michael Nygard (2011): https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions
- Architectural Decision Records (adr.github.io): https://adr.github.io/
- Amazon.com 2015 letter to shareholders (SEC filing): https://www.sec.gov/Archives/edgar/data/1018724/000119312516530910/d168744dex991.htm
- Feature Toggles (aka Feature Flags), Pete Hodgson on martinfowler.com: https://martinfowler.com/articles/feature-toggles.html

Dette technique, érosion et dérive de l'architecture

AVANT DE LIRE

Cette section s'appuie sur :

- 0.3 Organiser le code
- 0.8 Comment les logiciels sont construits
- 1.3 Pourquoi c'est important

Termes déjà vus qu'elle utilise : Module, Dépendance, Coût du changement, Gestion de versions

L'IDÉE EN BREF

Imaginez que vous déménagez dans l'urgence et que vous jetez tout dans des cartons sans étiquettes. Vous avez déménagé à temps, c'était le but, mais désormais, chaque fois que vous cherchez quelque chose, vous perdez dix minutes à fouiller. Ces minutes perdues sont les intérêts. Déballer et étiqueter correctement, c'est rembourser la dette. Le logiciel fonctionne de la même manière : les raccourcis, ou les conceptions qui avaient du sens avant que vous ne compreniez mieux le problème, ralentissent chaque changement ultérieur. Un peu de dette, c'est acceptable, et parfois sage. Le danger est de la laisser s'accumuler jusqu'à ce que l'équipe passe l'essentiel de son temps à fouiller dans les cartons.

CE QU'IL FAUT SAVOIR

- Ward Cunningham a forgé la métaphore de la dette en 1992 pour expliquer le fait de livrer du code reflétant la compréhension du moment, puis de le refactoriser à mesure que la compréhension progresse ; ce n'a jamais été un permis d'écrire du code négligé.
- Le principal est le coût de la correction du problème ; les intérêts sont le surcoût payé à chaque changement tant qu'il subsiste, si bien qu'une dette dans du code qui ne change jamais ne coûte rien.
- Le quadrant de Fowler classe la dette comme délibérée ou involontaire, et prudente ou imprudente ; la dette prudente est un sous-produit normal de la livraison et de l'apprentissage, la dette imprudente est un problème de compétences ou de discipline.
- La dette architecturale, dans la structure plutôt que dans le code, est la plus coûteuse, car la rembourser touche de nombreux modules, de nombreuses équipes et souvent les données.

- Perry et Wolf (1992) ont distingué l'érosion, causée par des violations de l'architecture, de la dérive, causée par une insensibilité à celle-ci.
- Les hotspots (code complexe et fréquemment modifié), le couplage de changement, les cycles de dépendances et les tendances de livraison rendent la dette visible ; un registre, une capacité protégée et la refactorisation menée en parallèle des fonctionnalités la maintiennent sous contrôle.

Ce que Ward Cunningham voulait vraiment dire

Le terme dette technique vient du retour d'expérience de Ward Cunningham sur le système de gestion de portefeuille WyCash, présenté à la conférence OOPSLA en 1992. Il y écrivait que livrer du code écrit pour la première fois revient à s'endetter : un peu de dette accélère le développement, à condition d'être remboursée rapidement par une réécriture, et le danger survient quand elle ne l'est pas : «Chaque minute passée sur du code pas tout à fait juste compte comme des intérêts sur cette dette.» Des organisations d'ingénierie entières, avertissait-il, peuvent être paralysées sous le poids de la dette.

On interprète souvent la métaphore comme une permission d'écrire du code bâclé. Cunningham a expliqué plus tard (dans une courte vidéo de 2009) que ce n'avait jamais été son intention. Son propos portait sur la *compréhension*. Une équipe livre du code qui reflète sa compréhension actuelle, partielle, d'un problème, parce que c'est le moyen le plus rapide d'apprendre de l'usage réel. À mesure que la compréhension s'améliore, le code ne correspond plus à ce que l'équipe sait désormais, et chaque changement doit contourner ce décalage. Rembourser la dette signifie refactoriser le code pour qu'il reflète ce qui a été appris. Cela ne fonctionne que si le code a été écrit assez proprement pour pouvoir être refactorisé au départ. Selon cette lecture, la dette est un sous-produit de l'apprentissage, et non de la négligence.

Depuis, l'usage s'est élargi. Aujourd'hui, « dette technique » couvre toute insuffisance de qualité interne qui rend les changements futurs plus lents ou plus risqués, qu'elle provienne de l'apprentissage, de raccourcis délibérés, de l'ignorance ou d'une simple négligence. Ce sens élargi est utile tant que vous gardez à l'esprit que les différents types de dette appellent des réponses différentes.

Le principal et les intérêts

C'est le vocabulaire financier qui rend la métaphore puissante dans les conversations avec des personnes qui ne sont pas ingénieurs. Le principal est l'effort nécessaire pour corriger le problème : restructurer le module, séparer la base de données, supprimer le cycle. Les intérêts sont l'effort supplémentaire payé sur chaque travail qui touche le problème tant qu'il subsiste.

L’illustration de Martin Fowler : si une fonctionnalité prendrait quatre jours dans du code propre mais en prend six à cause du code encombré (cruft), les deux jours supplémentaires sont des intérêts.

Trois conséquences en découlent. Premièrement, **une dette à laquelle on ne touche jamais ne coûte rien** : un module désordonné que personne ne modifie n’accumule pas d’intérêts, si bien que le rembourser peut être un effort perdu. Deuxièmement, **les intérêts se composent** : les contournements deviennent les fondations de nouveaux contournements, les nouveaux développeurs copient les façons de faire qu’ils voient, et le coût de la réparation finale augmente. Troisièmement, **le remboursement est une décision d’investissement** : il vaut la peine de payer le principal quand les intérêts que vous économiserez sur un horizon raisonnable le dépassent. Cela transforme un argument émotionnel (« le code est un fouillis ») en argument économique (« ce module nous coûte onze jours par trimestre »).

SI VOUS DÉBUTEZ

Pas à pas : quand le désordre coûte de l'argent, et quand il n'en coûte pas

L'idée qui demande le plus d'attention ici est que la dette technique se juge aux intérêts qu'elle prélève, et non à l'aspect plus ou moins désordonné du code. Le principal se paie une fois, pour corriger le problème ; les intérêts se paient à chaque changement qui doit le contourner. Deux modules tout aussi désordonnés l'un que l'autre montrent pourquoi c'est important.

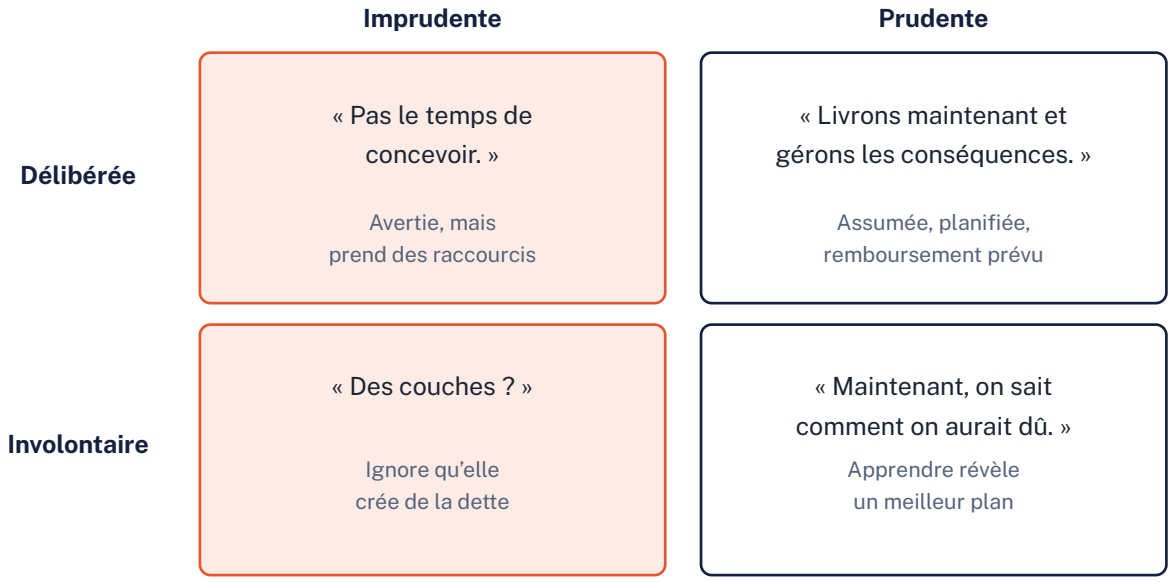
- Prenez les chiffres de Martin Fowler : un changement qui prendrait quatre jours dans du code propre en prend six dans un module désordonné, parce que les développeurs doivent d'abord comprendre le désordre et le contourner. Les deux jours supplémentaires sont des intérêts.
- Supposons que remettre de l'ordre dans le module, c'est-à-dire payer le principal, prenne dix jours-développeur et ramène chaque changement à environ quatre jours. Que cela en vaille la peine dépend de la fréquence à laquelle le module change.
- Si l'équipe le modifie environ une fois par mois, elle paie environ deux jours d'intérêts par mois, donc dix jours de remise en ordre se remboursent en cinq mois environ. Si on ne fait rien, les intérêts ont tendance à augmenter, car de nouveaux contournements se construisent sur les anciens.
- Un second module, tout aussi désordonné, n'est modifié qu'environ deux fois par an. Il prélève environ quatre jours d'intérêts par an, donc les mêmes dix jours de remise en ordre mettraient environ deux ans et demi à être remboursés.
- Un module que personne ne modifie, et qui ne provoque aucune défaillance, ne coûte presque rien en intérêts, aussi désordonné qu'il paraisse. Y remettre de l'ordre peut être un effort perdu.
- **La leçon :** ne demandez pas « à quel point ce code est-il désordonné ? » mais « combien chaque changement coûte-t-il en plus, et à quelle fréquence le modifions-nous ? ». Remboursez d'abord là où les intérêts sont les plus élevés. L'exemple commenté à la fin de cette section fait le même calcul à partir des tickets et de l'historique de la gestion de versions.

Le quadrant de la dette technique de Fowler

Toutes les dettes ne se valent pas. En 2009, Martin Fowler a proposé de classer la dette selon deux axes : a-t-elle été contractée **délibérément** ou **involontairement**, et la décision était-elle **prudente** ou **imprudente** ? Les quatre combinaisons appellent des réponses différentes.

Prudente et délibérée : l'équipe connaît la meilleure conception, juge que livrer maintenant vaut le coût, et prévoit de rembourser. C'est le sens originel de Cunningham, et cela peut être une bonne décision commerciale : une start-up qui prouve la demande avant d'investir dans le passage à l'échelle. **Imprudente et délibérée** : l'équipe sait faire mieux mais saute la conception parce qu'elle croit ne pas avoir le temps, et se ralentit généralement elle-même en quelques semaines. **Imprudente et involontaire** : l'équipe n'en sait pas assez pour voir qu'elle crée de la dette ; le remède passe par les compétences, les revues et le mentorat plutôt que par la seule refactorisation. **Prudente et involontaire** : une équipe compétente fait du bon travail et ne comprend que plus tard comment elle aurait dû structurer le système. Fowler soutient que c'est inévitable pour toute équipe qui apprend, et c'est pourquoi la refactorisation continue fait partie du travail normal au lieu d'être un signe d'échec.

Le quadrant de la dette technique



D'après Martin Fowler (2009) : les deux quadrants imprudents relèvent d'un problème de discipline ou de compétence ; les deux quadrants prudents sont des conséquences normales de la livraison en situation d'incertitude.

Dette architecturale ou dette de code

La dette existe à tous les niveaux, mais son profil de coût change avec l'échelle. La **dette de code** vit à l'intérieur d'un module : logique dupliquée, fonction qui fait cinq choses, tests manquants, noms mal choisis. Elle est réelle, mais locale ; un développeur peut généralement la rembourser en quelques heures ou quelques jours, souvent en travaillant sur une fonctionnalité voisine. La **dette architecturale** vit dans la structure : des services qui partagent une base de données, si bien qu'aucun ne peut modifier son schéma seul ; un domaine découpé selon les mauvaises lignes, si bien que chaque fonctionnalité touche quatre équipes ; une

chaîne d'appels synchrones qui permet à un seul service lent de faire tomber l'étape de paiement ; un framework soudé à chaque couche. La rembourser touche de nombreux modules, souvent de nombreuses équipes, et parfois les données.

La recherche confirme l'intuition selon laquelle la dette structurelle est la plus coûteuse. Une enquête de 2015 menée auprès de 1831 ingénieurs et architectes par Neil Ernst et ses collègues du Software Engineering Institute a montré que les décisions d'architecture étaient la principale source de dette technique signalée, devant les problèmes au niveau du code. Les outils qui signalent les défauts de code (code smells) sont bons pour trouver la dette de code et mauvais pour trouver la dette architecturale, car le problème se situe dans les relations entre les parties plutôt que dans un fichier en particulier.

La dette architecturale compte pour la même raison que l'architecture compte tout court : comme l'a montré Pourquoi l'architecture compte, les décisions structurelles sont celles dont le coût de changement est le plus élevé.

Érosion et dérive

La dette d'architecture s'accumule souvent non pas à cause d'une seule mauvaise décision, mais au fil de milliers de petits changements. Dewayne Perry et Alexander Wolf, dans leur article de 1992 « Foundations for the Study of Software Architecture », ont nommé deux façons dont cela se produit. L'**érosion de l'architecture** est causée par des *violations* de l'architecture : un contrôleur qui interroge directement la base de données en contournant la couche de services ; un module qui va puiser dans les tables internes d'un autre. Chaque violation rend le système plus fragile, comme lorsqu'on retire un mur porteur. La **dérive de l'architecture** est causée par une *insensibilité* à l'architecture : des changements qui n'enfreignent aucune règle explicite mais ignorent les concepts prévus, si bien que le système perd peu à peu sa cohérence et la clarté de sa forme. La dérive est moins immédiatement dangereuse que l'érosion, mais elle obscurcit l'architecture, ce qui à son tour facilite les violations.

Les chercheurs ultérieurs emploient souvent « érosion » pour désigner l'écart plus large entre l'architecture prévue (ce que disent les documents et la tête de l'architecte) et l'architecture implémentée (ce que fait réellement le code). Quels que soient les termes, les causes sont familières :

La pression des délais, qui récompense le chemin le plus court plutôt que le bon chemin. **La perte de connaissances**, quand les personnes qui comprenaient la conception partent et que les nouveaux arrivants copient ce qu'ils trouvent. **Une architecture invisible** : des règles qui ne vivent que dans un wiki que personne ne lit, sans rien dans le build pour les faire respecter. **Des exigences qui ont changé**, et qui laissent une structure autrefois bonne mal adaptée au problème. **Les changements d'organisation** : selon la loi de Conway, une réorganisation pousse

le système à se remodeler selon les nouvelles frontières des équipes. **Le vieillissement des technologies** : les frameworks et les plateformes qui ne sont plus pris en charge se transforment en dette sans que personne touche au code.

Les lois de Lehman : pourquoi cela se reproduit sans cesse

M. M. Lehman a commencé à étudier la façon dont les grands programmes évoluent à la fin des années 1960, à travers ses travaux sur le processus de l'OS/360 d'IBM, et a affiné ses conclusions en huit « lois » dans les années 1990. Elles s'appliquent aux systèmes de type E : des logiciels ancrés dans le monde réel, qui transforment le monde qu'ils servent. Trois comptent surtout ici. **Changement continu** : un tel système doit être adapté en permanence, sinon il devient de moins en moins satisfaisant. **Complexité croissante** : à mesure qu'il évolue, sa complexité augmente, à moins qu'un travail ne soit fait pour la maintenir ou la réduire. **Qualité décroissante** : sa qualité perçue baisse à moins qu'il ne soit rigoureusement maintenu. Ce sont des observations empiriques, pas de la physique ; une étude de 1997 en a confirmé certaines et a laissé d'autres incertaines. La plateforme de livraison de repas se dégradera si elle est livrée à elle-même, même sans aucune mauvaise décision.

La grosse boule de boue (Big Ball of Mud)

En 1997, Brian Foote et Joseph Yoder ont écrit « Big Ball of Mud », nommant ce qu'ils appelaient la plus fréquemment déployée des architectures logicielles : « un système structuré avec désinvolture, voire au petit bonheur », façonné davantage par l'opportunisme que par la conception. Leur article est inhabituel parce qu'il demande pourquoi la boue persiste au lieu de se contenter de la condamner. Les délais récompensent le logiciel qui fonctionne tout de suite, l'architecture est largement invisible, la compréhension initiale du domaine est mince, et le code jetable est rarement jeté. Les systèmes désordonnés fonctionnent aussi, et les personnes capables de s'y retrouver prennent de la valeur. Les auteurs la traitent donc comme un patron animé par des forces : une croissance fragmentaire exige des réparations régulières, un désordre peut être balayé sous le tapis derrière une interface propre, et la section 11.6 traite de la suite quand ce n'est pas possible.

Détecter la dette

Une dette que vous ne voyez pas ne peut pas être gérée, et l'intuition n'est pas fiable : les développeurs ont tendance à se plaindre le plus fort du code qu'ils touchent le plus, ce qui est utile, mais ils savent rarement combien coûte chaque problème. Plusieurs techniques rendent la dette visible.

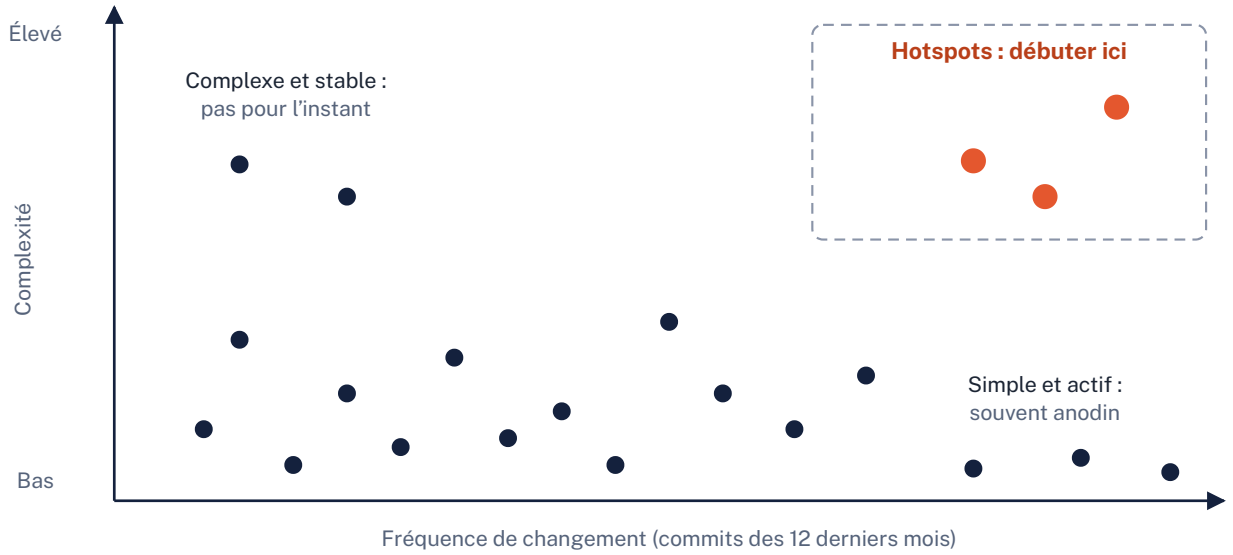
Hotspots (points chauds). *Your Code as a Crime Scene* d'Adam Tornhill (2015 ; deuxième édition 2024) applique des idées de criminalistique à l'historique de la gestion de versions. Un hotspot est du code à la fois complexe et souvent modifié. La complexité seule ne suffit pas : un fichier complexe auquel personne ne touche ne coûte aucun intérêt. La fréquence de changement seule ne suffit pas non plus : un simple fichier de configuration peut changer chaque jour sans dommage. Là où les deux coïncident, la dette se paie à chaque commit, et c'est là que le remboursement est rentable en premier. Dans la plupart des bases de code, une petite fraction des fichiers concentre l'essentiel de l'activité de changement.

Couplage de changement. Des fichiers ou des services qui changent sans cesse dans les mêmes commits sont couplés, que le code montre ou non une dépendance. Si chaque changement du service de tarification s'accompagne d'un changement du service de facturation, il s'agit d'une seule chose qui fait semblant d'en être deux. Un couplage de changement qui traverse les frontières d'équipes ou de services est un signe fort de dette architecturale.

Cycles de dépendances et violations de règles. L'analyse statique peut tracer le graphe de dépendances réel et le comparer au graphe prévu. Les cycles (A utilise B, B utilise C, C utilise A) signifient qu'aucun des trois ne peut être compris, testé ou mis en production seul. Les règles d'architecture (« la couche domaine ne doit pas dépendre de la couche web ») peuvent être vérifiées automatiquement à chaque build ; ce sont des fonctions d'aptitude (fitness functions), traitées au chapitre 11.

Tendances de livraison. L'allongement du délai de mise en œuvre des changements, davantage de défauts qui passent entre les mailles à chaque version, une intégration plus longue des nouveaux arrivants, davantage de réunions de coordination par fonctionnalité et une réticence croissante à toucher certaines zones sont des indicateurs retardés des intérêts accumulés. Les métriques de livraison DORA (comme le délai de mise en œuvre des changements et le taux d'échec des changements) offrent un moyen cohérent de suivre certains d'entre eux dans le temps.

Repérer les hotspots



Placer chaque fichier selon la fréquence à laquelle il change et sa complexité montre où la dette se paie le plus souvent : les quelques fichiers du coin supérieur droit.

Gérer la dette

Aucune base de code saine n'a une dette nulle, et chercher à atteindre zéro est en soi un gaspillage. Le but est de garder la dette visible, délibérée et abordable. Les équipes qui y parviennent combinent plusieurs pratiques.

Un registre de la dette. Consignez les éléments de dette importants là où l'équipe planifie son travail : quel est le problème, où il se trouve, comment il est apparu, le principal estimé, les intérêts estimés (avec des éléments probants comme les données de hotspots ou des tickets récents), et le déclencheur de son remboursement. Un registre transforme des plaintes vagues en une liste qui peut être priorisée au même titre que les fonctionnalités.

Une part fixe de la capacité. Beaucoup d'équipes réservent une proportion stable de chaque itération, couramment entre 10 et 20 %, au remboursement de la dette et à d'autres travaux d'amélioration. Aucune donnée ne justifie un chiffre magique ; ce qui compte, c'est que cette part soit protégée et ne disparaisse pas dès qu'une échéance approche.

Refactoriser en même temps que les fonctionnalités. Le moment le moins coûteux pour rembourser la dette est celui où vous travaillez déjà dans la zone, parce que vous avez le contexte en tête et les tests qui tournent. Fowler appelle cela la refactorisation opportuniste. Sa forme quotidienne est la règle du boy-scout, popularisée par Robert C. Martin : laissez le code un peu plus propre que vous ne l'avez trouvé. Comme les hotspots sont par définition les endroits où arrivent les fonctionnalités, cela concentre naturellement l'effort là où les intérêts sont les plus élevés.

Un remboursement structurel planifié. La dette architecturale cède rarement à de petites étapes seules. Elle exige un chantier explicite et financé, avec un objectif, un plan de migration par incréments et le soutien des parties prenantes. Des techniques comme le patron strangler fig (figuier étrangleur) (chapitre 11, Moderniser les systèmes hérités) permettent à une équipe de remplacer une structure morceau par morceau sans réécriture big bang.

La prévention. Des vérifications automatisées de l'architecture dans le build, une revue de code qui examine la structure autant que le style, et des enregistrements de décision à jour ralentissent l'érosion et la dérive à la source.

Le développement assisté par l'IA : un code plus rapide, une érosion plus rapide ?

Les assistants de code et les agents (section 12.4) peuvent désormais produire du code plus vite qu'une équipe ne peut le lire. Le rapport DORA 2025 de Google, établi auprès de près de 5000 professionnels de la technologie, a constaté que 90 % d'entre eux utilisaient l'IA au travail. L'adoption était associée à un débit de livraison plus élevé mais à une stabilité de livraison plus faible, et la principale conclusion de DORA était que l'IA est un amplificateur : elle grossit les forces et les faiblesses existantes d'une équipe, et les systèmes faiblement couplés dotés de retours rapides y gagnent le plus. Ce sont des corrélations d'enquête. Les preuves contrôlées sont plus minces et évoluent vite : l'essai de METR sur des outils du début de 2025 a montré des développeurs expérimentés plus lents de 19 % alors qu'ils se croyaient plus rapides, et son suivi de 2026 a suggéré que les outils plus récents aident probablement, tout en qualifiant ses propres données de preuve très faible de l'ampleur de cette aide. Mesurez dans votre propre contexte et vérifiez les recherches actuelles.

Le risque architectural est l'érosion à grande vitesse. Un outil travaille sur la tâche qu'il a devant lui, avec une vue limitée de l'architecture voulue, et il tend à imiter ce qui se trouve à proximité. Demandez à un agent d'ajouter un motif de remboursement à la plateforme de livraison de repas : il pourra interroger directement les tables de Paiements, parce que cela compile et que les tests passent. C'est une violation au sens de Perry et Wolf, produite en quelques secondes. GitClear, éditeur d'outils d'analyse de code, a signalé que les lignes clonées étaient passées de 8,3 à 12,3 % des lignes modifiées entre 2021 et 2024, et que le refactoring était tombé de 25 % à moins de 10 %. Cela montre une tendance, pas la preuve d'une cause. Un coût plus discret est la compréhension : du code peut arriver en production sans que personne dans l'équipe sache l'expliquer.

Le travail de l'architecte se déplace donc de la production de la structure vers son énoncé et sa défense. Ce qui vivait dans la tête des gens doit devenir explicite : des frontières de modules claires, des enregistrements de décision et des instructions de dépôt que les outils et les

personnes peuvent lire. Ce qui reposait sur la vigilance doit devenir automatique : des règles de dépendance et des fonctions d'aptitude dans le build (section 11.3). Des frontières bien placées limitent aussi les dégâts que peut causer un seul changement.

La gouvernance du code généré en découle. **Responsabilité** : une personne nommément désignée est propriétaire de chaque changement et le revoit selon les exigences habituelles, et les changements générés de grande taille sont découpés jusqu'à pouvoir être revus. **Contrôles automatiques** : les vérifications d'architecture, l'analyse statique, la détection de secrets et les tests s'appliquent quelle que soit la personne ou la chose qui a écrit le code. **Dépendances** : une étude présentée à USENIX Security 2025, portant sur 576 000 échantillons, a montré que les modèles inventent des noms de paquets, en moyenne dans au moins 5,2 % des cas pour les modèles commerciaux et 21,7 % pour les modèles open source, et un attaquant peut enregistrer un tel nom ; restreignez les installations comme à la section 8.6. **Sécurité** : dans une étude menée par Stanford avec un modèle ancien, les participants disposant d'un assistant ont écrit du code moins sûr et étaient plus enclins à le croire sûr. **Permissions** : les agents reçoivent le moindre privilège et aucun identifiant de production. **Politique** : consignez quels outils sont approuvés et quel code ou quelles données peuvent leur être envoyés, et prenez conseil sur les licences et la confidentialité.

Expliquer la dette aux parties prenantes non techniques

Les commanditaires et les product owners n'ont pas besoin d'entendre parler de dépendances cycliques ; ils ont besoin de comprendre l'effet sur ce qui leur importe : la vitesse, le coût, le risque et l'expérience client. La métaphore financière est ici un cadeau, à condition de l'utiliser avec des chiffres.

Parlez en **temps et en argent** : « Chaque changement de l'étape de paiement nous coûte environ trois jours-développeur de plus, à cause de la façon dont le paiement et les commandes sont enchevêtrés. » Parlez en **risques** : « Deux de nos quatre derniers incidents ont commencé dans ce module. » Parlez en **options** : « Tant que nous n'aurons pas séparé ces deux parties, nous ne pourrons pas proposer les nouveaux moyens de paiement que l'équipe commerciale a demandés. » Et présentez le remboursement comme un **investissement avec un délai de retour**, et non comme une supplique pour obtenir du temps d'ingénierie. Évitez deux pièges : présenter la dette comme une faute morale de ceux qui vous ont précédés (elle est généralement prudente ou involontaire), et demander un « sprint de refactorisation » sans limite ni résultat mesurable.

EXEMPLE COMMENTÉ

Exemple commenté : estimer les intérêts d'une dette

La logique de commande et de paiement d'une boutique en ligne partage un seul gros module et les mêmes tables de base de données. L'équipe soupçonne que cela lui coûte cher. Elle rassemble des éléments tirés des six derniers mois de tickets et de l'historique de la gestion de versions, puis fait une estimation.

Changements touchant le module	48 changements en six mois, soit environ 8 par mois (c'est le premier hotspot).
Effort supplémentaire par changement	En comparant avec des changements similaires ailleurs, les développeurs estiment environ 1,5 jour de plus pour chacun, pour comprendre, contourner les tables partagées et faire des tests de régression supplémentaires.
Intérêts en effort	8 changements × 1,5 jour = 12 jours-développeur par mois.
Intérêts en défauts	3 incidents de production en six mois attribués à ce module, chacun coûtant environ 4 jours-développeur d'investigation et de correction : 12 jours ÷ 6 = 2 jours par mois.
Intérêts totaux	Environ 14 jours-développeur par mois. À un coût complet de £600 par jour-développeur, environ £8 400 par mois, soit environ £100 000 par an.
Principal	Séparer les commandes des paiements derrière une interface et scinder les tables, de façon incrémentale : estimé à 45 jours-développeur, soit environ £27 000.
Intérêts résiduels après remboursement	Les développeurs s'attendent à ce qu'un certain surcoût subsiste : estimé à 3 jours-développeur par mois.
Délai de retour	Économie mensuelle d'environ 11 jours-développeur. 45 ÷ 11 donne environ 4 mois pour récupérer le principal.
Recommandation	Rembourser, par incréments d'environ deux semaines chacun, en commençant par les tables le plus souvent impliquées dans des incidents. Suivre le délai de mise en œuvre des changements de l'étape de paiement pour confirmer l'économie.

Chaque chiffre est une estimation avec de larges marges d'erreur. Même en doublant le principal pour le porter à 90 jours, l'investissement est encore remboursé en environ huit mois : la conclusion est donc robuste ; c'est une vérification qui mérite d'être montrée aux parties prenantes.

La même méthode peut montrer que certaines dettes ne valent pas la peine d'être remboursées : un module désordonné modifié deux fois par an peut avoir un délai de retour de plusieurs années.

L'argumentaire est construit dans les termes du commanditaire (jours, argent, incidents, délai de retour), et non en termes de qualité du code.

ATTENTION

Idées reçues courantes

- « La dette technique, c'est du mauvais code. » Une grande partie de la dette vient de bon code écrit avec une compréhension qui s'est améliorée depuis, ou d'exigences qui ont évolué.
- « Toute dette doit être remboursée. » Une dette dans du code qui ne change jamais ne coûte rien ; remboursez là où sont les intérêts.
- « On corrigera ça lors d'une grande réécriture. » Les réécritures big bang dépassent régulièrement leurs délais et perdent des comportements cachés ; un remboursement incrémental est généralement plus sûr.
- « Seuls les développeurs se soucient de la dette. » Ses effets (fonctionnalités plus lentes, davantage d'incidents, opportunités bloquées) sont des effets métier, et les commanditaires devraient aider à décider quoi rembourser.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi un module très désordonné peut-il avoir une faible priorité de remboursement ?

Parce que les intérêts ne se paient que lorsque le code est modifié ou provoque des défaillances. Si personne n'y touche et qu'il est stable, la dette ne coûte presque rien.

▼ Pourquoi la dette imprudente et involontaire ne se guérit-elle pas par la seule refactorisation ?

Parce que l'équipe ne savait pas qu'elle créait de la dette ; tant que ses compétences et ses pratiques de revue ne s'améliorent pas, elle continuera à créer le même type de dette aussi vite qu'elle la rembourse.

▼ Quelle est la différence entre l'érosion et la dérive selon Perry et Wolf ?

L'érosion résulte de la violation des règles de l'architecture ; la dérive résulte de changements faits sans égard pour l'architecture, qui n'enfreignent aucune règle explicite mais brouillent ses concepts.

▼ Pourquoi les analyses de hotspots combinent-elles la fréquence de changement et la complexité plutôt que d'utiliser l'une ou l'autre seule ?

Un code complexe qui ne change jamais accumule peu d'intérêts, et un code simple qui change souvent est peu coûteux à modifier ; seule la combinaison des deux montre où la dette coûte de l'effort de façon répétée.

▼ À vous : Un module désordonné est modifié 6 fois par mois, chaque modification coûtant 1 jour de plus à £600 par jour ; le corriger prend 60 jours. Un autre, tout aussi désordonné, n'a pas été touché depuis deux ans. Quelle dette rembourseriez-vous ?

Remboursez la première. Ses intérêts sont de 6 x £600 = £3600 par mois et son principal de 60 x £600 = £36 000 : la correction se rentabilise donc en 10 mois environ. La seconde a un principal mais pas d'intérêts : du code qui ne change jamais ne coûte rien de plus, laissez-la donc et notez-la dans le registre de la dette. Si elle se met à changer souvent, réexaminez la question. Montrez les chiffres au métier, pas le désordre.

SOURCES

- The WyCash Portfolio Management System, Ward Cunningham (OOPSLA 1992): https://c2.com/doc/oopsla92.html
- Technical Debt, Martin Fowler: https://martinfowler.com/bliki/TechnicalDebt.html
- Technical Debt Quadrant, Martin Fowler (2009): https://martinfowler.com/bliki/TechnicalDebtQuadrant.html
- Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt (SEI, 2015): https://sei.cmu.edu/library/measure-it-manage-it-ignore-it-software-practitioners-and-technical-debt
- Ward Explains Debt Metaphor (c2 wiki): https://wiki.c2.com/?WardExplainsDebtMetaphor
- Lehman: Laws of Software Evolution Revisited: https://www.cs.kent.edu/~jmaletic/cs63902/Papers/Lehman96.pdf
- Lehman et al.: Metrics and Laws of Software Evolution, The Nineties View: https://www.ifi.uzh.ch/dam/jcr:00000000-5852-3622-0000-00002e60ff8e/lehman-metrics-97.pdf
- Foote and Yoder: Big Ball of Mud: http://laputan.org/mud/
- Google Cloud: Announcing the 2025 DORA report: https://cloud.google.com/blog/products/ai-machine-learning/announcing-the-2025-dora-report
- METR: Measuring the impact of early-2025 AI on experienced open-source developer productivity: https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/
- METR: developer productivity experiment update (February 2026): https://metr.org/blog/2026-02-24-uplift-update/

- GitClear: AI Copilot Code Quality 2025: https://www.gitclear.com/ai_assistant_code_quality_2025_research
- Spracklen et al.: We Have a Package for You!: https://www.usenix.org/conference/usenixsecurity25/presentation/spracklen
- Perry et al.: Do Users Write More Insecure Code with AI Assistants?: https://arxiv.org/abs/2211.03622

SECTION 1.8 SUR 1.8 · INTERMÉDIAIRE · 21 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

L'architecture tout au long du cycle de vie : en amont, émergente et continue

AVANT DE LIRE

Cette section s'appuie sur :

- 0.8 Comment les logiciels sont construits
- 1.4 Le rôle de l'architecte
- 1.6 Décisions et compromis

Termes déjà vus qu'elle utilise : Agile, Livraison continue, Dernier moment responsable, Fonction d'aptitude (fitness function), Attribut de qualité

L'IDÉE EN BREF

Quand vous préparez un long voyage, vous réservez les vols et l'hôtel de la première nuit bien à l'avance, parce qu'ils coûtent cher à modifier et pourraient être complets. Vous ne planifiez pas chaque repas de chaque jour avant de partir ; vous en décidez en route, une fois que vous savez comment vous vous sentez et ce qui se trouve à proximité. Une bonne architecture logicielle fonctionne de la même manière. Les quelques grandes décisions difficiles à changer sont prises tôt et avec soin. Tout le reste se décide en chemin, à mesure que l'équipe apprend, et le plan est vérifié et ajusté tout au long du voyage au lieu d'être écrit une fois pour toutes puis classé.

CE QU'IL FAUT SAVOIR

- La conception complète en amont convient aux exigences stables et aux changements très coûteux ; la conception purement émergente convient aux petits systèmes faits de portes à double sens ; la plupart des équipes ont besoin d'une voie intermédiaire.
- Le principe du Manifeste agile selon lequel les meilleures architectures émergent d'équipes auto-organisées porte sur qui conçoit et comment, et n'affirme pas que l'architecture est inutile.
- Le modèle piloté par les risques de Fairbanks proportionne l'effort d'architecture au risque d'échec : identifier et hiérarchiser les risques, appliquer des techniques, évaluer la réduction obtenue et s'arrêter lorsque le risque restant est acceptable.
- La piste architecturale de SAFe et l'architecture continue d'Erder et Pureur traitent toutes deux l'architecture comme un travail permanent qui doit garder une longueur d'avance sur les fonctionnalités qui en dépendent.

- Un squelette ambulant prouve l'architecture de bout en bout dès le début ; les spikes transforment des décisions incertaines en décisions fondées sur des éléments probants.
- Dans la livraison continue et le DevOps, la déployabilité et l'exploitabilité deviennent des attributs de qualité clés, une architecture faiblement couplée favorise la performance de livraison, et le système en fonctionnement devient la principale source de retours sur l'architecture.

Le vieux débat : en amont ou émergente ?

Pendant la majeure partie de l'histoire du logiciel, l'architecture était quelque chose que l'on faisait au début. Dans un processus séquentiel (« en cascade »), on recueillait les exigences, puis on produisait et faisait approuver une architecture et une conception détaillée, puis on construisait le système selon cette conception. C'est la conception complète en amont (big design up front) (BDUF). Sa logique est solide dans certains contextes : lorsque les exigences sont stables et bien comprises, lorsque les changements après livraison coûtent extrêmement cher (logiciels de vol, dispositifs médicaux, systèmes liés au matériel), ou lorsque de nombreux prestataires doivent construire des parties qui s'emboîtent, une conception approfondie avant la construction réduit le risque.

Ses faiblesses sont tout aussi connues. Les exigences changent, et une conception élaborée tôt encode des hypothèses qui se révèlent fausses par la suite. Les retours de l'usage réel n'arrivent qu'à la fin, quand le changement coûte le plus cher. Les équipes conçoivent pour des besoins imaginaires (« nous pourrions avoir besoin de gérer cinq devises ») et paient pour une flexibilité que personne n'utilise. Et les documents dont l'approbation a pris des mois tendent à être suivis alors qu'ils ont cessé d'être vrais.

Les méthodes agiles ont réagi contre cela. L'Extreme Programming, en particulier, a plaidé pour la conception émergente : construire la chose la plus simple qui fonctionne pour les fonctionnalités en cours, garder le code propre grâce à une refactorisation continue et à des tests automatisés, et laisser la conception grandir avec le système. Les règles de conception simple de Kent Beck et le principe « you aren't gonna need it » (YAGNI) expriment la même idée. La conception émergente fonctionne bien à l'intérieur d'un module, et pour les petits systèmes où chaque décision est une porte à double sens. Sa faiblesse apparaît à grande échelle : certaines décisions (le modèle de données d'un système de référence, les frontières entre équipes, le modèle de sécurité, le choix entre intégration synchrone et asynchrone) sont coûteuses à inverser et façonnent tout ce qui est construit par-dessus. Les laisser émerger signifie souvent qu'elles sont prises par accident par la personne qui écrit le premier code.

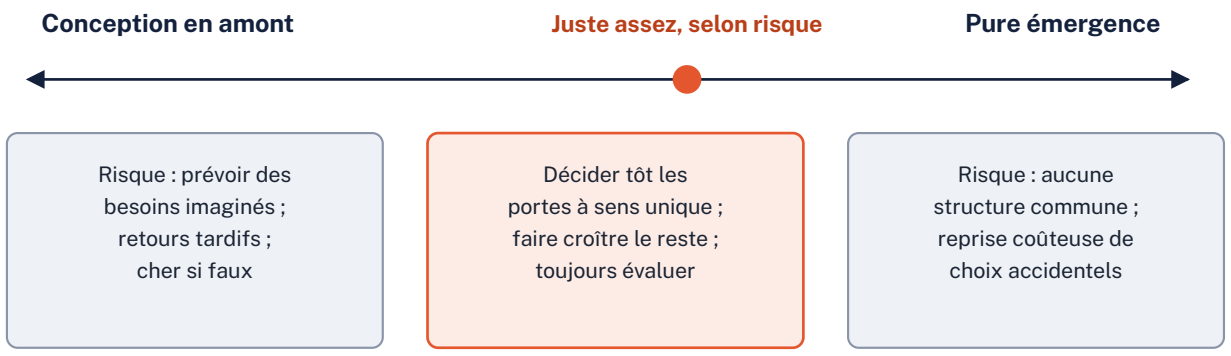
Ce que le Manifeste agile dit et ne dit pas

L'un des douze principes sous-jacents au Manifeste agile (2001) dit : «Les meilleures architectures, spécifications et conceptions émergent d'équipes auto-organisées.» On le cite souvent comme s'il signifiait que les équipes agiles n'ont besoin ni d'architecture ni d'architectes. Ce n'est pas ce qu'il dit.

Lu attentivement, le principe affirme quelque chose sur *qui* produit les meilleures architectures (les personnes qui construisent le système, organisées par elles-mêmes plutôt que dirigées de l'extérieur) et sur *comment* (elles émergent du travail, au lieu d'être transmises toutes faites). Il ne dit nullement que l'architecture serait sans importance, et le même document contient un autre principe : une attention continue à l'excellence technique et à une bonne conception renforce l'agilité. Une équipe qui ignore les décisions structurelles ne suit pas le manifeste ; elle en ignore la moitié.

Ce que le principe exclut bel et bien, c'est une architecture imposée depuis une tour d'ivoire par des personnes qui n'en vivront pas les conséquences, ainsi qu'une phase durant laquelle l'architecture est terminée et figée avant que quiconque écrive du code. En pratique, la conciliation la plus courante est que l'architecture devient une activité continue, partagée au sein de l'équipe, avec un architecte (ou des ingénieurs sensibles à l'architecture) dans le rôle de guide, de facilitateur et de gardien de la vue d'ensemble plutôt que dans celui de gardien du temple.

Un spectre, pas un choix



La plupart des équipes qui réussissent se situent entre les extrêmes : elles tranchent tôt les questions coûteuses à inverser et laissent le reste émerger.

Combien d'architecture ? Le modèle piloté par les risques

Si aucun des deux extrêmes n'est le bon, quelle quantité de travail d'architecture suffit ? George Fairbanks a donné l'une des réponses les plus claires dans *Just Enough Software Architecture: A Risk-Driven Approach* (2010). Son modèle piloté par les risques dit que l'effort

consacré à l'architecture doit être proportionné au risque d'échec. Il comporte trois étapes, répétées autant que nécessaire : **identifier et hiérarchiser les risques** (idéalement sous forme de scénarios de défaillance concrets, comme « la page de réservation dépasse son délai d'expiration quand 5000 patients se connectent à 8 h ») ; **choisir et appliquer des techniques** qui réduisent ces risques (un modèle, un prototype, un test de charge, une analyse, un patron de conception) ; et **évaluer la réduction du risque**, en s'arrêtant lorsque le risque restant est acceptable.

Le modèle est puissant parce qu'il fournit une règle d'arrêt. Un simple outil interne construit sur une technologie familière avec peu d'utilisateurs fait face à peu de risques architecturaux, et un croquis sur un tableau blanc peut suffire. Une plateforme de paiement qui traite des millions de transactions, soumise à une réglementation, en affronte beaucoup, et mérite une modélisation, un prototypage et une revue soigneux. Une même équipe peut appliquer des quantités d'architecture très différentes à différentes parties d'un même système. Fairbanks souligne aussi que le temps passé à concevoir est du temps qui n'est pas consacré à construire ou à tester, si bien que la surconception est un coût réel, et non un choix par défaut sans danger.

SI VOUS DÉBUTEZ

Pas à pas : juste assez d'architecture pour le système de réservation

Le point qui fait le plus souvent trébucher ici est le modèle piloté par les risques de Fairbanks : il n'existe pas de bonne quantité fixe d'architecture, seulement une quantité adaptée aux risques. Vous continuez à concevoir tant que le risque d'échec est trop élevé, et vous vous arrêtez lorsqu'il est acceptable. Suivez-le à travers le système de réservation hospitalier.

- Listez les risques sous forme de défaillances concrètes, comme « la page de réservation dépasse son délai d'expiration quand 5000 patients se connectent à 8 h » et « le système d'administration des patients, qui fournit les créneaux libres des cliniques, ne peut pas répondre à 50 requêtes par seconde ». Les parties construites sur une technologie familière pour peu d'utilisateurs présentent peu de risques sérieux.
- Hiérarchisez. La ruée de 8 h toucherait tous les patients qui réservent à ce moment-là : elle passe donc en tête. Les parties à faible risque reçoivent un croquis au tableau blanc, et rien de plus.
- Appliquez une technique adaptée à chaque risque. Pour la ruée du matin, l'équipe construit un petit prototype et le teste avec 5000 patients simulés qui se connectent en même temps. Pour le système d'administration des patients, une courte expérience, convenue avec son fournisseur, mesure combien de requêtes par seconde il peut traiter.
- Évaluez de combien chaque risque a diminué. Supposons que le prototype tienne la ruée, mais que le système d'administration des patients ralentisse fortement avant 50 requêtes par seconde. Le premier risque est désormais acceptable ; le second ne l'est pas, donc l'équipe refait un tour, en testant peut-être le cache de créneaux à courte durée de vie de la section 1.5.
- Arrêtez-vous lorsque chacun des principaux risques est suffisamment faible. Continuer à concevoir n'est pas un choix par défaut sans danger, prévient Fairbanks, car cela prend du temps à la construction et aux tests.
- **La leçon :** laissez les risques fixer l'effort. Une partie d'un système peut recevoir un croquis au tableau blanc et une autre des semaines de prototypage et de tests : c'est le modèle qui fonctionne.

L'architecture intentionnelle et la piste architecturale

Les grandes organisations font face à un problème de coordination qu'une équipe unique ne connaît pas : des dizaines d'équipes doivent construire des parties qui s'emboîtent. Le Scaled Agile Framework (SAFe), un cadre largement adopté, et aussi largement débattu, pour mettre à l'échelle les pratiques agiles, décrit l'architecture comme un équilibre entre la *conception émergente* au sein des équipes et l'*architecture intentionnelle* : la planification délibérée, entre

équipes, des structures et des lignes directrices qu'elles partagent. Il appelle le résultat la piste architecturale (architectural runway) : le code, les composants et l'infrastructure technique existants nécessaires pour implémenter les fonctionnalités à court terme sans refonte ni retard excessifs. Le travail d'extension de la piste (par exemple, introduire une plateforme de messagerie avant les fonctionnalités qui en ont besoin) est planifié explicitement à côté du travail sur les fonctionnalités, et les fonctionnalités consomment la piste à mesure qu'elles l'utilisent.

Décrite de façon neutre, la métaphore de la piste saisit une vérité valable pour toute organisation : si l'infrastructure dont une fonctionnalité a besoin n'existe pas encore, cette fonctionnalité sera lente, et construire l'infrastructure prend du temps. Les détracteurs de SAFe soutiennent qu'il peut réintroduire une planification lourde et descendante sous un vocabulaire agile ; ses partisans répondent qu'une certaine coordination centrale est inévitable à grande échelle. Vous n'avez pas besoin d'adopter SAFe pour utiliser l'idée : « de quelles fondations les fonctionnalités du prochain trimestre auront-elles besoin, et les construisons-nous à temps ? » est une bonne question pour tout architecte.

L'architecture continue

Murat Erder et Pierre Pureur, dans *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World* (2015) et *Continuous Architecture in Practice* (2021, avec Eoin Woods), décrivent l'architecture comme un flux continu de décisions plutôt que comme une phase ou un document. Leurs six principes sont : concevoir l'architecture de produits, et pas seulement de solutions pour des projets ; se concentrer sur les attributs de qualité, et non sur les exigences fonctionnelles ; retarder les décisions de conception jusqu'à ce qu'elles soient absolument nécessaires ; concevoir pour le changement en tirant parti de la puissance de petites parties faiblement couplées ; concevoir pour construire, tester, déployer et exploiter ; et modeler l'organisation de vos équipes sur la conception du système sur lequel vous travaillez.

Deux de ces principes font écho aux sections précédentes de ce chapitre : retarder les décisions, c'est le dernier moment responsable, et modeler les équipes sur le système est un usage délibéré de la loi de Conway. L'accent pratique est mis sur la *décision* comme unité du travail d'architecture, consignée au moment où elle est prise, et sur les boucles de rétroaction (tests automatisés, fonctions d'aptitude, supervision en production) qui indiquent à l'équipe si l'architecture atteint toujours ses objectifs de qualité.

Construire des architectures évolutives

Un courant voisin vient de *Building Evolutionary Architectures* de Neal Ford, Rebecca Parsons et Patrick Kua (2017 ; deuxième édition en 2022 avec Pramod Sadalage). Les systèmes de longue durée doivent suivre l'évolution de leur environnement ; au lieu de prédire l'avenir, les auteurs

conçoivent donc pour le changement : de nombreux petits pas, sur les dimensions techniques comme métier, en rendant explicite ce que signifie « aptitude » et en la vérifiant automatiquement partout où c'est possible. Si la plateforme de livraison de repas promet que l'étape de paiement reste rapide, un test automatisé peut protéger cette promesse à chaque mise en production pendant que le code sous-jacent change. La section 11.3 traite de ces vérifications, appelées fonctions d'aptitude, et des limites réelles de l'approche.

Squelettes ambulants et spikes

Deux techniques simples rapprochent l'architecture et la livraison dès la première semaine. Un squelette ambulant (walking skeleton), terme associé à Alistair Cockburn, est une implémentation minuscule du système qui réalise une petite fonction de bout en bout à travers tous les principaux composants architecturaux. Pour un système de réservation hospitalier, il pourrait permettre à un utilisateur de test de se connecter, de voir les créneaux libres d'une clinique codée en dur, récupérés depuis le système d'administration des patients, d'en réserver un et de recevoir un SMS, le tout déployé dans un environnement proche de la production via le vrai pipeline de livraison. Il ne fait presque rien d'utile, mais il prouve que les pièces se connectent, que le pipeline fonctionne et que l'architecture peut être construite. Chaque fonctionnalité ultérieure ajoute de la chair à un squelette qui marche déjà.

Un spike (autre terme issu de l'Extreme Programming) est une expérience courte, limitée dans le temps, destinée à répondre à une question ou à réduire un risque : l'interface du PAS (système d'administration des patients) peut-elle traiter 50 requêtes par seconde ? Combien de temps le fournisseur d'identité met-il à émettre un jeton ? La bibliothèque que nous avons choisie peut-elle générer des PDF accessibles ? Les spikes sont généralement jetés ; leur résultat est la connaissance, consignée dans une décision. C'est le moyen le moins coûteux de faire passer une décision de la conjecture aux éléments probants avant son dernier moment responsable.

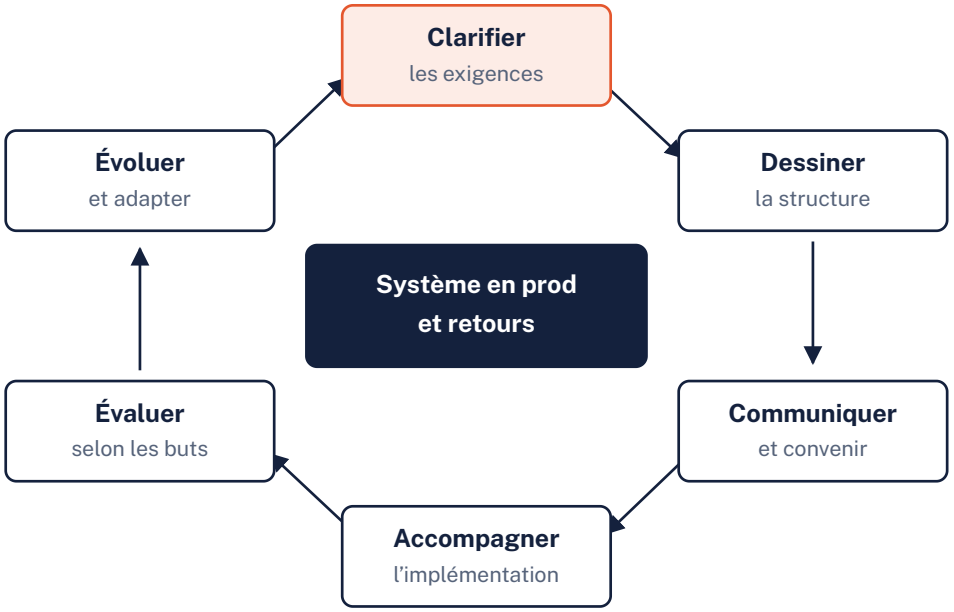
Les activités d'architecture dans un cycle itératif

Dans un processus itératif, le travail d'architecture n'a pas lieu une seule fois ; il revient à chaque cycle, avec une profondeur adaptée aux risques auxquels l'équipe fait face. Le programme de l'iSAQB décrit les tâches essentielles de l'architecte d'une manière qui se transpose naturellement à un tel cycle, et Le rôle de l'architecte les a présentées. Les voici sous forme de boucle :

Clarifier les exigences : identifier les parties prenantes et les préoccupations, préciser les objectifs de qualité et les contraintes, et identifier les risques (voir Parties prenantes et préoccupations). **Concevoir les structures** : prendre ou réviser les décisions significatives, en choisissant des styles et des patrons et en définissant les frontières et les interfaces.

Communiquer et convenir : expliquer les décisions aux personnes qui construisent, exploitent et financent le système, dans des vues adaptées à chaque public, et les consigner. **Accompagner l'implémentation** : travailler avec les équipes pendant qu'elles construisent, répondre aux questions, relire le code et ajuster la conception quand la réalité résiste. **Évaluer** : vérifier que le système atteint ses objectifs de qualité, par des revues, des tests, des mesures en production et des fonctions d'aptitude automatisées. **Faire évoluer** : agir sur ce que l'évaluation a révélé, rembourser la dette et s'adapter aux exigences qui ont changé, ce qui vous ramène à la clarification des exigences.

Le cycle de l'architecture



Chaque itération revisite les six activités, avec une profondeur proportionnée aux risques auxquels l'équipe fait face, autour d'un système en fonctionnement qui fournit des retours.

L'architecture dans le DevOps et la livraison continue

La livraison continue, dans laquelle chaque changement est construit, testé et maintenu prêt à être mis en production, et le DevOps, qui réunit les responsabilités du développement et de l'exploitation, modifient à la fois les objectifs de l'architecture et ses boucles de rétroaction.

Ils modifient les objectifs parce que **la déployabilité, la testabilité et l'exploitabilité deviennent des attributs de qualité de premier rang**. Un système ne peut pas être mis en production plusieurs fois par jour si les composants doivent être déployés ensemble dans un ordre fixe, si les tests nécessitent une copie complète de la production, ou si personne ne peut dire, d'après la supervision, si une mise en production a aggravé les choses. Les recherches résumées dans *Accelerate* (Nicole Forsgren, Jez Humble et Gene Kim, 2018) ont montré qu'une

architecture faiblement couplée, qui permet aux équipes de tester et de déployer leurs parties indépendamment, figurait parmi les plus puissants prédicteurs de la performance en livraison continue. L'architecture n'est donc pas seulement aidée par de bonnes pratiques de livraison ; elle en est l'une des conditions préalables.

Ils modifient les boucles de rétroaction parce que le système en fonctionnement devient la revue d'architecture la plus honnête. Les métriques de production, les traces et les revues d'incidents montrent où l'architecture est faible de façon bien plus fiable que les diagrammes. Les règles d'architecture peuvent être appliquées sous forme de fonctions d'aptitude automatisées dans le pipeline (chapitre 11), l'infrastructure est définie en tant que code et relue comme du code (chapitre 10), et des techniques comme les feature flags et les canary releases (déploiements canari) rendent de nombreuses décisions réversibles à moindre coût. Le rôle de l'architecte évolue en conséquence : moins d'approbation de documents, davantage de façonnage de la plateforme, du pipeline et des garde-fous qui permettent aux équipes de prendre rapidement de bonnes décisions.

Liste de contrôle : votre travail d'architecture est-il proportionné ?

Parcourez cette liste au début d'un projet et à chaque jalon important. Chaque élément est une question sur l'adéquation de l'effort au risque.

- [ ] **Vous pouvez nommer les cinq principaux risques architecturaux**
  Rédigés sous forme de scénarios de défaillance concrets, hiérarchisés, et chacun avec un responsable et un plan pour le réduire.
- [ ] **Votre effort de conception se concentre sur ces risques**
  Les zones à faible risque et à technologie familière reçoivent un croquis ; les zones à risque élevé reçoivent des prototypes, des mesures et une revue.
- [ ] **Les portes à sens unique sont identifiées et tranchées délibérément**
  Les décisions coûteuses à inverser sont connues, prises au dernier moment responsable et consignées ; les portes à double sens sont laissées aux équipes.
- [ ] **Un squelette ambulant existe tôt**
  Un chemin de bout en bout à travers les principaux composants fonctionne dans un environnement proche de la production dès les premières itérations.
- [ ] **Les décisions sont consignées brièvement au moment où elles sont prises**
  Un court enregistrement de décision existe pour chaque décision significative ; personne n'a à reconstituer le raisonnement de mémoire.
- [ ] **Les objectifs de qualité sont vérifiés en continu**
  Les attributs de qualité clés disposent de tests automatisés, de fonctions d'aptitude ou de mesures en production, et pas seulement d'une revue ponctuelle.
- [ ] **L'architecture ne bloque pas la livraison**
  Les équipes attendent rarement une décision ou une approbation d'architecture ; si c'est le cas, le processus est trop lourd pour le risque.
- [ ] **La livraison ne mine pas l'architecture**
  Les hotspots, le couplage de changement et les violations de règles sont suivis ; l'érosion est remarquée en quelques semaines, pas en quelques années.
- [ ] **La quantité de documentation correspond à ses lecteurs**
  Chaque document ou vue répond à la préoccupation réelle d'une partie prenante et est tenu à jour ; les documents non lus sont retirés.

ATTENTION

Deux modes d'échec opposés

L'astronautique architecturale (architecture astronautics) : des mois de diagrammes, des frameworks construits pour des besoins qui n'arrivent jamais, et une conception obsolète avant la première mise en production. **L'architecture accidentelle** : personne ne réfléchit à la structure jusqu'à ce que le système soit gros, lent à changer et impossible à comprendre, et un sauvetage coûteux devient inévitable. Le remède au premier est le retour d'un code qui tourne ; le remède au second est une attention délibérée aux quelques décisions qui comptent le plus. Le modèle piloté par les risques est un moyen de naviguer entre les deux.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Quand la conception complète en amont est-elle un choix judicieux ?**

Lorsque les exigences sont stables et bien comprises, lorsque le changement après livraison coûte extrêmement cher, ou lorsque de nombreuses parties distinctes doivent construire des éléments qui s'emboîtent exactement.

▼ **Quelle règle d'arrêt le modèle piloté par les risques donne-t-il pour le travail d'architecture ?**

S'arrêter lorsque les risques restants sont suffisamment faibles ; l'effort doit être proportionné au risque d'échec.

▼ **Quelle est la différence entre un squelette ambulant et un spike ?**

Un squelette ambulant est une implémentation minimale de bout en bout à travers l'architecture réelle, que l'on conserve et sur laquelle on construit. Un spike est une courte expérience destinée à répondre à une question, généralement jetée une fois qu'elle a produit de la connaissance.

▼ **Pourquoi la livraison continue rend-elle l'architecture plus importante, et non moins ?**

Des mises en production fréquentes et indépendantes exigent une structure faiblement couplée, testable et déployable ; sans elle, le pipeline de livraison ne peut pas fonctionner à vitesse élevée.

▼ **À vous : Un nouveau système de billetterie a un prestataire de paiement connu, mais personne ne sait si les blocages de places résisteront à 2500 visiteurs par seconde. Quelle quantité de conception initiale feriez-vous, et que construiriez-vous en premier ?**

Suivez le risque. Utilisez une conception légère là où les choses sont bien comprises, comme l'intégration du paiement, et consacrez l'effort au plus grand risque, le blocage des places sous charge. Construisez tôt un squelette ambulant sur tout le parcours d'achat, et menez un spike qui teste en charge l'approche de blocage des places, afin que la décision repose sur des preuves. Arrêtez l'analyse lorsque le risque restant est acceptable. Une conception complète de tout, ou aucune conception, seraient toutes deux inadaptées.

SOURCES

- Principles behind the Agile Manifesto: https://agilemanifesto.org/principles.html
- A Risk-Driven Model for Agile Software Architecture, George Fairbanks (Methods and Tools): https://www.methodsandtools.com/archive/agilearchi.html
- Is Design Dead?, Martin Fowler: https://martinfowler.com/articles/designDead.html
- Architectural Runway (SAFe): https://framework.scaledagile.com/architectural-runway
- Continuous Architecture (Erder, Pureur and Woods): https://continuousarchitecture.com/
- Evolutionary Architecture (authors' site): https://evolutionaryarchitecture.com/
- O'Reilly: Building Evolutionary Architectures, 2nd edition: https://www.oreilly.com/library/view/building-evolutionary-architectures/9781492097532/

Lectures complémentaires

- *Software Architecture in Practice, 4th edition*, Len Bass, Paul Clements and Rick Kazman (2021), Addison-Wesley. Le manuel de référence du SEI : structures, attributs de qualité et tactiques, et la définition utilisée dans tout ce cours.
  https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/
- *Fundamentals of Software Architecture, 2nd edition*, Mark Richards and Neal Ford (2025), O'Reilly Media. Une introduction pratique et agréable à lire au rôle de l'architecte, à la réflexion en termes de compromis et aux styles architecturaux.
- *ISO/IEC/IEEE 42010:2022 Software, systems and enterprise: Architecture description*, ISO/IEC JTC 1/SC 7 and IEEE (2022), ISO. La norme internationale sur les descriptions d'architecture, les parties prenantes, les préoccupations et les vues.
  https://www.iso.org/standard/74393.html
- *Documenting Software Architectures: Views and Beyond, 2nd edition*, Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord and Judith Stafford (2010), Addison-Wesley. La source des catégories de structures module, composants et connecteurs, et allocation.
  https://www.sei.cmu.edu/library/documenting-software-architectures-views-and-beyond-second-edition/
- *Who Needs an Architect?*, Martin Fowler (2003), IEEE Software. Un essai court et influent sur ce qu'est l'architecture et sur le type d'architecte dont une équipe a besoin.
  https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf
- *Just Enough Software Architecture: A Risk-Driven Approach*, George Fairbanks (2010), Marshall & Brainerd. Explique quelle quantité de travail d'architecture fournir, en fonction des risques auxquels un projet est réellement exposé.
  https://www.georgefairbanks.com/book/
- *The Software Architect Elevator*, Gregor Hohpe (2020), O'Reilly Media. Sur le rôle de l'architecte dans les grandes organisations, qui relie la stratégie métier et la réalisation technique.
- *The WyCash Portfolio Management System (OOPSLA '92 experience report)*, Ward Cunningham (1992), OOPSLA 1992. Le court rapport dans lequel la métaphore de la dette technique est apparue pour la première fois.
  https://c2.com/doc/oopsla92.html

Glossaire

Architecturalement significatif
Qualifie une décision ou une exigence coûteuse à modifier par la suite, qui affecte de nombreuses parties d'un système ou qui détermine un attribut de qualité clé.

Architecture d'entreprise
Architecture à l'échelle de toute une organisation : comment ses capacités métier, ses informations, ses applications et sa technologie s'articulent et évoluent dans le temps.

Architecture de solution
Architecture d'une solution à un problème métier donné, qui couvre souvent plusieurs applications, fournisseurs et équipes, par exemple un nouveau processus de déclaration de sinistres en ligne pour un assureur.

Architecture logicielle
Les structures fondamentales d'un système logiciel et les décisions importantes qui les ont façonnées : ses parties principales, leurs relations, leurs propriétés importantes et les principes qui guident sa construction et son évolution.

Attribut de qualité
Propriété mesurable d'un système qui indique dans quelle mesure il remplit bien sa fonction, comme les performances, la disponibilité, la sécurité ou la modifiabilité. Le chapitre 2 les traite en détail.

Biais cognitif
Erreur systématique de raisonnement, comme l'ancrage ou le biais des coûts irrécupérables, qui conduit à juger les options de manière injuste.

Big design up front (BDUF, conception complète en amont)
Approche dans laquelle l'architecture et la conception sont achevées en détail avant le début de l'implémentation.

Code mort
Code qui n'est plus censé s'exécuter mais qui est toujours présent dans le système déployé, où il peut être déclenché par erreur.

Composant
Unité qui existe lorsque le système s'exécute, comme un service, un processus ou une base de données, et qui interagit avec d'autres composants par l'intermédiaire de connecteurs.

Compromis
Choix dans lequel obtenir un avantage implique d'accepter un coût ou de renoncer à un autre avantage, par exemple une livraison plus rapide au prix d'un coût d'exploitation plus élevé.

Conception détaillée
Décisions portant sur l'intérieur d'une partie d'un système (ses classes, fonctions, algorithmes et structures de données locales), qui peuvent être modifiées sans affecter le reste.

Conception émergente
Approche dans laquelle la conception se développe de manière incrémentale, façonnée par les besoins de chaque nouvelle fonctionnalité et par une refactorisation continue.

Connecteur
Mécanisme d'exécution par lequel les composants interagissent : un appel HTTP, une file de messages, une base de données partagée, un flux d'événements ou un appel de fonction.

Couplage de changement
Deux fichiers ou composants ou plus qui ont tendance à changer dans les mêmes commits, révélant une dépendance qui peut ne pas être visible dans le code.

Coût du changement
Effort, argent, temps et risque nécessaires pour modifier une décision une fois que du travail a été construit dessus.

Cycle de dépendances
Boucle dans laquelle des modules dépendent les uns des autres directement ou indirectement, de sorte qu'aucun ne peut être compris, testé ou publié seul.

Dérive de l'architecture
Modifications faites sans connaissance de l'architecture prévue qui, sans enfreindre de règles explicites, éloignent le système de celle-ci et brouillent ses concepts.

Dernier moment responsable
Le point à partir duquel retarder davantage une décision supprimerait une option importante ; décider à ce moment-là permet de disposer du maximum d'informations sans causer de tort.

Description d'architecture
Produit de travail (documents, diagrammes, modèles, enregistrements de décisions) qui exprime une architecture. L'architecture existe qu'elle soit décrite ou non ; la description est la manière dont les personnes la partagent.

Dette architecturale
Dette technique dans la structure d'un système, comme de mauvaises frontières, une base de données partagée ou des dépendances cycliques, coûteuse à rembourser parce qu'elle touche de nombreuses parties.

Dette technique
Le surcoût futur des changements causé par des raccourcis ou par une conception qui n'est plus adaptée, payé sous forme d'un travail plus lent et plus risqué jusqu'à ce qu'il soit remboursé en améliorant la conception.

Élément
Partie d'un système sur laquelle un architecte raisonne comme sur une unité, comme un module, un service, une base de données, une file ou un serveur.

Enregistrement de décision d'architecture (ADR)
Court document qui consigne une décision importante : son contexte, les options envisagées, la décision, son statut et ses conséquences.

Érosion de l’architecture
Dégradation progressive d’un système à mesure que des modifications enfreignent son architecture prévue, par exemple en contournant des couches ou en ajoutant des dépendances interdites.

Étendue technique (technical breadth)
Savoir que de nombreuses technologies et approches existent, ce pour quoi elles sont bonnes ou mauvaises et quand les utiliser, même sans expertise pratique approfondie de chacune.

Feature flag (indicateur de fonctionnalité)
Interrupteur de configuration qui active ou désactive un comportement à l’exécution sans déployer de nouveau code.

Fonction d’aptitude (fitness function)
Vérification automatisée qui vous indique si un système possède toujours une propriété architecturale, comme un test qui échoue lorsqu’un module dépend d’un autre qu’il ne doit pas utiliser.

Hotspot (point chaud)
Partie du code à la fois complexe et fréquemment modifiée, repérée en combinant l’historique de la gestion de versions avec une mesure de complexité ; un endroit tout désigné pour rembourser de la dette.

Intérêts (de la dette technique)
Effort, retard ou risque supplémentaire payé à chaque modification parce que la dette n’a pas été remboursée.

Loi de Conway
Observation de Melvin Conway (1968) selon laquelle les organisations conçoivent des systèmes dont la structure reflète leur propre structure de communication.

Matrice de décision pondérée
Tableau qui note chaque option selon des critères pondérés afin que le raisonnement derrière un choix soit explicite et puisse être contesté.

Matrice pouvoir/intérêt
Matrice à quatre cases qui place les parties prenantes selon leur pouvoir sur un projet et leur intérêt pour celui-ci, utilisée pour planifier le degré d’implication de chaque groupe.

Modèle piloté par les risques
L’approche de George Fairbanks dans laquelle la quantité de travail d’architecture est fixée par les risques auxquels un projet fait face : identifier les risques, appliquer des techniques, évaluer la réduction obtenue.

Monolithe
Système construit et déployé comme une seule unité, même si son code est divisé à l’intérieur en modules bien séparés.

Partie prenante
Personne, équipe ou organisation qui a un intérêt dans un système, comme les utilisateurs, les commanditaires, les développeurs, les exploitants, les responsables de la sécurité ou les autorités de régulation.

Piste architecturale (architectural runway)
Dans le Scaled Agile Framework, le code, les composants et l'infrastructure existants nécessaires pour implémenter les fonctionnalités à court terme sans refonte ni retard excessifs.

Point de vue
Spécification de la manière de construire et de lire un type de vue d'architecture : quelles parties prenantes et quelles préoccupations elle sert, et quelles conventions de modélisation elle utilise.

Préoccupation
Intérêt qu'une partie prenante porte à un système, comme son coût, ses performances, sa sécurité, sa facilité de modification ou sa conformité juridique.

Préoccupation transversale (cross-cutting concern)
Préoccupation qui touche simultanément de nombreuses parties d'un système, comme la journalisation, la sécurité, la gestion des erreurs ou la persistance, et qui fait donc généralement l'objet d'une décision unique pour tout le système.

Preuve de concept (proof of concept)
Petit morceau de code fonctionnel, réalisé en un temps limité, pour vérifier si une idée, une technologie ou un choix de conception fonctionne avant de s'y engager.

Principal (de la dette technique)
L'effort nécessaire pour corriger un élément de dette technique, c'est-à-dire pour amener le code ou la structure à l'état dans lequel il devrait être.

Pyramide de la connaissance
Modèle à trois couches : ce que vous savez, ce que vous savez ne pas savoir, et ce que vous ne savez pas que vous ne savez pas. Les architectes élargissent délibérément la couche du milieu.

Refactorisation (refactoring)
Modifier la structure interne du code sans changer son comportement externe, pour le rendre plus facile à comprendre et à modifier.

Relation
Lien entre des éléments, par exemple un module qui en utilise un autre, un service qui en appelle un autre ou un service déployé sur un serveur.

Réversibilité
La facilité et la sûreté avec lesquelles on peut revenir sur une décision à moindre coût. Les décisions très réversibles peuvent être prises rapidement ; les décisions irréversibles méritent une analyse plus poussée.

Serverless
Modèle de cloud dans lequel le fournisseur exécute vos fonctions ou vos flux de travail à la demande et facture à l'usage, de sorte que vous ne gérez pas vous-même de serveurs.

Spike (investigation limitée dans le temps)
Investigation ou prototype court, limité dans le temps, réalisé pour répondre à une question technique ou réduire un risque, et normalement jeté ensuite.

Squelette ambulant (walking skeleton)
Implémentation minuscule du système qui réalise une petite fonction de bout en bout en traversant tous les principaux composants architecturaux, généralement construite en premier.

Staff engineer
Rôle de contributeur individuel senior, au-dessus de l'ingénieur senior, qui exerce un leadership technique à travers plusieurs équipes sans encadrer de personnes ; le principal engineer est généralement le niveau suivant.

Structure
Ensemble d'éléments et des relations entre eux, vu sous un angle donné : par exemple les modules de code et leurs dépendances, ou les services en fonctionnement et la manière dont ils communiquent.

TOGAF
The Open Group Architecture Framework, cadre et méthode très utilisés pour l'architecture d'entreprise, publiés par The Open Group ; sa version actuelle est le TOGAF Standard, 10th Edition (2022).

Vue
Représentation d'une ou de plusieurs structures d'un système, destinée à des préoccupations précises de parties prenantes précises, comme un diagramme de déploiement pour les équipes d'exploitation.