BE SOFTWARE ARCHITECTURE WISE · CHAPITRE 2

Attributs de qualité et exigences architecturalement significatives

L'architecture existe pour apporter des qualités : la rapidité sous la charge, le maintien en service quand des parties tombent en panne, la possibilité de modifier sans risque, la confidentialité des données. Ce chapitre montre comment transformer des souhaits vagues comme « rapide », « fiable » et « sécurisé » en exigences précises et testables, capables d'orienter les décisions de conception. Il couvre la différence entre exigences fonctionnelles, attributs de qualité et contraintes, le modèle de qualité ISO/IEC 25010, les scénarios d'attributs de qualité, l'arithmétique des performances et de la disponibilité, la conception pour le changement, la sécurité, l'utilisabilité et le coût en tant que qualités, et la manière de les éliciter, de les hiérarchiser et d'arbitrer entre eux. Il s'adresse aux développeurs qui s'orientent vers l'architecture et à toute personne qui rédige ou relit des exigences.

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

EN UN COUP D'ŒIL

Vous apprendrez à

- Distinguer les exigences fonctionnelles, les attributs de qualité et les contraintes, et reconnaître une exigence architecturalement significative.
- Utiliser le modèle de qualité des produits de l'ISO/IEC 25010:2023 comme liste de contrôle sans le laisser se transformer en paperasse.
- Rédiger des scénarios d'attributs de qualité en six parties, avec des mesures de la réponse chiffrées, et transformer des énoncés vagues en énoncés testables.
- Calculer avec les percentiles, la loi de Little, des formules simples de files d'attente et la loi d'Amdahl pour raisonner sur les performances et la scalabilité.
- Calculer la disponibilité à partir des « neuf », du MTBF et du MTTR, et pour des composants en série et en parallèle.
- Expliquer comment la modifiabilité, la testabilité, la déployabilité, la sécurité, l'utilisabilité, le coût et la durabilité sont spécifiés et obtenus.
- Éliciter et hiérarchiser les exigences de qualité à l'aide d'un arbre d'utilité, et expliquer les compromis entre qualités concurrentes.

Avant de commencer

Le chapitre 1, en particulier les sections sur les parties prenantes et leurs préoccupations, et sur les décisions et les compromis. Une certaine aisance avec l'arithmétique simple (pourcentages, multiplications et divisions).

Durée

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

Termes clés

attribut de qualité · contrainte · exigence architecturalement significative · ISO/IEC 25010 · scénario d'attribut de qualité · mesure de la réponse · latence et débit · percentile · Loi de Little · disponibilité · MTTR · tactique · arbre d'utilité · fonction d'aptitude (fitness function)

Pourquoi ce sont les qualités, et non les fonctionnalités, qui décident de l'architecture

Demandez à n'importe quel groupe de parties prenantes ce qu'un système doit faire, et vous obtiendrez une liste de fonctionnalités : rechercher un vol, payer un panier, prendre un rendez-vous. Demandez-leur à quel point il doit bien le faire, et les réponses sont généralement plus vagues : il doit être rapide, il ne doit pas tomber en panne, il doit être sécurisé, il doit être facile à modifier. Ces secondes réponses sont des attributs de qualité, et c'est surtout d'eux que traite l'architecture. La même liste de fonctionnalités peut être fournie par un programme unique sur un seul serveur ou par quarante services répartis sur trois continents. Ce qui distingue ces

conceptions, ce n'est pas ce que fait le système, mais le nombre de personnes qu'il doit servir en même temps, la durée d'indisponibilité que l'entreprise peut tolérer, la fréquence à laquelle le code changera et la sensibilité des données.

Le chapitre 1 a décrit l'architecture comme l'ensemble des décisions importantes qui répondent aux préoccupations des parties prenantes au moyen de compromis. Ce chapitre rend ces préoccupations précises. Une exigence qui a un effet mesurable sur la structure d'un système est une exigence architecturalement significative, et la plupart d'entre elles s'avèrent être des attributs de qualité ou des contraintes plutôt que des fonctions. Les repérer tôt, et les formuler de façon à pouvoir les tester, est l'une des choses les plus précieuses que fasse un architecte.

Comment s'articule le chapitre

La section 2.1 distingue exigences fonctionnelles, attributs de qualité et contraintes, explique pourquoi de nombreux architectes évitent le terme « exigences non fonctionnelles », et étudie un système de billetterie de concerts le jour où une grande tournée est mise en vente. La section 2.2 présente le modèle de qualité des produits de l'ISO/IEC 25010:2023, avec ses neuf caractéristiques, et montre comment l'utiliser comme aide-mémoire plutôt que comme formulaire à remplir. La section 2.3 enseigne la technique la plus utile du chapitre : le scénario d'attribut de qualité en six parties, qui transforme « doit être rapide » en un énoncé que vous pouvez tester.

Les sections 2.4 à 2.7 passent ensuite en revue les qualités qui façonnent le plus souvent les architectures. La section 2.4 traite des performances et de la scalabilité avec de vrais calculs : latence et débit, percentiles, latence de queue (tail latency), loi de Little, files d'attente et loi d'Amdahl. La section 2.5 fait de même pour la disponibilité et la fiabilité : les « neuf », le temps moyen entre défaillances et le temps moyen de réparation, et la disponibilité de composants en série et en parallèle. La section 2.6 couvre les qualités liées au changement (modifiabilité, testabilité et déployabilité), et la section 2.7 rassemble la sécurité, l'utilisabilité et l'accessibilité, l'interopérabilité, le coût, l'efficacité énergétique, la sûreté et les qualités réglementaires.

La section 2.8 boucle la boucle. Elle montre d'où viennent les exigences de qualité, comment un atelier sur les attributs de qualité (Quality Attribute Workshop) et un arbre d'utilité transforment une foule d'opinions en une liste classée, quelles qualités entrent en conflit avec lesquelles, et comment les fonctions d'aptitude (fitness functions) continuent de vérifier automatiquement les qualités importantes une fois l'architecture construite.

Des souhaits vagues aux chiffres

C'est le chapitre où le cours commence à calculer. Vous calculerez combien de requêtes sont en cours en même temps à partir d'un taux d'arrivée et d'un temps de réponse, vous verrez pourquoi une requête adressée à un serveur occupé à 90 % prend, en moyenne, dix fois son temps de service, pourquoi répartir un travail séquentiel à 10 % sur seize processeurs ne donne qu'une accélération d'un facteur 6,4, et combien de minutes d'indisponibilité par an une promesse de « quatre neuf » autorise réellement. Aucune de ces mathématiques n'est plus difficile que l'arithmétique de l'école, mais elles changent les conversations : « la base de données est lente » devient « le pool de connexions est dimensionné pour 100 requêtes simultanées et la loi de Little dit qu'il nous en faut 400 pendant le pic du soir ».

Les chiffres dissipent aussi les fausses certitudes. Un temps de réponse moyen de 120 millisecondes peut masquer le fait qu'une requête sur cinquante prend plus d'une seconde : c'est pourquoi les professionnels spécifient le 95e ou le 99e percentile. Une exigence sans mesure de la réponse ne peut pas être testée, et une exigence qui ne peut pas être testée sera discrètement sacrifiée à l'approche d'une échéance.

Les compromis sont le cœur du métier

Aucun système ne peut être à la fois le plus rapide, le plus disponible, le plus sécurisé, le moins cher et le plus facile à modifier possible. La mise en cache améliore la vitesse mais complique la cohérence ; le chiffrement et les contrôles supplémentaires protègent les données mais ajoutent de la latence ; la redondance augmente à la fois la disponibilité et le coût ; les conceptions souples à base de plug-ins facilitent le changement et rendent les performances plus difficiles à prévoir. Le chapitre traite donc les qualités comme des objectifs concurrents qu'il faut classer selon le contexte, et non comme des cases à cocher. Le talent de l'architecte consiste à savoir quelles sont les deux ou trois qualités qui comptent le plus pour ce système, à les formuler avec précision et à choisir des conceptions dont les faiblesses tombent là où l'entreprise peut se les permettre.

Les techniques de conception qui permettent d'obtenir chaque qualité, appelées tactiques, ne sont présentées ici que dans les grandes lignes. Les chapitres suivants les développent : les principes de conception au chapitre 3, les styles architecturaux aux chapitres 5 et 6, les pratiques relatives aux données, à la sécurité, à la résilience et au cloud aux chapitres 7 à 10, et les méthodes d'évaluation comme l'ATAM et les fonctions d'aptitude au chapitre 11.

L’idée à retenir

Si vous ne retenez qu’une chose de ce chapitre, que ce soit celle-ci : une qualité n’est pas une exigence tant qu’elle n’a pas de stimulus, de contexte et de mesure. « L’étape de paiement doit être rapide » est un souhait. « Lorsque 2000 clients par minute passent commande pendant le pic du vendredi soir, 99 % des paiements aboutissent en 1,5 seconde au plus » est une exigence. On peut concevoir en fonction d’elle, la tester, la superviser et en débattre preuves à l’appui, et elle indique à chacun, du product owner à l’équipe d’exploitation, à quoi ressemble la réussite.

SECTION 2.1 SUR 2.8 · INTERMÉDIAIRE · 23 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Exigences fonctionnelles, attributs de qualité et contraintes

AVANT DE LIRE

Cette section s'appuie sur :

- 0.8 Comment les logiciels sont construits
- 1.3 Pourquoi c'est important
- 1.5 Parties prenantes et préoccupations

Termes déjà vus qu'elle utilise : Partie prenante, Récit utilisateur (user story), Compromis

L'IDÉE EN BREF

Pensez à la commande d'un gâteau de mariage. « Trois étages, génoise au citron, glaçage blanc » décrit ce qu'est le gâteau : ses fonctions. « Prêt pour 14 h samedi, capable de supporter 40 minutes de route, sans danger pour un invité allergique aux fruits à coque » décrit à quel point il doit bien fonctionner : ses qualités. « Le lieu de réception n'accepte que les gâteaux de pâtissiers agréés » est une contrainte : un choix déjà fait pour vous. Un pâtissier peut réaliser la même génoise au citron de façons très différentes, et ce sont les qualités et les contraintes, et non le parfum, qui décident de la façon dont le gâteau est construit et transporté. Il en va de même pour les logiciels.

CE QU'IL FAUT SAVOIR

- Les exigences fonctionnelles indiquent ce qu'un système doit faire ; les attributs de qualité indiquent avec quel niveau de qualité il doit le faire ; les contraintes sont des décisions déjà prises, comme les réglementations, les technologies imposées, les délais et les budgets.
- De nombreux architectes évitent le terme « exigences non fonctionnelles », car il définit une vaste catégorie par exclusion, laisse entendre que ces exigences sont séparées des fonctions et semble facultatif.
- Une exigence architecturalement significative (ASR) est une exigence dont la modification imposerait de changer la structure du système ; la plupart des ASR sont des attributs de qualité ou des contraintes, mais certaines fonctions sont aussi significatives.
- Les mêmes fonctions peuvent être fournies par des architectures très différentes ; ce sont les attributs de qualité et les contraintes qui permettent de choisir entre elles.

> - Les exigences implicites, comme ne jamais perdre de commande ou ne jamais débiter deux fois, sont rarement écrites et doivent être activement recueillies auprès des parties prenantes et tirées des incidents et des réglementations.

Trois types d'exigences

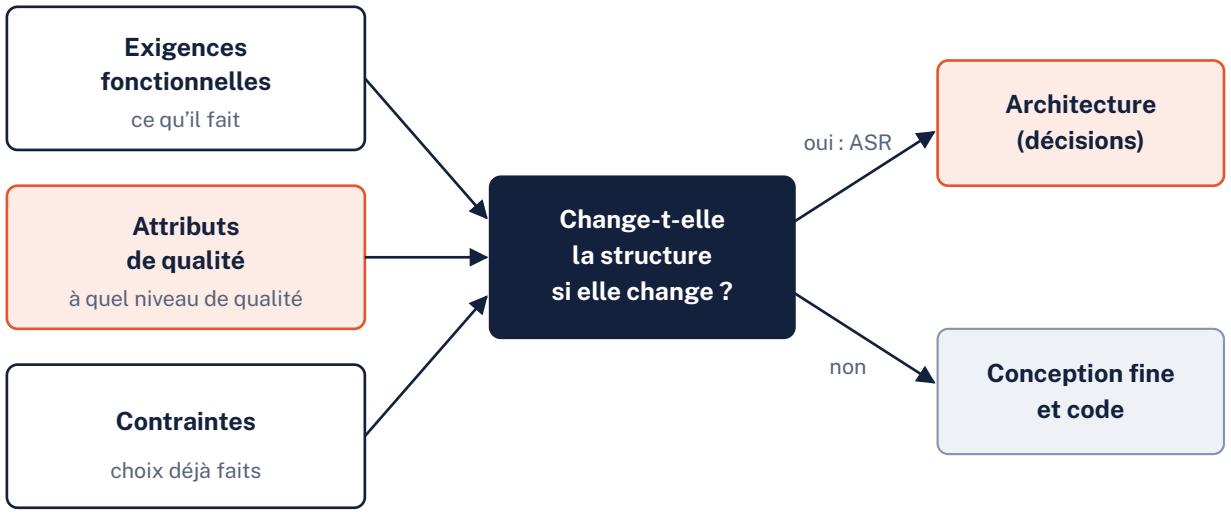
Une exigence est tout ce qu'un système doit satisfaire pour être acceptable aux yeux de ses parties prenantes. Les architectes trouvent utile de classer les exigences en trois types, car chaque type influence la conception d'une manière différente. Une exigence fonctionnelle indique ce que le système doit faire : « un client peut ajouter un article au panier », « le système de la pharmacie vérifie chaque nouvelle ordonnance au regard des allergies du patient », « un passager peut voir la position du chauffeur sur une carte ». Les exigences fonctionnelles décrivent le comportement, les données et les règles qui les relient. C'est ce que capturent la plupart des récits utilisateur (user stories) et des cas d'utilisation.

Un attribut de qualité indique à quel point le système doit bien le faire. « Les résultats de recherche s'affichent en moins d'une demi-seconde », « le service de réservation est disponible 99,95 % du temps », « un nouveau prestataire de paiement peut être ajouté en moins de deux semaines », « les dossiers des patients ne sont lisibles que par les cliniciens qui traitent ce patient ». Les attributs de qualité sont des propriétés du système dans son ensemble ou d'une fonction, et ils n'ont de sens qu'assortis d'une mesure : assez rapide pour qui, sous quelle charge, mesuré comment ?

Une contrainte est une décision qui a déjà été prise pour vous. « Le système doit tourner dans le tenant Azure existant de l'entreprise », « toutes les données à caractère personnel restent dans l'Union européenne », « l'équipe connaît Java et n'a pas de budget pour se former à autre chose », « la première version doit être en production avant le début de l'exercice fiscal, le 6 avril ». Les contraintes suppriment des options plutôt que de décrire un comportement. Elles ne sont pas négociables au sein du projet, même si un bon architecte vérifie que chacune est réellement une contrainte et pas seulement une habitude ou une hypothèse.

Les frontières ne sont pas toujours nettes, et ce n'est pas grave. Le but de la classification n'est pas l'ordre bureaucratique ; il est de s'assurer que personne ne recueille cinquante récits utilisateur en oubliant de demander combien d'utilisateurs il y aura, combien de temps l'entreprise peut survivre à une panne, ou quel régulateur surveille.

Comment les exigences atteignent l'architecture



Les trois types d'exigences alimentent l'architecture, mais seules celles dont la modification imposerait un changement structurel sont architecturalement significatives.

Pourquoi de nombreux architectes évitent le terme « exigences non fonctionnelles »

Les documents d'exigences séparent traditionnellement tout en exigences fonctionnelles et exigences non fonctionnelles (NFR). Le terme est partout, dans les contrats, les dossiers d'appel d'offres et les offres d'emploi, il faut donc savoir le reconnaître. De nombreux architectes et chercheurs l'évitent pourtant, pour trois raisons.

Premièrement, il définit une catégorie vaste et importante par ce qu'elle n'est pas. Tout ce qui n'est pas une fonction finit dans le même sac : temps de réponse, règles de chiffrement, choix du fournisseur de cloud et couleur du logo. L'article de Martin Glinz *On Non-Functional Requirements* (IEEE International Requirements Engineering Conference, 2007) a montré que les définitions publiées ne s'accordent pas sur ce qui appartient à cette catégorie, et qu'un même besoin peut être classé d'un côté ou de l'autre selon sa formulation : « seul le personnel autorisé peut voir les salaires » se lit comme une fonction lorsqu'il est écrit comme une règle d'accès, et comme une qualité lorsqu'il est écrit comme un objectif de sécurité.

Deuxièmement, le nom laisse entendre que ces exigences sont séparées des fonctions et peuvent être ajoutées plus tard. Ce n'est pas le cas. Un attribut de qualité qualifie une fonction : c'est l'étape de paiement qui doit être rapide, le paiement qui ne doit jamais être perdu, le dossier médical qui doit rester confidentiel. Bass, Clements et Kazman, dans *Software Architecture in Practice*, parlent donc d'exigences d'attributs de qualité et traitent les contraintes comme une catégorie distincte.

Troisièmement, « non fonctionnel » semble facultatif, comme si un système qui marche mais qui est lent, peu sûr ou impossible à modifier « fonctionnait » quand même. En pratique, ce sont exactement ces systèmes-là que l'on réécrit. Certaines équipes préfèrent « exigences de qualité » ; d'autres, notamment chez Thoughtworks, disent « exigences transverses » (cross-functional requirements) pour souligner qu'elles traversent toutes les fonctionnalités. Ce cours parle d'attributs de qualité et de contraintes. Si votre organisation parle de NFR, inutile de se battre sur le mot ; assurez-vous simplement que chaque NFR est soit une qualité mesurable, soit une contrainte clairement formulée.

Contraintes : techniques, organisationnelles, réglementaires, de délai et de budget

Les contraintes méritent plus d'attention qu'on ne leur en accorde habituellement, car elles déterminent souvent une plus grande part de l'architecture que n'importe quelle fonctionnalité. Une contrainte est simplement une exigence qui ne laisse plus aucune liberté. Il est utile de les regrouper selon leur origine.

Types de contraintes et leur effet sur l'architecture

Type	Exemple	Effet typique sur l'architecture
Technique	Doit s'intégrer au système financier SAP existant par ses interfaces publiées ; doit tourner sur la plateforme Kubernetes de l'entreprise.	Fixe les styles d'intégration, les formats de données et le modèle de déploiement ; peut exclure certaines bases de données ou certains langages.
Organisationnelle	Trois équipes sur deux fuseaux horaires ; l'équipe d'exploitation ne prend en charge que PostgreSQL ; l'équipe sécurité doit approuver chaque dépendance externe.	Détermine la façon dont le système est découpé (loi de Conway, chapitre 3) et les technologies réalistes.
Réglementaire et juridique	Les données à caractère personnel des résidents de l'UE sont traitées dans le cadre du RGPD ; les données de cartes relèvent de PCI DSS ; un dispositif médical doit respecter les réglementations en matière de sécurité.	Détermine la localisation des données, le chiffrement, la journalisation d'audit, les règles de conservation et la séparation des composants sensibles.
Durée	Les nouvelles règles fiscales s'appliquent à partir d'une date fixe ; une plateforme de commerce de détail doit être prête avant le pic de novembre.	Favorise les technologies éprouvées et les conceptions plus simples ; peut imposer une mise en production par étapes avec une dette technique connue.
Budget	Les coûts de fonctionnement doivent rester sous un montant mensuel fixe ; pas d'argent pour des licences commerciales.	Limite la redondance, les régions et les services managés ; fait du coût par transaction une qualité explicite.

Deux habitudes aident. D'abord, notez la source de chaque contrainte (une réglementation, un contrat, un décideur nommément désigné) afin qu'elle puisse être vérifiée et, si les circonstances changent, réexaminée. Ensuite, distinguez les vraies contraintes des préférences déguisées en contraintes. « Nous devons utiliser des microservices » est rarement une contrainte ; c'est une décision de conception que quelqu'un a prise tôt, et elle doit être justifiée par des attributs de qualité comme toute autre décision. La section du chapitre 1 sur les décisions et les compromis explique pourquoi les choix précoces et difficiles à défaire méritent cet examen.

Exigences architecturalement significatives

La plupart des exigences n'affectent pas l'architecture. Que la facture affiche le code postal du client au-dessus ou en dessous du nom de la rue est un détail ; cela peut se changer en un après-midi. Une exigence architecturalement significative (ASR) est une exigence qui a un effet mesurable sur l'architecture : si elle changeait, la structure du système devrait changer. Repérer les ASR tôt est important, car il coûte cher d'en tenir compte tardivement. Ajouter une salle d'attente à un site de billetterie la semaine précédant une mise en vente est possible ; partitionner l'inventaire des places sur plusieurs serveurs après le lancement revient à tout réécrire.

Lianping Chen, Muhammad Ali Babar et Bashar Nuseibeh ont interrogé des praticiens sur la façon dont ils reconnaissent les ASR (*Characterizing Architecturally Significant Requirements*, IEEE Software, 2013). Leurs conclusions rejoignent l'expérience de la plupart des architectes. Les ASR sont généralement difficiles à satisfaire, elles concernent souvent le système dans son ensemble plutôt qu'une seule fonctionnalité, elles entrent fréquemment en conflit les unes avec les autres, et elles sont souvent formulées de façon vague, voire pas du tout. Les attributs de qualité et les contraintes en constituent la majorité, mais certaines exigences fonctionnelles sont aussi significatives.

Exemples d'ASR fonctionnelles : « chaque modification d'un dossier patient doit pouvoir être rattachée à une personne et à un moment » (il faut un mécanisme d'audit dans tous les composants), « les utilisateurs peuvent collaborer sur le même document en temps réel » (cela exige un modèle de synchronisation et probablement un autre style de communication), et « les clients peuvent annuler un virement bancaire dans les 30 minutes » (cela affecte la façon dont l'argent circule entre les comptes). Chacune impose une décision structurelle, pas seulement davantage de code.

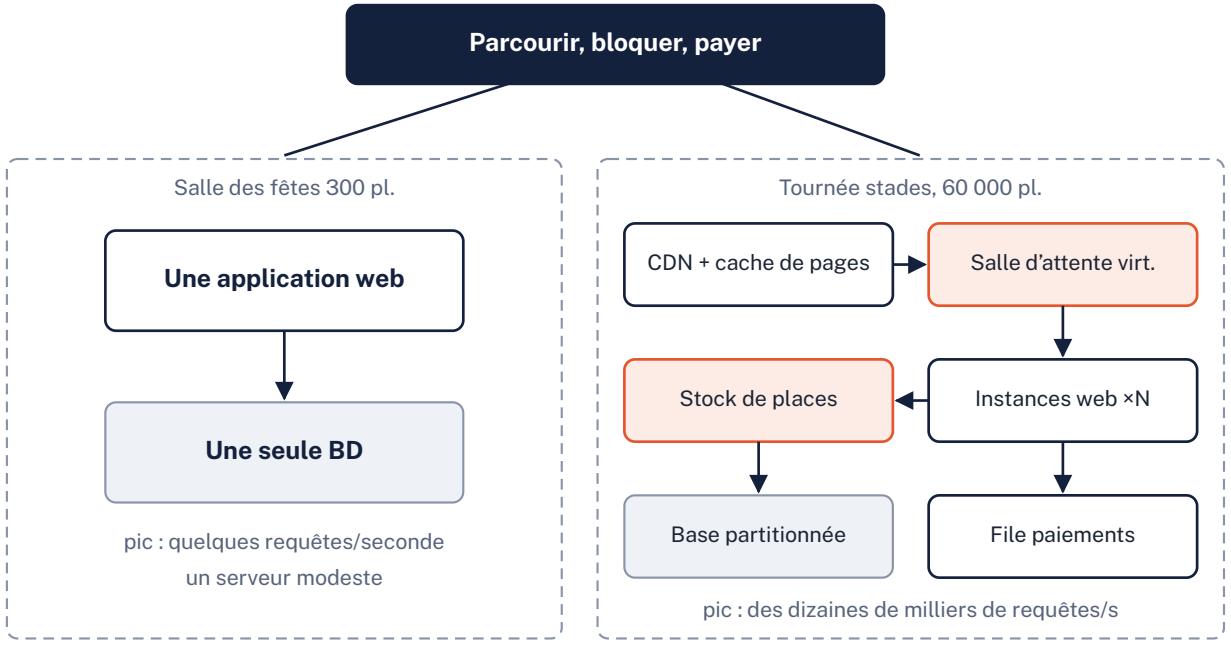
Pourquoi les qualités façonnent l'architecture davantage que les fonctions

Voici l'idée centrale de ce chapitre. Si seule la fonctionnalité comptait, presque n'importe quelle structure ferait l'affaire : vous pourriez écrire tout le système comme un seul gros programme, et il calculerait toujours les bonnes réponses. Bass, Clements et Kazman le disent directement : la fonctionnalité ne détermine pas l'architecture, car de nombreuses architectures différentes peuvent fournir les mêmes fonctions. Ce qui les distingue, c'est la manière dont chacune atteint les qualités.

Prenons deux systèmes de billetterie aux fonctionnalités identiques : parcourir les événements, bloquer des places, payer. L'un vend des billets pour une salle des fêtes de 300 places ; l'autre vend une tournée dans des stades de 60 000 places. Le moment le plus chargé de la salle des

fêtes apportera peut-être quelques requêtes par seconde, et une seule application web avec une seule base de données est la bonne réponse : bon marché, simple et facile à modifier. La tournée des stades doit absorber un afflux des milliers de fois plus important, ne doit jamais vendre deux fois la même place et doit empêcher les bots de tout acheter dans la première seconde. Ce système a besoin d'un réseau de diffusion de contenu, d'une file d'attente devant la boutique, d'un inventaire des places soigneusement conçu, d'un stockage partitionné et d'un flux de paiement capable d'attendre un prestataire surchargé. Mêmes fonctions, architectures complètement différentes, et la différence vient entièrement des attributs de qualité et des contraintes.

Mêmes fonctions, qualités différentes



Des fonctionnalités identiques peuvent conduire à des structures radicalement différentes lorsque les exigences de charge, de cohérence et de sécurité diffèrent.

SI VOUS DÉBUTEZ

Pas à pas : cette exigence est-elle significative ?

L'idée la plus difficile de cette section consiste à savoir quelles exigences sont architecturalement significatives (ASR). Le critère n'est pas de savoir si une exigence est appelée fonction ou qualité, mais si sa modification changerait la structure du système. Essayez ce critère sur trois exigences.

- Le critère : imaginez que l'exigence change (le chiffre doublé ou réduit, la règle durcie ou supprimée) et demandez-vous ce qui devrait bouger : des composants, des données ou les chemins entre eux.
- Une facture affiche le code postal du client sous le nom de la rue. Placez-le au-dessus et un seul modèle de document change, en un après-midi. Rien d'autre ne bouge : il s'agit donc de conception détaillée, pas d'architecture.
- La tournée des stades doit absorber un afflux des milliers de fois plus important que les quelques requêtes par seconde de la salle des fêtes. Cet afflux explique en grande partie pourquoi il lui faut un réseau de diffusion de contenu, une salle d'attente et un stockage partitionné. Ramenez la charge à celle de la salle des fêtes et la plus grande partie de cette machinerie pourrait disparaître ; la salle des fêtes se contente d'une application web et d'une base de données.
- Le chiffre de charge est donc une ASR. Ce qui le rend significatif, c'est son effet sur la structure, et non son étiquette de qualité.
- Les fonctions peuvent aussi passer le critère. Prenez la règle selon laquelle chaque modification du dossier d'un patient doit indiquer qui l'a faite et quand. Elle ressemble à une fonctionnalité, mais elle exige un mécanisme d'audit dans tous les composants. Supprimez-la, et ce mécanisme disparaît : elle est donc significative.
- **La leçon :** demandez-vous « qu'est-ce que ce changement ferait bouger ? », et non « de quel type d'exigence s'agit-il ? ». La plupart des exigences, de tous types, restent à l'intérieur d'une seule partie. Les rares qui feraient bouger des composants, des données ou les chemins entre eux sont celles autour desquelles l'architecture doit être construite.

Les qualités implicites que personne n'écrit

Certaines des qualités les plus importantes ne sont jamais énoncées, car toutes les parties prenantes les tiennent pour acquises. Ces exigences implicites n'apparaissent que lorsqu'elles sont violées, généralement en production. Personne n'écrit « ne pas perdre de commandes », « ne pas débiter deux fois le client », « continuer à fonctionner lors du passage à l'heure d'été », « afficher le même solde sur le téléphone et sur le site web » ou « ne pas divulguer les données

d’un client à un autre ». Pourtant, chacune est une ASR aux lourdes conséquences : traitement idempotent des paiements, écritures durables, planification robuste aux changements d’heure, cohérence entre les canaux et isolation stricte entre clients.

Les exigences implicites se cachent pour des raisons sociales. Les parties prenantes métier les jugent trop évidentes pour être mentionnées ; les développeurs supposent que quelqu’un d’autre s’en est occupé ; et les gabarits d’exigences y font rarement penser. Voici des questions utiles pour les débusquer : « Qu’est-ce qui vous mettrait hors de vous si cela arrivait ? », « Qu’est-ce qui nous ferait passer dans les journaux ? », « Qu’est-ce qui a mal tourné avec le système que nous remplaçons ? » et « Pour quoi réveille-t-on l’équipe d’exploitation ? ». Les rapports d’incident et les tickets de support des systèmes précédents en sont une source riche ; il en va de même des recommandations publiées par les régulateurs et des obligations figurant dans les contrats clients.

EXEMPLE COMMENTÉ

Exemple commenté : un système de billetterie de concerts le jour de la mise en vente

Une société de billetterie va mettre en vente quatre soirs de concerts dans un stade (60 000 places par soir) un vendredi à 10 h 00. Un jour normal, le site reçoit environ 2000 visiteurs par heure. Le marketing s'attend à ce que 1,5 million de personnes arrivent dans les dix premières minutes. Les fonctionnalités sont les habituelles : parcourir, choisir des places, les bloquer pendant dix minutes, payer. Quelles sont les exigences architecturalement significatives ?

Billets en vente	4 soirs × 60 000 places = 240 000 billets
Taux d'arrivée normal	2000 visiteurs ÷ 3600 s ≈ 0,56 visiteur par seconde
Taux d'arrivée le jour de la vente	1 500 000 visiteurs ÷ 600 s = 2500 visiteurs par seconde, soit environ 4500 fois le taux normal
Débit de requêtes	Si le navigateur de chaque visiteur envoie environ 8 requêtes au site dans la première minute (pages, plans de salle, appels d'API) : 2500 × 8 = 20 000 requêtes par seconde
Demande et offre	1 500 000 visiteurs pour 240 000 billets : la plupart des gens seront déçus quelle que soit la conception, donc l'équité et une communication claire comptent autant que la vitesse
Contrainte de paiement	Le contrat du prestataire de paiement autorise 300 autorisations de paiement par seconde. Avec une moyenne de 2,5 billets par commande, 240 000 ÷ 2,5 = 96 000 commandes ; 96 000 ÷ 300 = 320 s, donc les paiements à eux seuls prennent au moins 5,3 minutes, même si tout le reste est instantané
ASR 1 : charge de pointe (performances)	Absorber 2500 arrivées par seconde sans s'effondrer ; servir les pages statiques et les plans de salle depuis un cache ou un CDN, afin que seuls les vrais achats atteignent le cœur du système
ASR 2 : aucune place vendue deux fois (cohérence)	Le blocage des places doit être atomique, même lorsque des milliers de personnes cliquent sur le même bloc au même moment ; cela façonne la conception de l'inventaire et son stockage
ASR 3 : équité (règle métier à impact structurel)	Admettre les acheteurs dans leur ordre d'arrivée grâce à une salle d'attente virtuelle, afin que le cœur du système voie un flux contrôlé plutôt qu'une ruée
ASR 4 : résistance aux bots (sécurité)	Détecter et bloquer les acheteurs automatisés avant qu'ils n'atteignent l'inventaire ; limiter le débit par compte et par appareil

ASR 5 : dégradation gracieuse (disponibilité)	Si les paiements ralentissent, maintenir la validité des places bloquées et mettre les tentatives de paiement en file d'attente plutôt que de faire échouer des commandes entières
Contrainte	Date et heure fixes annoncées à l'avance ; limite du prestataire de paiement de 300 par seconde

Aucune de ces ASR ne figure dans la liste des fonctionnalités. Toutes découlent de la charge, de la cohérence, de l'équité, de la sécurité et d'une contrainte imposée par un tiers.

La limite de paiement se traduit directement en décision de conception : comme les acheteurs ne peuvent pas tous payer en même temps, le système doit bloquer les places et mettre les paiements en file d'attente, ce qui détermine à son tour la durée du blocage.

Le monde réel offre un rappel. En novembre 2022, la prévente de Ticketmaster pour une grande tournée des stades a subi de longs retards et des défaillances sous l'effet de la demande ; l'entreprise a fait état de 3,5 milliards de requêtes système au total, quatre fois son précédent pic.

Liste de contrôle : repérer les exigences architecturalement significatives

Passez ces questions en revue pour chaque exigence candidate, et pour le système dans son ensemble, tôt dans le projet.

- [ ] **Sa modification changerait-elle la structure ?**
  Si doubler le chiffre, assouplir la règle ou supprimer l'exigence ferait bouger des composants, des données ou des chemins de communication, elle est significative.
- [ ] **S'applique-t-elle à de nombreuses fonctionnalités ?**
  Les exigences qui traversent tout le système (audit, sécurité, disponibilité, internationalisation) appellent généralement une réponse structurelle.
- [ ] **Est-elle difficile ou nouvelle ?**
  Un chiffre exigeant (moins de 100 ms, 99,99 %), une charge très supérieure à tout ce que l'équipe a déjà géré ou une fonctionnalité inédite sont des signaux forts.
- [ ] **Porte-t-elle une forte valeur métier ou un risque élevé ?**
  Si ne pas la satisfaire ferait perdre du chiffre d'affaires, enfreindrait la loi ou causerait du tort à quelqu'un, traitez-la comme significative même si elle paraît simple.
- [ ] **Entre-t-elle en conflit avec une autre qualité ?**
  Les exigences en tension (rapide et fortement cohérent, ouvert et sécurisé) nécessitent des compromis délibérés, qui sont des décisions architecturales.
- [ ] **S'agit-il d'une contrainte déguisée ?**
  Les technologies imposées, les réglementations, les échéances et les budgets suppriment des options ; consignez-les avec leur source.
- [ ] **A-t-elle déjà causé des problèmes ?**
  Examinez les incidents, les tickets de support et les post-mortems du système remplacé, ainsi que de systèmes similaires ailleurs.
- [ ] **Quelqu'un la suppose-t-il en silence ?**
  Demandez ce qui mettrait les parties prenantes hors d'elles ou ferait parler de l'entreprise dans les médias ; ces réponses révèlent des exigences implicites.

ATTENTION

Erreurs courantes

- Recueillir des centaines de récits utilisateur et une seule ligne disant « le système doit être rapide, sécurisé et scalable ».
- Traiter chaque exigence comme significative. Si tout est une ASR, rien n'est priorisé ; la plupart des exigences relèvent de la conception détaillée.
- Accepter une décision de conception (« utiliser Kafka », « utiliser des microservices ») comme s'il s'agissait d'une contrainte, sans attribut de qualité qui la justifie.
- Rédiger des qualités sans mesures. On ne peut ni concevoir ni tester pour « hautement disponible » ; la section 2.3 montre comment y remédier.
- Supposer que la qualité peut être ajoutée une fois les fonctionnalités construites. Les performances, la sécurité et la disponibilité dépendent de la structure et coûtent cher à ajouter après coup.

Comment les professionnels travaillent avec les ASR

Les architectes expérimentés tiennent une liste courte et vivante d'ASR, généralement de cinq à quinze éléments, plutôt qu'un long document d'exigences. Chaque élément formule la qualité sous forme de scénario testable (section 2.3), nomme la partie prenante concernée, consigne sa priorité et renvoie aux décisions qu'il a orientées. Le modèle arc42, par exemple, demande les trois à cinq principaux objectifs de qualité dès sa première page, puis une liste plus complète des exigences de qualité plus loin. La liste est revue chaque fois qu'une décision importante est prise ou que l'entreprise change de cap, et les éléments les plus importants finissent par être adossés à des vérifications automatisées appelées fonctions d'aptitude, présentées à la section 2.8.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ « Les patients peuvent prendre rendez-vous en ligne » : est-ce une exigence fonctionnelle, un attribut de qualité ou une contrainte ?

Une exigence fonctionnelle : elle indique ce que le système doit faire. Une qualité ajouterait à quel point il doit bien le faire, par exemple la rapidité de confirmation des réservations ou le nombre de réservations que l'on peut traiter le lundi à 8 h.

▼ **Pourquoi « le système doit être scalable » n'est-il pas encore une exigence utile ?**

Elle n'a ni stimulus, ni contexte, ni mesure. Elle ne dit pas quelle charge, à quelle vitesse la charge augmente, ni quelle réponse doit être préservée, si bien que personne ne peut concevoir en fonction d'elle ni la tester.

▼ **Donnez un exemple d'exigence fonctionnelle architecturalement significative.**

Par exemple, une piste d'audit complète de chaque modification des dossiers médicaux, ou l'édition collaborative en temps réel. Chacune nécessite un mécanisme structurel dans tout le système, et pas seulement davantage de code à un seul endroit.

▼ **Quelle question permet le mieux de distinguer une exigence significative d'une exigence ordinaire ?**

La structure du système devrait-elle changer si cette exigence changeait ? Si oui, elle est architecturalement significative.

▼ **À vous : Un énoncé de billetterie indique : les fans peuvent choisir leurs places ; confirmation en moins de 2 secondes au pic du jour de la mise en vente ; l'hébergement doit rester chez le fournisseur cloud actuel ; personne n'est débité deux fois. Classez chaque élément et indiquez lesquels sont architecturalement significatifs.**

Le choix des places est fonctionnel ; il compte, mais il décide à peine de la structure. La confirmation en moins de 2 secondes en période de pointe est un attribut de qualité, clairement significatif, car elle commande les files et la capacité. La règle d'hébergement est une contrainte, une décision déjà prise, et elle est significative parce qu'elle limite les options. Ne facturer personne deux fois est une qualité implicite, généralement non écrite et très significative : elle façonne l'idempotence et la façon d'enregistrer les paiements.

SOURCES

- Martin Glinz, On Non-Functional Requirements (RE 2007): https://files.ifi.uzh.ch/rerg/amadeus/publications/papers/RE07_Glinz.pdf
- Chen, Ali Babar and Nuseibeh, Characterizing Architecturally Significant Requirements (IEEE Software, 2013): https://pure.itu.dk/en/publications/characterizing-architecturally-significant-requirements/
- SEI: Software Architecture in Practice, 4th edition: https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/
- arc42: Introduction and goals (quality goals): https://docs.arc42.org/section-1/

SECTION 2.2 SUR 2.8 · INTERMÉDIAIRE · 21 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Les modèles de qualité : l'ISO/IEC 25010 et comment l'utiliser

AVANT DE LIRE

Cette section s'appuie sur :

- 1.5 Parties prenantes et préoccupations
- 2.1 Exigences fonctionnelles et de qualité

Termes déjà vus qu'elle utilise : Attribut de qualité, Partie prenante, Exigence

L'IDÉE EN BREF

Quand vous achetez une voiture d'occasion, une bonne liste de contrôle vous évite d'oublier des choses : moteur, freins, pneus, rouille, papiers, consommation de carburant. Vous n'avez pas besoin que chaque point soit parfait, mais vous voulez avoir examiné chacun d'eux et décidé s'il compte pour vous. Un modèle de qualité est cette liste de contrôle pour le logiciel. La norme internationale ISO/IEC 25010 énumère neuf domaines, comme les performances, la fiabilité, la sécurité et la facilité à modifier le logiciel, afin que rien d'important ne soit oublié. Tout l'art consiste à s'en servir pour susciter de bonnes questions, et non pour remplir des formulaires.

CE QU'IL FAUT SAVOIR

- Un modèle de qualité fournit un vocabulaire commun de caractéristiques et de sous-caractéristiques de qualité et sert de liste de contrôle pour repérer les qualités qui comptent.
- L'ISO/IEC 25010:2023 définit neuf caractéristiques de produit : adéquation fonctionnelle, efficacité des performances, compatibilité, capacité d'interaction, fiabilité, sécurité, maintenabilité, flexibilité et sûreté.
- Par rapport à 2011, l'utilisabilité est devenue la capacité d'interaction, la portabilité est devenue la flexibilité, la sûreté a été ajoutée et la qualité en utilisation est passée dans une norme distincte, l'ISO/IEC 25019:2023.
- D'autres listes, comme FURPS+, les qualités de Software Architecture in Practice et Q42, recoupent largement l'ISO/IEC 25010 ; aucune n'est la seule correcte.
- Utilisez un modèle pour susciter une décision rapide et explicite sur chaque caractéristique, puis ne transformez que les caractéristiques pertinentes en quelques scénarios mesurables et hiérarchisés.

-

À quoi sert un modèle de qualité

La section 2.1 a soutenu que les attributs de qualité déterminent l'essentiel d'une architecture. La question suivante est évidente : quelles qualités ? Livrées à elles-mêmes, les parties prenantes citent les deux ou trois qui les ont fait souffrir en dernier, généralement la vitesse et la disponibilité, et oublient le reste jusqu'à ce qu'il soit trop tard. Un modèle de qualité est une liste structurée de caractéristiques de qualité, chacune découpée en sous-caractéristiques plus fines, avec des noms et des définitions convenus. Il remplit deux fonctions. Il donne à tous un vocabulaire commun, afin que « fiabilité » signifie la même chose pour le product owner, le testeur et le fournisseur. Et il sert de liste de contrôle, en vous incitant à envisager des qualités que personne n'a encore évoquées.

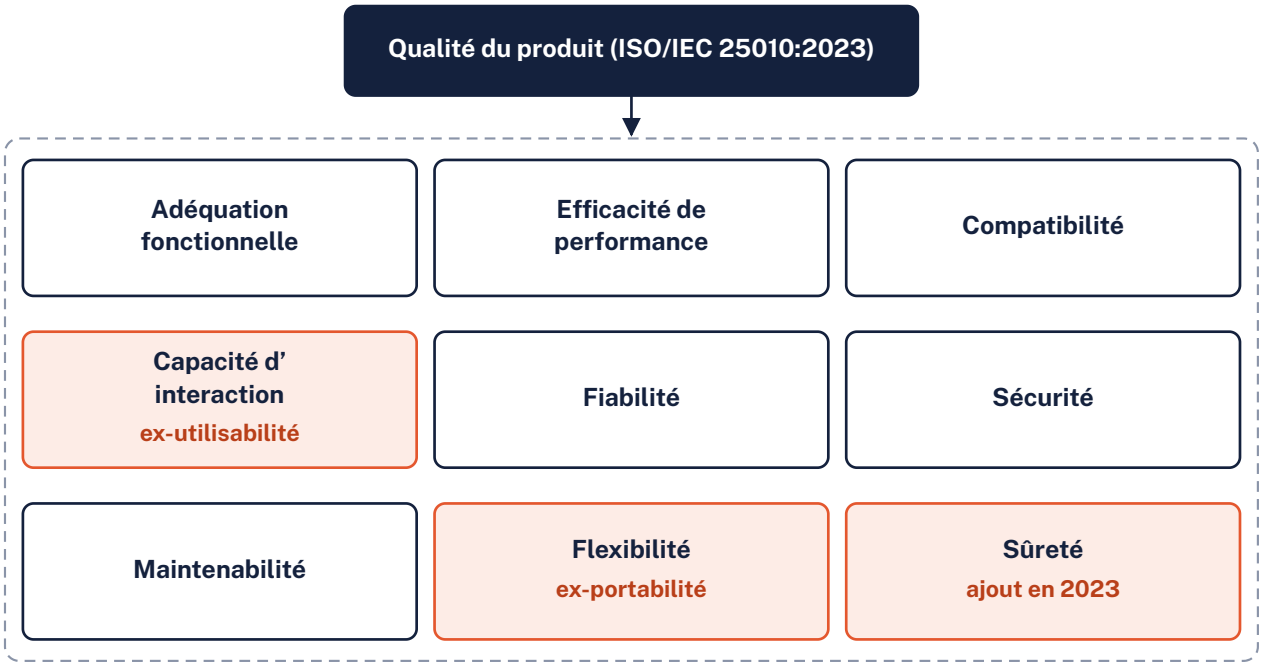
Un modèle ne vous dit pas quelles qualités comptent pour votre système, ni quel niveau de chacune il vous faut. Cela vient des parties prenantes et des objectifs métier (section 2.8). Il ne vous dit pas non plus comment les atteindre ; c'est le rôle des tactiques, des patrons et du reste de ce cours. C'est une carte du territoire, pas un itinéraire pour le traverser.

L'ISO/IEC 25010:2023 : le modèle de qualité des produits

Le modèle le plus souvent cité est l'ISO/IEC 25010, qui fait partie de la famille de normes SQuaRE (Systems and software Quality Requirements and Evaluation). La deuxième édition, publiée en novembre 2023, définit un modèle de qualité des produits en neuf caractéristiques. Elle s'applique aux produits logiciels et, plus largement, aux produits TIC, et la norme en énumère les usages : spécifier des exigences, vérifier que les exigences sont complètes, fixer des objectifs de conception et de test, définir des critères d'acceptation et établir des mesures de qualité. D'autres normes de la famille définissent comment mesurer chaque sous-caractéristique (ISO/IEC 25023) et comment évaluer un produit.

Les neuf caractéristiques sont l'adéquation fonctionnelle, l'efficacité des performances, la compatibilité, la capacité d'interaction, la fiabilité, la sécurité, la maintenabilité, la flexibilité et la sûreté. La première indique si le système fait les bonnes choses correctement ; les huit autres décrivent à quel point il les fait bien et à quel point il peut être exploité, protégé et modifié. Le schéma ci-dessous montre la structure ; le tableau qui le suit énumère chaque sous-caractéristique avec un exemple de question qu'un architecte pourrait poser.

Les neuf caractéristiques



L'ISO/IEC 25010:2023 regroupe la qualité des produits en neuf caractéristiques ; par rapport à l'édition 2011, deux ont été renommées et une a été ajoutée.

Caractéristiques, sous-caractéristiques et exemple de question pour une application bancaire mobile

Caractéristique	Sous-caractéristiques	Exemple de question
Adéquation fonctionnelle	Complétude fonctionnelle, exactitude fonctionnelle, pertinence fonctionnelle	L'app calcule-t-elle correctement les remboursements du prêt, et fait-elle tout ce dont le client a besoin pour mener sa tâche à bien ?
Efficacité des performances	Comportement temporel, utilisation des ressources, capacité	En combien de temps le solde s'affiche-t-il sur un téléphone de milieu de gamme en 4G, et combien d'utilisateurs simultanés le backend peut-il servir ?
Compatibilité	Coexistence, interopérabilité	L'app peut-elle échanger des données de paiement avec le système national de paiements rapides, et fonctionner aux côtés d'autres apps sans épuiser les ressources partagées ?
Capacité d'interaction (anciennement utilisabilité)	Identifiabilité de la pertinence, facilité d'apprentissage, opérabilité, protection contre les erreurs de l'utilisateur, engagement de l'utilisateur, inclusivité, assistance à l'utilisateur, autodescriptivité	Un nouvel utilisateur peut-il mettre en place un virement permanent sans aide, et un utilisateur de lecteur d'écran peut-il en faire autant ?
Fiabilité	Absence de fautes, disponibilité, tolérance aux pannes, récupérabilité	Si le partenaire de traitement des cartes est en panne, les clients peuvent-ils toujours consulter leurs soldes et transférer de l'argent entre leurs propres comptes ?
Sécurité	Confidentialité, intégrité, non-répudiation, imputabilité, authenticité, résistance	Pouvons-nous prouver quel client a autorisé un virement, et l'app résiste-t-elle aux attaques par bourrage d'identifiants (credential stuffing) ?
Maintenabilité	Modularité, réutilisabilité, analysabilité, modifiabilité, testabilité	Combien de temps faut-il pour ajouter un nouveau type de compte, et les règles de virement peuvent-elles être testées sans un vrai système bancaire central ?
Flexibilité (anciennement portabilité)	Adaptabilité, scalabilité, facilité d'installation, remplaçabilité	Le backend peut-il passer de 1 à 5 millions de clients, et pourrions-nous remplacer le fournisseur de notifications push en un trimestre ?

Caractéristique	Sous-caractéristiques	Exemple de question
Sûreté (safety)	Contrainte opérationnelle, identification des risques, sécurité intégrée (fail safe), avertissement de danger, intégration sûre	Concerne surtout les systèmes qui peuvent nuire aux personnes, aux biens ou à l'environnement : une pompe à perfusion, un système de contrôle ferroviaire, un robot industriel.

Quelques définitions méritent d'être connues avec précision, car elles diffèrent de l'usage courant. Dans l'ISO/IEC 25010, la **disponibilité** est une sous-caractéristique de la fiabilité : le degré auquel un système est opérationnel et accessible lorsqu'on en a besoin. L'**absence de fautes** est le degré auquel un système exécute ses fonctions sans faute en fonctionnement normal. La **capacité**, sous l'efficacité des performances, est la limite maximale d'un paramètre, comme le nombre d'utilisateurs simultanés ou de transactions par seconde, qui satisfait encore les exigences. La **scalabilité** se range sous la flexibilité, ce qui reflète une vision du passage à l'échelle comme capacité à s'adapter à une charge changeante plutôt que comme une question de vitesse brute. La **résistance**, nouvelle sous la sécurité, est la capacité à maintenir les opérations en cours d'attaque. Les sections 2.4 à 2.7 reviennent en profondeur sur ces qualités ; la section 2.5 montre comment calculer la disponibilité.

Ce qui a changé depuis l'édition 2011

La première édition, l'ISO/IEC 25010:2011, a remplacé l'ancien modèle ISO/IEC 9126 et est devenue la référence pendant une décennie. De nombreux manuels, dossiers d'appel d'offres et gabarits utilisent encore ses huit caractéristiques ; il vaut donc la peine de savoir ce qui a changé.

ISO/IEC 25010 : comparaison des éditions 2011 et 2023

Sujet	Édition 2011	Édition 2023
Nombre de caractéristiques	Huit	Neuf
Utilisabilité	Utilisabilité, y compris l'esthétique de l'interface utilisateur et l'accessibilité	Renommée capacité d'interaction ; l'esthétique remplacée par l'engagement de l'utilisateur ; l'accessibilité scindée en inclusivité et assistance à l'utilisateur ; autodescriptivité ajoutée
Portabilité	Portabilité : adaptabilité, facilité d'installation, remplaçabilité	Renommée flexibilité ; scalabilité ajoutée comme sous-caractéristique
Fiabilité	Maturité, disponibilité, tolérance aux pannes, récupérabilité	Maturité remplacée par absence de fautes
Sécurité	Confidentialité, intégrité, non-répudiation, imputabilité, authenticité	Résistance ajoutée (capacité à résister aux attaques)
Sûreté (safety)	Traitée uniquement dans la qualité en utilisation (absence de risque)	Nouvelle caractéristique de produit, avec cinq sous-caractéristiques
Qualité en utilisation	Fait partie de l'ISO/IEC 25010 (efficacité, efficience, satisfaction, absence de risque, couverture du contexte)	Déplacée dans une norme distincte, l'ISO/IEC 25019:2023 (caractère bénéfique, absence de risque, acceptabilité)

Les changements de nom sont plus que cosmétiques. « Capacité d'interaction » élargit le champ de la facilité d'utilisation à l'ensemble des échanges d'informations entre les utilisateurs et le produit dans différents contextes d'utilisation, et la nouvelle sous-caractéristique d'inclusivité place l'accessibilité pour des personnes d'âges et de capacités différents au centre plutôt qu'en marge. « Flexibilité » regroupe les qualités qui permettent à un produit de faire face aux changements de son environnement : nouvelles plateformes, nouvelles cibles d'installation, et charge plus ou moins forte. L'ajout de la sûreté comme caractéristique de produit reflète la part de logiciel qui pilote désormais des objets physiques, des voitures aux dispositifs médicaux.

Le second changement structurel est que la qualité en utilisation, c'est-à-dire la qualité vécue par les personnes qui utilisent le produit dans un contexte réel, dispose désormais de sa propre norme, l'ISO/IEC 25019:2023. Ses trois caractéristiques sont le caractère bénéfique

(utilisabilité, accessibilité et adéquation en utilisation), l'absence de risque (risques économiques, pour la santé, pour la vie humaine, et risques environnementaux et sociétaux) et l'acceptabilité (expérience, confiance et conformité). Cette séparation clarifie une distinction utile pour les architectes : la qualité du produit est ce que vous construisez et pouvez tester dans le produit ; la qualité en utilisation est le résultat pour des personnes réelles, qui dépend aussi de la formation, du contexte et de la manière dont le produit est déployé.

ATTENTION

Attention à l'édition

Lorsqu'un contrat ou le gabarit d'un client mentionne « ISO 25010 », vérifiez de quelle édition il s'agit. Une exigence qui cite « utilisabilité et portabilité » repose presque certainement sur le modèle de 2011. Ce n'est pas un problème en soi, mais mélanger les deux listes dans un même document crée des lacunes (la sûreté et la résistance sont faciles à oublier) et des doublons (l'accessibilité qui apparaît deux fois). Choisissez une édition et indiquez-la.

SI VOUS DÉBUTEZ

Pas à pas : ce que vous construisez et ce que les gens obtiennent

L'idée qui demande le plus d'attention ici est la différence entre la qualité du produit et la qualité en utilisation. La qualité du produit est une propriété du logiciel, que l'équipe peut construire et tester. La qualité en utilisation est ce que des personnes réelles en retirent effectivement, ce qui dépend aussi de qui elles sont et de la situation dans laquelle elles se trouvent. L'application bancaire mobile illustre la différence.

- Au titre de la protection contre les erreurs de l'utilisateur, qui fait partie de la capacité d'interaction, l'équipe de la banque rédige un scénario : l'app avertit le client avant un paiement vers un bénéficiaire dont le nom ne correspond pas au compte.
- C'est la qualité du produit. Les testeurs envoient des paiements avec des noms concordants et non concordants, et vérifient que l'avertissement apparaît chaque fois qu'il le doit.
- Supposons qu'il passe tous les tests. La banque veut encore savoir autre chose : les clients perdent-ils réellement moins d'argent à cause de paiements erronés ou frauduleux ? C'est l'absence de risque, qui fait partie de la qualité en utilisation et dispose depuis 2023 de sa propre norme, l'ISO/IEC 25019.
- La réponse ne dépend pas que de l'app. Un client pressé peut passer l'avertissement sans le lire ; un fraudeur au téléphone peut le convaincre de l'ignorer ; un client qui a entendu parler de ces arnaques peut s'arrêter et vérifier. Le logiciel est le même à chaque fois ; le résultat, non.
- Les deux se vérifient donc différemment : la qualité du produit par des tests sur le produit, la qualité en utilisation en observant les résultats réels, par exemple le nombre de clients qui signalent avoir payé la mauvaise personne.
- **La leçon :** l'équipe peut concevoir et tester directement la qualité du produit ; la qualité en utilisation ne se révèle qu'à l'usage réel. Une bonne qualité du produit rend une bonne qualité en utilisation plus probable, mais ne peut pas la garantir à elle seule.

D'autres listes que vous rencontrerez

L'ISO/IEC 25010 n'est pas la seule taxonomie, et les praticiens utilisent souvent des listes plus simples ou découpées autrement. FURPS+ est issu de FURPS, une classification développée chez Hewlett-Packard et exposée par Robert Grady et Deborah Caswell dans *Software Metrics: Establishing a Company-Wide Program* (1987). FURPS regroupe les exigences en fonctionnalité, utilisabilité, fiabilité, performances et supportabilité ; l'extension « + », plus tardive, ajoute les contraintes : contraintes de conception, d'implémentation, d'interface et physiques et, dans

certaines versions, règles métier. FURPS+ s'est popularisé grâce au Rational Unified Process et à ses successeurs. Sa force est sa concision ; sa faiblesse est que la « supportabilité » met dans le même sac maintenabilité, testabilité, configurabilité, facilité d'installation et localisation.

Bass, Clements et Kazman, dans la quatrième édition de *Software Architecture in Practice* (2021), ne cherchent pas l'exhaustivité. Ils consacrent un chapitre à chacune des qualités qui orientent le plus souvent les architectures : disponibilité, déployabilité, efficacité énergétique, intégrabilité, modifiabilité, performances, sûreté, sécurité, testabilité et utilisabilité, suivis d'un chapitre sur le travail avec les autres qualités. Pour chacune, ils donnent un scénario général et un catalogue de tactiques. Leur liste est organisée autour de la conception plutôt que de l'évaluation, ce qui la rend particulièrement utile aux architectes.

Mark Richards et Neal Ford, dans *Fundamentals of Software Architecture*, emploient le terme de caractéristiques d'architecture et les répartissent en groupes : opérationnelles (disponibilité, performances, scalabilité), structurelles (modifiabilité, extensibilité, portabilité) et transversales (sécurité, accessibilité, aspects juridiques). Le modèle ouvert Q42 de Gernot Starke, tenu à jour parallèlement au modèle arc42, adopte encore une autre approche : il associe bien plus d'une centaine de qualités nommées à une courte liste de dimensions (comme efficace, flexible, exploitable, fiable, sûr, sécurisé, adapté et utilisable) et fournit des exemples d'exigences pour chacune.

Ces listes divergent dans le détail mais se recoupent largement. Ce désaccord est lui-même instructif : il n'existe pas de décomposition unique et correcte de la « qualité », seulement des décompositions plus ou moins utiles selon l'objectif. Utiliser un modèle normalisé aide lorsqu'il faut communiquer avec les achats, les auditeurs ou d'autres organisations ; une liste plus légère peut mieux servir une équipe produit au quotidien.

Utiliser un modèle comme liste de contrôle sans créer de bureaucratie

Le grand risque d'un modèle de qualité est qu'il devienne un formulaire. Un gabarit d'exigences comportant une rubrique pour chacune des 40 sous-caractéristiques incite à remplir chaque case, et le résultat est un document rempli d'énoncés comme « le système doit être maintenable » que personne ne lit et que personne ne teste. L'effort se répartit uniformément sur des qualités dont l'importance est très inégale, et les deux ou trois qualités qui décideront réellement de l'architecture se perdent parmi les autres.

Une meilleure approche consiste à traiter le modèle comme point de départ d'une conversation. Parcourez les caractéristiques avec les parties prenantes une fois, rapidement, en demandant pour chacune : est-ce important pour notre système et, si oui, à quoi ressemblerait un échec ? Écartez explicitement celles qui ne sont pas pertinentes (écrire « sûreté : sans objet, l'app ne

peut causer aucun dommage physique » est utile, car cela consigne une décision). Transformez celles qui sont pertinentes en un petit nombre de scénarios concrets et mesurables (section 2.3). Classez-les (section 2.8). Puis rangez le modèle jusqu'à la prochaine revue.

Une façon légère d'appliquer le modèle

1. **Parcourir les neuf caractéristiques**
   Lors d'une séance d'une heure, lisez chaque caractéristique à voix haute et demandez aux parties prenantes si elle compte. Notez qui s'en soucie et pourquoi.

2. **Évaluer la pertinence**
   Attribuez à chaque caractéristique une pertinence élevée, moyenne, faible ou « sans objet », et consignez une phrase de justification pour chaque décision.

3. **Approfondir les plus pertinentes**
   Pour les caractéristiques de pertinence élevée seulement, examinez les sous-caractéristiques afin de trouver la préoccupation précise, comme la récupérabilité plutôt que la fiabilité en général.

4. **Rédiger des scénarios**
   Exprimez chaque préoccupation précise sous forme d'un ou deux scénarios mesurables, avec un stimulus, un contexte et une mesure de la réponse.

5. **Hiérarchiser et relier**
   Classez les scénarios et reliez chacun aux décisions de conception qu'il oriente, afin que la liste reste courte et vivante.

EXEMPLE COMMENTÉ

Exemple commenté : appliquer l'ISO/IEC 25010 à une application bancaire mobile

Une banque de taille moyenne remplace son application mobile et son backend. Elle compte 1,2 million d'utilisateurs actifs de l'app ; le soir du jour de paie, environ 25 % d'entre eux se connectent entre 18 h 00 et 20 h 00. L'équipe parcourt les neuf caractéristiques en une heure et consigne le résultat.

Estimation de la charge de pointe	1 200 000 × 25 % = 300 000 connexions en 2 heures = 300 000 ÷ 7200 s ≈ 42 connexions par seconde en moyenne ; prévoir un facteur 3 pour les rafales ≈ 125 par seconde
Adéquation fonctionnelle : élevée	Les intérêts, les frais et les soldes doivent être exacts au centime près. Scénario : chaque solde affiché concorde avec le grand livre central dans les 5 secondes suivant l'enregistrement d'une transaction.
Efficacité des performances : élevée	Scénario : à 125 connexions par seconde, l'écran d'accueil avec les soldes s'affiche en 2 secondes au plus pour 95 % des utilisateurs sur une connexion 4G. La consommation de batterie et de données sur le téléphone compte aussi (utilisation des ressources).
Compatibilité : moyenne	Doit interopérer avec le système national de paiements rapides et avec les API d'open banking qu'impose la réglementation ; doit fonctionner sur les deux dernières versions majeures des deux systèmes d'exploitation mobiles.
Capacité d'interaction : élevée	L'inclusivité et la protection contre les erreurs de l'utilisateur comptent le plus. Scénario : un utilisateur de lecteur d'écran peut effectuer un paiement vers un nouveau bénéficiaire sans l'aide d'une personne voyante ; l'app avertit avant un paiement vers un bénéficiaire dont le nom ne correspond pas au compte.
Fiabilité : élevée	Récupérabilité et tolérance aux pannes. Scénario : si la plateforme de cartes tombe en panne, les soldes et les virements entre les comptes du client continuent de fonctionner ; le service complet est rétabli en 30 minutes au plus.
Sécurité : élevée	Authenticité, non-répudiation et résistance. Scénario : chaque paiement est autorisé par une authentification forte du client et enregistré de sorte que la banque puisse prouver qui l'a approuvé ; 10 000 tentatives de bourrage d'identifiants par minute sont freinées sans bloquer les utilisateurs légitimes.
Maintenabilité : moyenne	Modifiabilité et testabilité. Scénario : un nouveau produit d'épargne peut être lancé en moins de 6 semaines sans modifier le composant de paiement.
Flexibilité : moyenne	Scalabilité. Scénario : le backend peut monter à 3 millions d'utilisateurs par ajout d'instances, sans refonte.

Sûreté : sans objet	L'app ne peut causer aucun dommage physique ; les risques financiers sont couverts par la sécurité et la fiabilité. Consigné explicitement pour que la décision soit visible.
Résultat	Cinq caractéristiques jugées élevées, trois moyennes, une sans objet. Onze scénarios au total, chacun chiffré. Les trois premiers (sécurité, fiabilité, performances au pic du jour de paie) deviennent les objectifs de qualité de la première page du document d'architecture.

Le calcul transforme « beaucoup d'utilisateurs le jour de paie » en un chiffre de conception (environ 125 connexions par seconde en rafale) qui peut être testé.

Les sous-caractéristiques ont joué leur rôle : « sécurité » est devenue authenticité, non-répudiation et résistance, chacune suggérant des mécanismes différents.

Classer la sûreté comme sans objet est une décision, pas un oubli. Si la banque ajoute plus tard une fonctionnalité susceptible de causer des dommages, la trace écrite montre qu'il faut la réexaminer.

Questions fréquentes

Dois-je citer l'ISO/IEC 25010 pour faire de l'architecture correctement ?
Non. Elle est précieuse comme vocabulaire commun et comme liste de contrôle, et indispensable lorsque des contrats ou des régulateurs y font référence. Beaucoup d'excellentes équipes utilisent une liste plus légère. Ce qui compte, c'est que les qualités importantes soient identifiées, rendues mesurables et hiérarchisées.

Où se range la scalabilité : performances ou flexibilité ?
Dans l'ISO/IEC 25010:2023, c'est une sous-caractéristique de la flexibilité, tandis que la capacité se range sous l'efficacité des performances. En pratique, les deux sont étroitement liées, et la section 2.4 les traite ensemble. Ne laissez pas les débats de classification retarder la rédaction du scénario.

L'« observabilité » figure-t-elle dans le modèle ?
Pas sous ce nom. Certaines de ses composantes correspondent à l'analysabilité (maintenabilité) et à l'imputabilité (sécurité). De nombreux praticiens la traitent aujourd'hui comme une qualité à part entière ; la section 2.7 en parle.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Comment l'utilisabilité et la portabilité ont-elles été renommées dans l'ISO/IEC 25010:2023 ?**

L'utilisabilité est devenue la capacité d'interaction et la portabilité est devenue la flexibilité.

▼ **Sous quelle caractéristique l'ISO/IEC 25010:2023 place-t-elle la disponibilité ?**

Sous la fiabilité, aux côtés de l'absence de fautes, de la tolérance aux pannes et de la récupérabilité.

▼ **Pourquoi est-il utile d'écrire « sans objet » en face d'une caractéristique plutôt que de la laisser vide ?**

Cela consigne une décision délibérée avec sa justification, afin que les relecteurs voient qu'elle a été examinée et que la décision puisse être réexaminée si le système change.

▼ **Quel est le principal danger de transformer un modèle de qualité en gabarit d'exigences ?**

Les gens remplissent chaque case d'énoncés vagues, l'effort se répartit uniformément sur des qualités d'importance très inégale, et les rares qualités qui décident réellement de l'architecture se perdent.

▼ **À vous : En parcourant ISO/IEC 25010 pour une app de livreurs, un collègue note les neuf caractéristiques à un niveau élevé « par sécurité ». Qu'est-ce qui ne va pas ? Nommez-en deux que vous noteriez à un niveau élevé et une à un niveau faible, et justifiez.**

Si tout est élevé, rien ne guide la conception et le modèle est devenu de la bureaucratie. La fiabilité est élevée : les livreurs qui captent mal ne doivent pas perdre de courses, regardez donc la récupérabilité. La capacité d'interaction est élevée : les livreurs ont besoin d'étapes courtes et simples, et la sûreté mérite un examen pour la même raison. La compatibilité peut être plus faible si l'app ne dialogue qu'avec les services de l'entreprise. Consignez une phrase de raisonnement pour chacune, puis rédigez des scénarios uniquement pour celles qui sont élevées.

SOURCES

- ISO/IEC 25010:2023 Product quality model (ISO catalogue page): https://www.iso.org/standard/78176.html
- ISO/IEC 25019:2023 Quality-in-use model (ISO catalogue page): https://www.iso.org/standard/78177.html

- Q42: ISO 25010 update 2023 (summary of changes): https://quality.arc42.org/articles/iso-25010-update-2023
- SEI: Software Architecture in Practice, 4th edition: https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/

SECTION 2.3 SUR 2.8 · INTERMÉDIAIRE · 19 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Rédiger des scénarios d'attributs de qualité testables

AVANT DE LIRE

Cette section s'appuie sur :

- 2.1 Exigences fonctionnelles et de qualité
- 2.2 Modèles de qualité

Termes déjà vus qu'elle utilise : Attribut de qualité, Partie prenante, Production

L'IDÉE EN BREF

Dire à un constructeur « faites un toit solide » ne lui donne rien à vérifier. Dire « le toit doit supporter 50 centimètres de neige mouillée pendant une tempête d'hiver sans fuir ni fléchir de plus de 2 centimètres », si. Cela précise ce qui se passe (la neige), les conditions (une tempête), la partie concernée (le toit), ce qui doit se passer (aucune fuite) et un chiffre mesurable (2 centimètres). Un scénario d'attribut de qualité fait la même chose pour le logiciel. Il transforme un souhait comme « le site web doit être rapide » en un énoncé assez précis pour concevoir et tester.

CE QU'IL FAUT SAVOIR

- Un scénario d'attribut de qualité comporte six parties : la source du stimulus, le stimulus, l'artefact, l'environnement, la réponse et la mesure de la réponse.
- Les scénarios généraux sont des modèles indépendants de tout système, propres à une qualité ; les scénarios concrets renseignent des valeurs précises pour un système donné et servent d'exigences.
- Un scénario sans mesure de la réponse, ou qui utilise une moyenne au lieu d'un percentile pour un temps, ne peut pas être testé de manière fiable.
- Les erreurs courantes consistent à mêler plusieurs stimuli dans un même scénario, à omettre l'environnement et à écrire une solution, comme un choix technologique, à la place du besoin.
- Les scénarios se traduisent directement en tests d'acceptation, puis en fonctions d'aptitude qui vérifient la qualité en continu.

Pourquoi des scénarios

La section 2.2 vous a donné le vocabulaire des qualités ; cette section vous en donne la grammaire. Le nom d'une qualité à lui seul, comme les performances ou la sécurité, ne permet ni de concevoir ni de tester. Même un chiffre seul ne suffit pas : « 2 secondes » ne veut rien dire tant que l'on ne sait pas pour quelle opération, sous quelle charge, mesuré où et pour quelle proportion des requêtes. Le scénario d'attribut de qualité, mis au point au Software Engineering Institute (SEI) et exposé par Len Bass, Paul Clements et Rick Kazman dans *Software Architecture in Practice*, résout ce problème en imposant à chaque exigence de qualité la même forme en six parties.

L'idée est simple. Chaque qualité se manifeste lorsqu'il arrive quelque chose à un système : un afflux d'utilisateurs, une panne, une demande de changement, une attaque. Un scénario nomme cet événement, les circonstances, la partie du système touchée, ce qui doit se passer et la manière dont vous saurez que cela s'est passé suffisamment bien. Il est court (en général une à trois phrases), concret et testable. Et comme toutes les qualités utilisent la même forme, des scénarios de performances, de sécurité et de modifiabilité peuvent figurer côte à côte dans une même liste hiérarchisée.

Les six parties

Les six parties sont la source du stimulus, le stimulus, l'artefact, l'environnement, la réponse et la mesure de la réponse. Apprenez-les dans cet ordre, car ensemble elles racontent une courte histoire : *quelqu'un ou quelque chose* fait *quelque chose* à *une partie du système* dans *certaines conditions* ; le système *fait quelque chose*, et nous *le mesurons* ainsi. Le stimulus et la mesure de la réponse sont les deux parties que l'on oublie le plus souvent, et ce sont les deux qui comptent le plus.

Anatomie d'un scénario



Exemple : 500 clients par seconde (source, stimulus) arrivent au paiement (artefact) le Black Friday (environnement) ; commandes confirmées (réponse), p99 sous 1,5 s, aucune commande perdue (mesure de la réponse).

Une source produit un stimulus qui atteint un artefact dans un environnement ; la réponse de l'artefact est évaluée par une mesure de la réponse.

Les six parties en détail

Composant	Signification	Exemples
Source du stimulus	La personne, le système ou l'événement qui génère le stimulus.	Des clients sur téléphone mobile ; un traitement par lots interne ; un développeur ; un attaquant ; un disque défaillant.
Stimulus	L'événement auquel le système doit répondre.	Une rafale de requêtes ; la panne d'un serveur ; une demande de changement ; une tentative de connexion avec des identifiants volés.
Artefact	La partie du système qui reçoit le stimulus : le système entier ou un composant précis.	Le service de paiement ; la base de données des commandes ; le module de tarification ; le pipeline de déploiement.
Environnement	Les conditions dans lesquelles le stimulus survient.	Fonctionnement normal ; pic du Black Friday ; un centre de données hors service ; pendant un déploiement ; en développement.
Réponse	Ce que fait le système (ou les personnes qui l'entourent) en conséquence.	Traite la commande ; bascule sur la réplique ; la modification est réalisée, testée et déployée ; la tentative est bloquée et journalisée.
Mesure de la réponse	La manière dont la réponse est mesurée, pour que le scénario puisse être testé.	p99 inférieur à 800 ms ; service rétabli en moins de 60 s sans aucune commande perdue ; moins de 2 jours-personne ; 100 % des tentatives bloquées.

L'environnement mérite une attention particulière, car un même stimulus peut exiger des réponses très différentes selon les conditions. Un serveur en panne pendant le fonctionnement normal peut exiger un failover (basculement) automatique en moins d'une minute ; la même panne pendant une fenêtre de maintenance planifiée peut être acceptable avec une reprise manuelle. Une demande de changement au moment de la conception (avant la mise en production) coûte bien moins cher que le même changement une fois que des données se sont accumulées en production. Écrire l'environnement explicitement évite que l'on compare des scénarios qui portent en réalité sur des situations différentes.

L'artefact compte pour une raison similaire. « Le système répond en 200 ms » cache quelle partie est responsable. Nommer l'artefact (le service de recherche, l'API gateway, le client mobile) indique à l'équipe où le travail de conception et la mesure doivent avoir lieu. Pour certaines qualités, comme la modifiabilité et la déployabilité, l'artefact peut être du code, de la configuration ou le pipeline de livraison plutôt qu'un composant en cours d'exécution.

Scénarios généraux et scénarios concrets

Bass, Clements et Kazman en distinguent deux types. Un scénario général est indépendant de tout système : pour un attribut de qualité donné, il énumère les valeurs typiques que peut prendre chaque partie. Pour la disponibilité, par exemple, les sources peuvent être internes ou externes, des personnes, du matériel, du logiciel ou de l'infrastructure ; les stimuli peuvent être une panne, une omission, un timing incorrect ou une réponse incorrecte ; les réponses peuvent consister à détecter, enregistrer, rétablir, masquer ou dégrader ; et les mesures peuvent être le temps de détection, le temps de réparation, le pourcentage de disponibilité ou la proportion de requêtes traitées.

Un scénario concret choisit des valeurs précises pour un système particulier : « la base de données principale des commandes tombe en panne pendant le fonctionnement normal ; la réplique prend le relais et les écritures reprennent en moins de 60 secondes sans perte d'aucune commande validée ». Les scénarios généraux sont des modèles et des listes de contrôle ; les scénarios concrets sont des exigences. Vous utilisez le scénario général pour vous assurer d'avoir envisagé les bonnes sources, les bons stimuli et les bonnes mesures, puis vous rédigez la poignée de scénarios concrets qui comptent pour votre système.

Des énoncés vagues aux scénarios testables

La plupart des exigences de qualité commencent leur vie sous forme d'adjectifs. Tout l'art consiste à poser les questions qui transforment chaque adjectif en scénario : quelle opération ? déclenchée par qui ? à quelle fréquence ou en quelle quantité ? dans quelles conditions ? que doit-il se passer ? quel chiffre nous dira que cela a fonctionné ? Le tableau présente des paires

« avant » et « après » typiques. Remarquez que les versions « après » révèlent parfois que la partie prenante voulait dire plusieurs choses différentes, qui deviennent alors plusieurs scénarios.

Avant et après

Énoncé vague	Scénario testable
Le site doit être rapide.	Pendant le pic des soirs de semaine, à 300 recherches par seconde, la recherche de produits renvoie des résultats en moins de 400 ms pour 95 % des requêtes et en moins de 1 s pour 99 %, mesuré au niveau de l'équilibreur de charge (load balancer).
Le système doit être hautement disponible.	Si l'un des trois serveurs d'application tombe en panne pendant le fonctionnement normal, le service de réservation continue d'accepter des réservations ; aucune réservation confirmée n'est perdue et les utilisateurs voient des erreurs pendant 30 secondes au plus.
Le système doit être sécurisé.	Lorsqu'un attaquant soumet 10 000 tentatives de connexion par minute avec des identifiants divulgués, le système bloque la source après 20 échecs, alerte l'équipe de sécurité en moins de 5 minutes, et les utilisateurs légitimes connectés depuis d'autres adresses peuvent toujours se connecter.
Le code doit être facile à modifier.	Lorsque le métier demande une nouvelle option de livraison, un développeur qui connaît le code peut l'implémenter, la tester et la déployer en 3 jours au plus, en ne modifiant que le module de livraison et sa configuration.
L'application doit être conviviale.	Un nouvel utilisateur sans formation peut s'inscrire et passer une commande en moins de 4 minutes ; 90 % des participants aux tests réussissent du premier coup.
Les déploiements doivent être faciles.	Un développeur peut mettre en production une modification du service de commande en moins de 30 minutes après le merge, sans interruption de service, et la retirer par un retour arrière (rollback) en moins de 5 minutes si le taux d'erreur augmente.

SI VOUS DÉBUTEZ

Si vous débutez : d'où viennent les chiffres ?

On hésite souvent à écrire un chiffre parce qu'on ne connaît pas le « bon ». Partez des données disponibles : les journaux de trafic actuels, la supervision de l'ancien système, le comportement des concurrents, les objectifs métier (« nous prévoyons de doubler les commandes d'ici l'an prochain ») et les études sur la patience des utilisateurs. Proposez ensuite un chiffre et laissez les parties prenantes réagir. « 400 millisecondes pour 95 % des recherches, est-ce acceptable ? » suscite une bien meilleure conversation que « à quelle vitesse cela doit-il aller ? ». Un chiffre convenu aujourd'hui peut être révisé demain ; une exigence sans chiffre ne peut même pas être discutée.

Des exemples de scénarios pour six qualités

Les exemples ci-dessous portent tous sur une même boutique en ligne, pour que vous puissiez voir comment des qualités différentes se présentent dans le même format. Chacun est écrit de façon compacte avec ses six parties étiquetées ; en pratique, les équipes les écrivent souvent en une ou deux phrases.

Scénarios concrets pour une boutique en ligne

Qualité	Scénario (source, stimulus, artefact, environnement, réponse, mesure)
Performances	Source : clients connectés. Stimulus : 2000 commandes soumises par minute. Artefact : service de paiement. Environnement : pic du vendredi soir. Réponse : commandes validées, enregistrées et confirmées. Mesure : p95 ≤ 1,5 s, p99 ≤ 3 s, aucune commande perdue.
Disponibilité	Source : matériel interne. Stimulus : le nœud principal de la base de données tombe en panne. Artefact : base de données des commandes. Environnement : fonctionnement normal. Réponse : réplique promue automatiquement, requêtes en cours relancées par de nouvelles tentatives. Mesure : écritures reprises en moins de 60 s ; aucune commande validée perdue (RPO = 0).
Modifiabilité	Source : product owner. Stimulus : demande d'ajout d'un nouveau moyen de paiement. Artefact : composant des moyens de paiement. Environnement : conception, après la première mise en production. Réponse : modification réalisée, testée et déployée. Mesure : ≤ 5 jours-personne ; aucune modification hors du composant des moyens de paiement ; aucune régression sur les moyens existants.
Sécurité	Source : un client connecté agissant de manière malveillante. Stimulus : tente de lire les commandes d'un autre client en modifiant un identifiant dans l'URL. Artefact : API des commandes. Environnement : fonctionnement normal. Réponse : requête refusée et journalisée avec l'identité de l'appelant. Mesure : 100 % de ces requêtes rejetées ; alerte levée en moins de 1 minute après 5 tentatives.
Utilisabilité	Source : client novice. Stimulus : veut changer d'adresse de livraison après sa commande. Artefact : frontends web et mobile. Environnement : exécution, aucune aide disponible. Réponse : effectue le changement sans aide. Mesure : 90 % des participants aux tests d'utilisabilité réussissent en moins de 2 minutes.
Déployabilité	Source : développeur. Stimulus : une modification fusionnée dans le service catalogue. Artefact : pipeline de déploiement et service catalogue. Environnement : heures ouvrées, charge normale. Réponse : construite, testée et déployée progressivement. Mesure : en production en moins de 45 minutes ; aucune interruption de service ; retour arrière automatique en moins de 5 minutes si le taux d'erreur dépasse 1 %.

Deux caractéristiques de ces exemples méritent d'être relevées. D'abord, les scénarios ne portent pas uniquement sur le logiciel en cours d'exécution. Les scénarios de modifiabilité et de déployabilité ont pour sources des développeurs et des product owners, et leurs mesures portent sur l'effort, le temps écoulé et l'étendue du changement. Ensuite, les bonnes mesures combinent souvent un chiffre de rapidité et un chiffre d'exactitude : « en moins de 60 secondes » et « aucune commande validée perdue ». Un failover rapide qui perd des données n'est pas ce que voulait le métier.

Erreurs courantes

À relire des centaines de scénarios rédigés par des équipes, on retrouve sans cesse les mêmes erreurs. Chacune rend un scénario intestable ou trompeur.

Les erreurs et comment les corriger

Erreur	Exemple	Correction
Pas de mesure de la réponse	« Sous la charge de pointe, le paiement répond rapidement. »	Ajoutez un chiffre et une statistique : « p95 ≤ 1,5 s, p99 ≤ 3 s ».
Une moyenne au lieu d'un percentile	« Temps de réponse moyen inférieur à 300 ms. »	Les moyennes masquent les valeurs lentes extrêmes ; spécifiez des percentiles (section 2.4).
Plusieurs stimuli dans un même scénario	« Quand la charge double, qu'un serveur tombe en panne ou qu'un déploiement est en cours, le système reste rapide. »	Divisez-le en trois scénarios, chacun avec son propre environnement et sa propre mesure.
Une solution déguisée en exigence	« Le système utilise Redis pour mettre en cache les pages produits. »	Énoncez le besoin (« les pages produits se chargent en moins de 300 ms pour 95 % des affichages à 1000 affichages par seconde ») et laissez le cache comme option de conception.
Environnement manquant	« Le service se rétablit en moins de 5 minutes. »	Précisez après quoi et quand : « après la panne d'une seule zone pendant le fonctionnement normal ».
Des mots non mesurables	« Gracieusement », « de façon transparente », « de manière robuste », « convivial ».	Remplacez chacun par un comportement observable et un chiffre.
Des absolus impossibles	« Le système ne tombe jamais en panne. » « 100 % des attaques bloquées. »	Utilisez des objectifs réalistes et précisez ce qui se passe quand ils ne sont pas atteints.

ATTENTION

Des solutions déguisées en exigences

C’est l’erreur la plus dommageable, car elle ressemble à un progrès. « Utiliser Kubernetes », « utiliser des microservices », « utiliser une file de messages » sont des décisions de conception. Les écrire comme des exigences fait sauter l’étape où l’équipe se demande si ce sont les meilleurs moyens d’atteindre la qualité visée, et cela supprime les éléments nécessaires pour revenir sur ce choix plus tard. Si une partie prenante tient à une technologie, demandez-lui quel problème elle résout ; la réponse est le vrai scénario. Si la technologie est réellement imposée par une politique, enregistrez-la comme une contrainte en indiquant son origine.

GUIDE DE DÉCISION

Guide de décision : ce scénario est-il prêt ?

Soumettez chaque brouillon de scénario à ces questions avant qu'il n'entre dans le document d'architecture ou dans le backlog.

Q1 **Décrit-il un seul stimulus avec une source nommée ?**

- **Oui** Continuez.
- **Non** Divisez-le, ou nommez qui ou quoi provoque l'événement.

Q2 **Indique-t-il l'environnement (normal, pic, dégradé, pendant un déploiement, au moment de la conception) ?**

- **Oui** Continuez.
- **Non** Ajoutez les conditions ; sinon, la mesure ne peut pas être interprétée.

Q3 **Nomme-t-il l'artefact, c'est-à-dire la partie du système responsable ?**

- **Oui** Continuez.
- **Non** Nommez le composant, l'interface ou le pipeline, pour que l'équipe sache où concevoir et où mesurer.

Q4 **La réponse est-elle un comportement observable plutôt qu'un choix technologique ?**

- **Oui** Continuez.
- **Non** Réécrivez le besoin et déplacez la technologie dans une décision de conception ou une contrainte.

Q5 **La mesure de la réponse contient-elle un chiffre, une unité et, pour un temps, un percentile ?**

- **Oui** Le scénario est testable : ajoutez une priorité et un responsable.
- **Non** Ajoutez la mesure ; sans elle, le scénario n'est qu'un souhait.

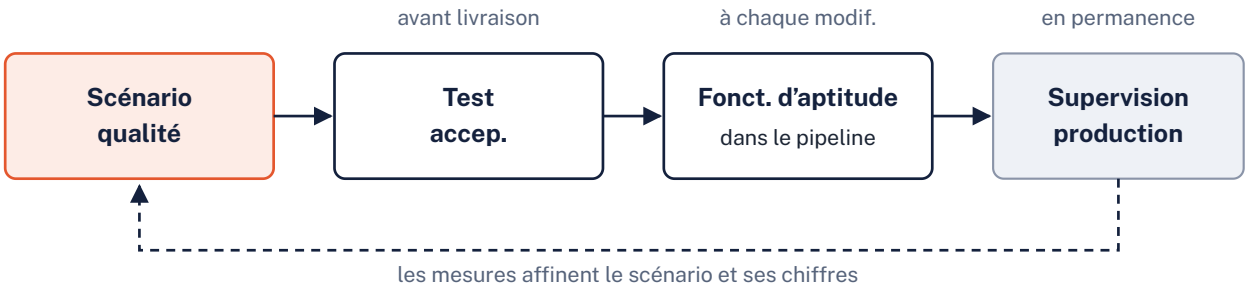
Des scénarios aux tests d'acceptation et aux fonctions d'aptitude

Un scénario bien rédigé constitue déjà l'essentiel d'un test. La source et le stimulus vous disent quelle charge ou quel événement générer, l'environnement vous dit comment préparer le test, l'artefact vous dit quoi observer et la mesure de la réponse fournit la condition de réussite. Un scénario de performances devient un test de charge ; un scénario de disponibilité devient un

exercice de failover ou une expérience de chaos ; un scénario de modifiabilité devient l'examen de la quantité de code touchée par une modification type et du temps qu'elle a pris ; un scénario de sécurité devient un cas de test d'intrusion.

L'étape suivante consiste à continuer de vérifier. Une fonction d'aptitude (fitness function), terme popularisé par Neal Ford, Rebecca Parsons et Patrick Kua dans *Building Evolutionary Architectures*, est toute vérification automatisée ou reproductible de la mesure dans laquelle le système atteint un objectif architectural. Exécutée dans le pipeline de livraison ou sur la télémétrie de production, elle fait passer un scénario du statut de test d'acceptation ponctuel à celui de garde-fou permanent : le build échoue, ou une alerte se déclenche, quand la qualité dérive. Le chapitre 11 traite des fonctions d'aptitude en profondeur dans Fonctions d'aptitude et architecture évolutive.

La vie d'un scénario



Un scénario donne lieu à un test d'acceptation, puis à une fonction d'aptitude exécutée à chaque modification, et les constats faits en production reviennent affiner le scénario.

```
# A performance scenario expressed as an automated check (illustrative)
# Scenario: at 300 searches/s, p95 ⩽ 400 ms and p99 ⩽ 1000 ms.
import statistics

def check_search_latency(latencies_ms, p95_target=400, p99_target=1000):
    cuts = statistics.quantiles(latencies_ms, n=100)  # 99 cut points
    p95, p99 = cuts[94], cuts[98]
    assert p95 ⩽ p95_target, f"p95 {p95:.0f} ms exceeds {p95_target} ms"
    assert p99 ⩽ p99_target, f"p99 {p99:.0f} ms exceeds {p99_target} ms"
    return p95, p99

# latencies_ms comes from a load test that replays 300 searches per second
# for ten minutes against a production-like environment.
```

Un scénario de performances transformé en une vérification automatisée simple de la latence p95 et p99.

Comment les professionnels utilisent les scénarios

En pratique, les équipes rédigent rarement de longs tableaux de scénarios en six parties pour tout. Elles en écrivent un petit nombre, souvent de dix à trente pour un système d'une certaine taille, pour les qualités qui sont architecturalement significatives, et elles les conservent là où tout le monde peut les voir : dans le document d'architecture (la section 10 d'arc42 demande exactement cela), dans le backlog sous forme de critères d'acceptation, ou sur un tableau partagé des qualités. Certaines équipes utilisent une forme plus courte ; le projet Q42 de Gernot Starke, par exemple, recommande une structure compacte : contexte, déclencheur et critères d'acceptation. La forme compte moins que la discipline : chaque qualité importante a un stimulus, un contexte et un chiffre.

Les scénarios alimentent aussi les méthodes d'évaluation. Le Quality Attribute Workshop et l'Architecture Tradeoff Analysis Method (ATAM), tous deux issus du SEI, recueillent et hiérarchisent des scénarios avec les parties prenantes, puis confrontent l'architecture aux plus importants d'entre eux. La section 2.8 montre comment les scénarios sont recueillis et classés dans un arbre d'utilité.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Nommez dans l'ordre les six parties d'un scénario d'attribut de qualité.**

Source du stimulus, stimulus, artefact, environnement, réponse et mesure de la réponse.

▼ **Qu'est-ce qui ne va pas dans « le système utilise un CDN pour que les pages se chargent vite » ?**

Cet énoncé formule une solution, pas un besoin, et n'a pas de mesure. L'exigence devrait décrire la charge, les conditions et le temps de réponse visé ; le CDN est une conception possible pour y répondre.

▼ **Qui est généralement la source du stimulus dans un scénario de modifiabilité ?**

Une personne qui veut un changement, comme un product owner, un développeur ou un régulateur ; le stimulus est la demande de changement.

▼ **En quoi un scénario diffère-t-il d'une fonction d'aptitude ?**

Un scénario est l'énoncé d'une exigence ; une fonction d'aptitude est une vérification automatisée ou reproductible, exécutée en continu, qui mesure si le système y satisfait.

▼ **À vous : Transformez « le site de billetterie doit rester rapide le jour de la mise en vente » en scénario d'attribut de qualité en six parties, en inventant des chiffres raisonnables. Nommez chaque partie. Donnez ensuite une raison pour laquelle la phrase d'origine ne pourrait jamais être testée.**

Source : les navigateurs des fans. Stimulus : 2500 personnes arrivent chaque seconde. Artefact : le parcours de sélection des places et d'achat. Environnement : pic d'ouverture des ventes, toutes les zones saines. Réponse : chaque requête de page est servie, ou le fan est placé dans une salle d'attente. Mesure de la réponse : temps de réponse des pages p99 inférieur à 2 secondes. L'original ne pouvait pas être testé parce que « rapide » n'a pas de chiffre et qu'il ne cite ni charge ni conditions : personne ne pouvait donc dire s'il était satisfait.

SOURCES

- SEI: Software Architecture in Practice, 4th edition: https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/
- arc42 section 10: Quality requirements and quality scenarios: https://docs.arc42.org/section-10/
- Q42: How to specify quality requirements: https://quality.arc42.org/articles/specify-quality-requirements

SECTION 2.4 SUR 2.8 · INTERMÉDIAIRE · 27 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Performances et scalabilité : latence, débit, percentiles et charge

AVANT DE LIRE

Cette section s'appuie sur :

- 0.2 Données et algorithmes
- 0.4 Comment les ordinateurs communiquent
- 0.7 Serveurs et cloud

Termes déjà vus qu'elle utilise : Serveur, Cache, Base de données

L'IDÉE EN BREF

Imaginez un supermarché. Le temps que vous passez à la caisse, c'est la latence ; le nombre de clients que les caisses servent en une heure, c'est le débit. Votre temps à la caisse se compose en partie du passage des articles (le travail lui-même) et en partie de la file devant vous. Quand les caisses ne sont occupées qu'à moitié, vous n'attendez presque pas ; quand elles sont presque toujours occupées, les files s'allongent très vite, même si chaque passage d'article prend le même temps. Ouvrir plus de caisses aide, jusqu'à ce que tout le monde doive partager un seul responsable pour les remboursements. Les performances et la scalabilité portent exactement sur ces effets dans le logiciel.

CE QU'IL FAUT SAVOIR

- La latence est le temps que prend une unité de travail ; le débit est la quantité de travail accomplie par unité de temps. Une exigence de performances devrait énoncer les deux, avec la charge à laquelle ils s'appliquent.
- Le temps de réponse est le temps de service plus le temps d'attente (plus le temps réseau) ; sous charge, c'est généralement le temps d'attente qui domine.
- Spécifiez et surveillez des percentiles comme p95 et p99, pas des moyennes ; dans les systèmes en fan-out, une réponse lente rare sur un serveur devient courante pour les utilisateurs (Dean et Barroso, 2013).
- La loi de Little, $L = \lambda \times W$, relie le débit de requêtes, le temps de réponse et la concurrence ; elle sert de base au dimensionnement des pools et des limites.
- Dans le modèle de file le plus simple, M/M/1, le temps de réponse moyen vaut $S \div (1 - \rho)$; il double donc entre 80 % et 90 % de taux d'utilisation ; les autres files diffèrent dans le détail mais présentent la même falaise : gardez donc les ressources critiques bien en

dessous de la saturation.

- La loi d'Amdahl limite l'accélération à 1 ÷ s pour une fraction séquentielle s ; la loi universelle de scalabilité ajoute des coûts de cohérence qui font culminer puis décroître la capacité.

Deux questions différentes : à quelle vitesse, et combien

Les performances sont la qualité que l'on cite en premier et que l'on spécifie le plus mal. Elles recouvrent deux questions différentes. La latence demande combien de temps prend une unité de travail : combien de millisecondes s'écoulent entre le moment où l'on touche « Payer » et celui où l'on voit « Paiement accepté ». Le débit demande quelle quantité de travail le système accomplit par unité de temps : 1200 paiements par seconde, 40 000 commandes par heure. Les deux sont liés mais pas interchangeables. Un autocar qui transporte 50 personnes de Londres à Oxford en 100 minutes a une moins bonne latence et un bien meilleur débit qu'une voiture de sport qui transporte deux personnes en 60 minutes.

Dans le logiciel, la même tension apparaît partout. Regrouper les écritures en base de données par lots améliore le débit, mais fait attendre chaque écriture individuelle jusqu'à son lot. Ajouter un cache améliore les deux pour les lectures répétées, mais ne fait rien pour la première. Une bonne exigence de performances précise donc la question qu'elle traite, et énonce généralement les deux : « à 1200 paiements par seconde (débit), 99 % aboutissent en moins de 800 ms (latence) ». L'ISO/IEC 25010 les range sous l'efficacité des performances, en tant que comportement temporel, utilisation des ressources et capacité (section 2.2).

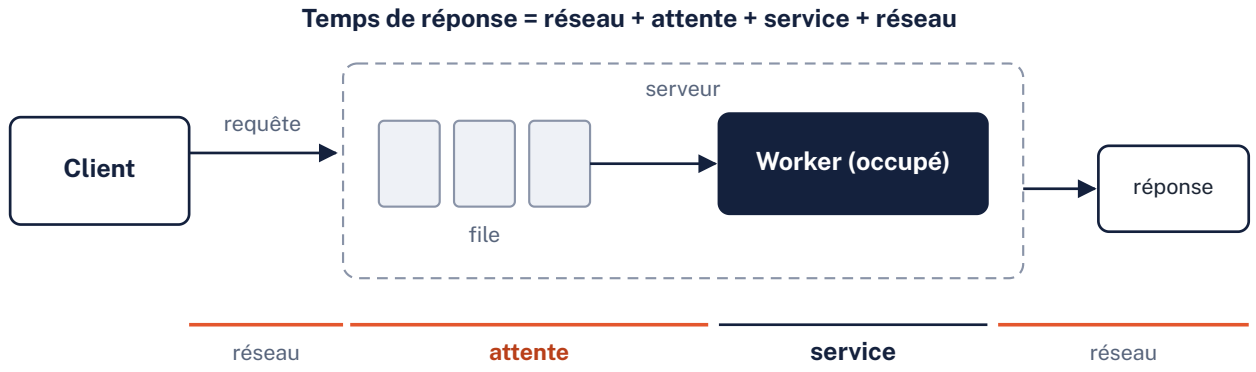
Temps de réponse = temps de service + temps d'attente

Quand vous mesurez la latence du côté du client, vous obtenez le temps de réponse : tout ce qui s'est passé entre l'envoi de la requête et la réception de la réponse. Il est utile de le décomposer. Le temps de service est le temps qu'une ressource passe réellement à travailler sur votre requête : l'analyser, exécuter la requête en base, produire le résultat. Le temps d'attente est le temps que votre requête passe dans une file parce que la ressource est occupée par celle de quelqu'un d'autre. Le temps de transfert réseau s'ajoute par-dessus, dans les deux sens.

La distinction compte, car les deux parties réagissent à des remèdes différents. Le temps de service diminue quand vous rendez le travail lui-même moins coûteux : un meilleur index, une charge utile plus petite, un algorithme plus rapide. Le temps d'attente diminue quand vous ajoutez de la capacité ou réduisez la charge, et il est souvent bien plus long que le temps de

service quand un système est chargé. Bien des enquêtes sur une « base de données lente » se terminent par la découverte que chaque requête prend 5 ms mais attend 300 ms une connexion libre dans le pool.

Où passe le temps



Le temps de réponse que voit un client comprend le transfert réseau, l'attente dans une file qu'un worker occupé se libère, et le temps de service lui-même.

Percentiles : pourquoi les moyennes trompent

Les temps de réponse ne se résument jamais à un seul nombre ; ils forment une distribution, et celle-ci est presque toujours asymétrique. La plupart des requêtes sont rapides, et une minorité est lente parce qu'elle tombe sur un cache froid, une pause du ramasse-miettes, un verrou, une nouvelle tentative ou un voisin surchargé. La moyenne arithmétique mélange les deux groupes en un chiffre qui ne décrit personne. Les professionnels utilisent plutôt des percentiles. Le 50e percentile (p50, la médiane) est la valeur que bat la moitié des requêtes ; p95 est la valeur que battent 95 % des requêtes ; p99 et p99.9 décrivent les 1 % et 0,1 % les plus lents.

L'extrémité lente de la distribution s'appelle la queue (tail), et elle compte plus qu'il n'y paraît. Les utilisateurs situés dans la queue sont souvent vos meilleurs clients : ceux qui ont les plus gros paniers, les historiques les plus longs et le plus de données. Un utilisateur qui charge 40 ressources pour afficher une page subit la plus lente de ces 40, pas la moyenne. Et la probabilité qu'une page de 40 ressources tombe sur au moins une requête au niveau du p99 est de $1 - 0{,}99^{40} \approx 33$ % : un événement « rare » à 1 % touche un tiers des affichages de pages.

EXEMPLE COMMENTÉ

Exemple commenté : la moyenne qui cache le problème

Une API de recherche de produits enregistre 1000 temps de réponse en une minute : 900 à 80 ms, 80 à 150 ms, 15 à 1200 ms (un cache froid) et 5 à 4000 ms (un délai d'expiration et une nouvelle tentative).

Somme de tous les temps	900 × 80 + 80 × 150 + 15 × 1200 + 5 × 4000 = 72 000 + 12 000 + 18 000 + 20 000 = 122 000 ms
Moyenne	122 000 ÷ 1000 = 122 ms. Semble satisfaisant au regard d'un objectif de 200 ms.
p50 (médiane)	La 500e requête la plus rapide : 80 ms
p95	La 950e requête la plus rapide se trouve dans le groupe à 150 ms (requêtes 901 à 980) : 150 ms
p99	La 990e se trouve dans le groupe à 1200 ms (requêtes 981 à 995) : 1200 ms
p99.9	La 999e se trouve dans le groupe à 4000 ms (996 à 1000) : 4000 ms
Qui est touché	20 requêtes sur 1000 (2 %) ont pris plus d'une seconde. À 300 recherches par seconde, cela fait 6 utilisateurs par seconde, soit environ 21 600 par heure.

La moyenne de 122 ms respecte un objectif de 200 ms alors que 2 % des utilisateurs attendent plus d'une seconde. Un scénario rédigé sous la forme « moyenne inférieure à 200 ms » validerait ce système.

Rédiger l'exigence sous la forme « p95 ≤ 200 ms et p99 ≤ 500 ms » le fait échouer, à juste titre, et oriente l'équipe vers les chemins du cache froid et des nouvelles tentatives.

Les percentiles ne se moyennent pas. Le p99 sur dix serveurs n'est pas la moyenne de leurs dix valeurs de p99 ; il doit être calculé à partir des données combinées, ou estimé à partir d'histogrammes fusionnés.

```
# Why the mean misleads: 1,000 response times from one minute of traffic
import statistics

times_ms = [80] * 900 + [150] * 80 + [1200] * 15 + [4000] * 5

mean = statistics.mean(times_ms)
cuts = statistics.quantiles(times_ms, n=1000, method="inclusive")
p50, p95, p99, p999 = cuts[499], cuts[949], cuts[989], cuts[998]
print(f"mean {mean:.0f} ms, p50 {p50:.0f}, p95 {p95:.0f}, p99 {p99:.0f}, p99.9
{p999:.0f}")
# mean 122 ms, p50 80, p95 150, p99 1200, p99.9 4000
```

Le même calcul en quelques lignes de Python : la moyenne paraît correcte, tandis que p99 et p99.9 révèlent le problème.

La latence de queue à grande échelle

La queue devient l'événement principal dans les grands systèmes qui recourent au fan-out (diffusion en éventail), où une requête d'utilisateur est divisée en de nombreuses requêtes parallèles vers d'autres serveurs et où la réponse doit les attendre toutes. Un moteur de recherche, un fil d'actualité social ou une page de recommandations peut consulter des dizaines ou des centaines de shards back-end pour chaque requête. Jeffrey Dean et Luiz André Barroso, de Google, ont analysé ce phénomène dans *The Tail at Scale* (Communications of the ACM, 2013).

Leur illustration est simple. Supposons que chaque serveur réponde généralement en 10 ms, mais mette 1 seconde pour une requête sur 100. Une requête d'utilisateur qui attend un seul serveur est lente 1 % du temps. Une requête d'utilisateur qui attend 100 serveurs de ce type est lente dès que l'un d'eux est lent : $1 - 0{,}99^{100} \approx 63$ % du temps. Un événement rare sur un serveur devient le cas courant pour l'utilisateur. Dean et Barroso rapportent des mesures d'un service Google dans lequel la latence au 99e percentile pour une seule requête feuille était de 10 ms, mais le 99e percentile de l'attente de toutes les requêtes feuilles était de 140 ms.

Comme on ne peut pas éliminer toutes les sources de variabilité (matériel partagé, tâches d'arrière-plan, ramasse-miettes, files réseau), ils plaident pour des conceptions tolérantes à la queue, de même que les conceptions tolérantes aux pannes acceptent que le matériel tombe en panne. Leurs techniques comprennent les requêtes de couverture (hedged requests : envoyer une deuxième copie à une autre réplique si la première n'a pas répondu dans, disons, le temps du 95e percentile, et utiliser celle qui revient en premier), les requêtes liées (tied requests) qui s'annulent l'une l'autre, des partitions plus fines qui peuvent être rééquilibrées, l'exclusion temporaire des machines lentes, et le renvoi d'un résultat suffisamment bon quand

la plupart des shards, mais pas tous, ont répondu. Dans un banc d'essai qu'ils décrivent, déclencher la couverture après 10 ms a fait passer la latence au 99,9e percentile de 1800 ms à 74 ms, en n'envoyant qu'environ 2 % de requêtes en plus.

La loi de Little : combien de requêtes sont en cours

La loi de Little, démontrée par John Little en 1961, est la formule la plus utile en planification de la capacité. Pour tout système stable, le nombre moyen d'éléments qu'il contient (L) est égal au taux d'arrivée moyen (λ, lambda) multiplié par le temps moyen que chaque élément passe à l'intérieur (W) : **L = λ × W**. Elle est vraie quels que soient le profil des arrivées ou l'ordre de service, à condition que le système soit stable (en moyenne, autant d'éléments sortent qu'il en arrive) et que vous utilisiez des unités cohérentes.

Pour un logiciel, L est le nombre de requêtes en cours simultanément (la concurrence), λ est le débit de requêtes et W est le temps de réponse. Si un service reçoit 200 requêtes par seconde et que chacune y passe 0,25 seconde, alors en moyenne 200 × 0,25 = 50 requêtes sont en cours à tout instant. Ce nombre détermine les pools de threads, les pools de connexions, la mémoire et le nombre de licences. La loi fonctionne aussi à rebours : si vous pouvez traiter au plus 50 requêtes à la fois et que chacune prend 0,25 s, vous ne pouvez pas soutenir plus de 50 ÷ 0,25 = 200 requêtes par seconde.

EXEMPLE COMMENTÉ

Exemple commenté : dimensionner un pool de connexions à une base de données avec la loi de Little

Le service des trajets d'une application de transport avec chauffeur traite 1800 requêtes par seconde au pic du soir. Chaque requête fait des appels à la base de données qui occupent une connexion pendant 40 ms au total. Chacune des 6 instances du service dispose d'un pool de 10 connexions. Est-ce suffisant ?

Taux d'arrivée λ	1800 requêtes par seconde
Temps d'occupation d'une connexion W	40 ms = 0,040 s
Connexions utilisées L = λ × W	1800 × 0,040 = 72 connexions occupées en moyenne
Connexions disponibles	6 instances × 10 = 60
Verdict	72 > 60. Les requêtes feront la queue pour obtenir une connexion, le temps d'attente augmentera sans limite au pic, et les temps de réponse grimperont jusqu'à ce que les délais d'expiration se déclenchent.
Objectif avec marge	Visez un taux d'utilisation du pool d'environ 70 % au plus : 72 ÷ 0,7 ≈ 103 connexions, soit environ 18 par instance
Vérifier la base de données	Le serveur de base de données doit aussi accepter 103 connexions ; sinon, réduisez W (requêtes plus rapides, moins d'allers-retours) ou le nombre d'appels à la base par requête
Et si les requêtes deviennent plus rapides ?	Ramener W à 25 ms donne L = 1800 × 0,025 = 45 connexions, ce qui tient dans les 60 existantes avec un taux d'utilisation de 75 %

La loi de Little convertit un objectif de débit et une latence en un chiffre de concurrence que l'on peut comparer directement à une limite configurée.

Elle donne des moyennes. Les rafales poussent le nombre instantané plus haut, c'est pourquoi l'exemple ajoute une marge au lieu de dimensionner à exactement 72.

Taux d’utilisation et files d’attente : pourquoi les systèmes s’effondrent brutalement

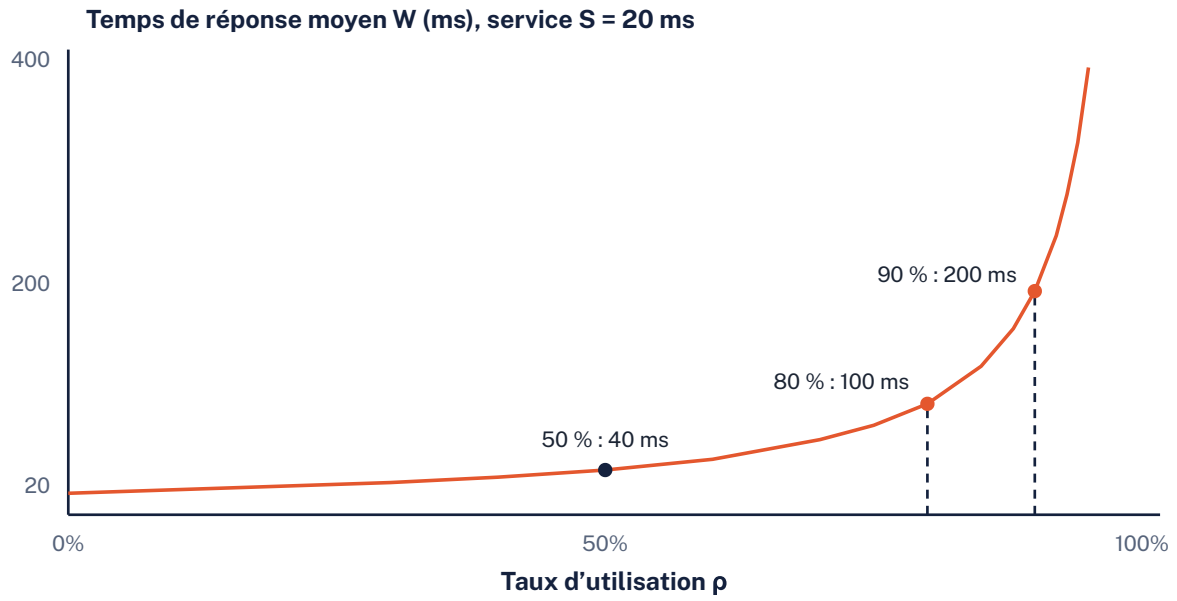
Le taux d’utilisation (ρ, rho) est la fraction du temps pendant laquelle une ressource est occupée. Un serveur qui reçoit 40 requêtes par seconde, nécessitant chacune 20 ms de service, est occupé 40 × 0,020 = 0,8 du temps : il est utilisé à 80 %. L’intuition suggère qu’une ressource occupée à 80 % ne devrait être qu’un peu plus lente qu’une ressource occupée à 50 %. La théorie des files d’attente dit le contraire, car les requêtes n’arrivent pas de façon régulière ; elles se regroupent, et quand elles se regroupent, les dernières attendent.

Le modèle le plus simple est la file M/M/1 : arrivées aléatoires, temps de service aléatoires et un seul serveur. Son temps de réponse moyen est **W = S ÷ (1 − ρ)**, où S est le temps de service moyen. À 50 % d’utilisation, W = 2S. À 80 %, W = 5S. À 90 %, W = 10S. À 99 %, W = 100S. Le dénominateur tend vers zéro à mesure que le taux d’utilisation approche 100 %, si bien que le temps de réponse augmente sans limite. Les systèmes réels ne sont pas exactement des M/M/1, et les pools multiserveurs se comportent mieux au même taux d’utilisation, mais la forme est universelle : plate, puis un coude, puis une falaise.

File M/M/1 : temps de réponse moyen pour un temps de service de 20 ms

Taux d’utilisation ρ	W = S ÷ (1 − ρ)	Temps de réponse moyen	Dont attente
50%	20 ÷ 0.5	40 ms	20 ms
70%	20 ÷ 0.3	67 ms	47 ms
80%	20 ÷ 0.2	100 ms	80 ms
90%	20 ÷ 0.1	200 ms	180 ms
95%	20 ÷ 0.05	400 ms	380 ms
99%	20 ÷ 0.01	2000 ms	1980 ms

La falaise du taux d'utilisation



Dans une file M/M/1, le temps de réponse moyen double entre 80 % et 90 % de taux d'utilisation et continue de monter en flèche vers 100 %.

Les leçons pratiques sont directes. Ne prévoyez pas de faire tourner des ressources critiques à près de 100 % d'occupation ; gardez une marge pour que les rafales ne vous fassent pas passer le coude. Surveillez ensemble le taux d'utilisation et la longueur des files (la méthode USE de Brendan Gregg vérifie l'utilisation, la saturation et les erreurs de chaque ressource). Et n'oubliez pas que la falaise rend les petits changements dangereux : à 90 % d'utilisation, une hausse du trafic d'environ 6 % seulement vous amène à 95 % et double le temps de réponse moyen. Le chapitre 9 traite des files d'attente, de la back pressure et de la planification de la capacité dans Asynchronisme, files d'attente et back pressure.

SI VOUS DÉBUTEZ

Pas à pas : pourquoi un serveur chargé fait attendre les requêtes

Le point qui fait le plus souvent trébucher ici est de comprendre pourquoi l'attente augmente si fortement à mesure qu'une ressource est plus occupée, alors que chaque requête ne demande pas plus de travail qu'avant. Suivez un serveur dont le trafic augmente.

- Chaque requête demande en moyenne 20 ms de travail : c'est son temps de service. À 40 requêtes par seconde, le serveur est occupé 40 × 0,020 = 0,8 du temps ; son taux d'utilisation est donc de 80 %.
- Si les requêtes arrivaient régulièrement, une toutes les 25 ms, chacune trouverait le serveur libre et ne prendrait que 20 ms.
- Les requêtes réelles arrivent au hasard, elles se regroupent donc. Si trois arrivent ensemble, la première est servie immédiatement, la deuxième attend environ 20 ms et la troisième environ 40 ms. Il peut ensuite y avoir un creux pendant lequel le serveur reste inactif, et ce temps d'inactivité ne peut pas être mis de côté pour le groupe suivant.
- À 50 % d'occupation, les creux sont nombreux, et les files se vident vite. À 80 %, ils sont plus rares, et la file d'un groupe est souvent encore là quand le suivant arrive. Près de 100 %, il n'y a presque plus de temps libre, et les files deviennent très longues.
- Le tableau de la section pour un seul serveur avec des arrivées aléatoires (la file M/M/1) montre l'effet : un temps de réponse moyen de 40 ms à 50 % d'occupation, de 100 ms à 80 % et de 200 ms à 90 %. Passer de 40 à 45 requêtes par seconde fait passer le serveur de 80 % à 90 % d'occupation et double le temps de réponse moyen.
- **La leçon :** le travail lui-même n'a pas ralenti ; le temps supplémentaire est entièrement dû à la mise en file d'attente. Gardez les ressources critiques bien en dessous de leur pleine capacité, car près de la saturation, une petite hausse du trafic entraîne une forte hausse du temps de réponse.

Scalabilité : charge, élasticité, scalabilité verticale et horizontale

La scalabilité est la capacité à absorber plus de charge en ajoutant des ressources, idéalement avec une capacité qui croît proportionnellement et sans refonte. Elle n'a de sens qu'une fois que l'on a précisé ce qu'est la « charge ». Les paramètres de charge diffèrent d'un système à l'autre : requêtes par seconde pour une API, connexions simultanées pour un service de messagerie instantanée, ratio lectures/écritures pour un réseau social, volume de données pour une plateforme d'analytique, ou nombre d'abonnés du compte le plus suivi pour un fil d'actualité. Un scénario de scalabilité nomme le paramètre de charge, la croissance et la réponse qui doit être

préservée : « quand les commandes quotidiennes passent de 50 000 à 500 000, la latence p95 du paiement reste inférieure à 1,5 s grâce à l'ajout d'instances, avec un coût par commande au plus 20 % plus élevé ».

L'élasticité ajoute le temps : la capacité à ajouter et retirer de la capacité automatiquement à mesure que la charge monte et descend, pour qu'un site de billetterie puisse grandir pour une mise en vente et rétrécir ensuite. C'est une grande partie de l'argument économique en faveur des plateformes cloud. Il existe deux façons fondamentales d'ajouter de la capacité. La scalabilité verticale (scaling up) donne à une machine plus de CPU, de mémoire ou un stockage plus rapide ; elle est simple, ne demande aucune modification du logiciel, et se heurte à des limites strictes et à des prix croissants. La scalabilité horizontale (scaling out) ajoute des machines et répartit le travail ; elle n'a pas de plafond fixe, mais exige que le logiciel soit conçu pour cela, avec des instances sans état, de l'équilibrage de charge et des données partitionnées. Le chapitre 9 traite des stratégies en détail dans Stratégies de passage à l'échelle : verticale, horizontale et cube de scalabilité.

La loi d'Amdahl : la limite du parallélisme

Ajouter des processeurs ou des nœuds n'aide que pour le travail qui peut se faire en parallèle. La loi d'Amdahl, tirée de l'article de Gene Amdahl de 1967 sur les limites du calcul parallèle, exprime cette idée. Si une fraction s du travail est séquentielle (elle doit se faire une étape après l'autre) et que le reste, 1 − s, peut être réparti parfaitement sur N processeurs, la meilleure accélération possible est **accélération(N) = 1 ÷ (s + (1 − s) ÷ N)**. Quand N croît sans limite, l'accélération tend vers 1 ÷ s. Avec 10 % de travail séquentiel, aucun nombre de processeurs ne peut rendre la tâche plus de dix fois plus rapide.

EXEMPLE COMMENTÉ

Exemple commenté : un traitement de facturation nocturne

Le traitement de facturation nocturne d'un opérateur télécom prend 8 heures sur un serveur. Le profilage montre que 10 % du temps est séquentiel (chargement des tarifs, puis écriture d'un fichier récapitulatif unique) ; les 90 % restants valorisent les appels de chaque client indépendamment. Le traitement doit se terminer en moins de 1,5 heure. Combien de serveurs faut-il ?

Fraction séquentielle s	0,10 ; fraction parallèle 0,90
Accélération nécessaire	8 h ÷ 1.5 h ≈ 5.33
N = 4	1 ÷ (0,10 + 0,90 ÷ 4) = 1 ÷ 0,325 ≈ 3,08 ; durée 8 ÷ 3,08 ≈ 2,6 h
N = 8	1 ÷ (0,10 + 0,1125) = 1 ÷ 0,2125 ≈ 4,71 ; durée ≈ 1,70 h
N = 16	1 ÷ (0,10 + 0,05625) = 1 ÷ 0,15625 = 6,4 ; durée = 1,25 h
Plus petit N qui atteint l'objectif	Il faut 0,10 + 0,90 ÷ N ≤ 1 ÷ 5,33 = 0,1875, donc N ≥ 0,90 ÷ 0,0875 ≈ 10,3 : 11 serveurs donnent 1 ÷ (0,10 + 0,0818) ≈ 5,50, durée ≈ 1,45 h
N = 64	1 ÷ (0,10 + 0,0141) ≈ 8,77 ; durée ≈ 0,91 h
Limite quand N croît	1 ÷ 0,10 = 10 ; durée jamais inférieure à 0,8 h
Variante	Réduire de moitié la partie séquentielle (s = 0,05), par exemple en chargeant les tarifs en parallèle. Alors 8 serveurs donnent 1 ÷ (0,05 + 0,11875) ≈ 5,93, durée ≈ 1,35 h : l'objectif est atteint avec 8 serveurs au lieu de 11.

Passer de 16 à 64 serveurs (quatre fois le coût) ne réduit la durée que de 1,25 h à 0,91 h. Les rendements diminuent vite.

Réduire la fraction séquentielle coûte souvent bien moins cher qu'ajouter du matériel. Les architectes traquent les parties séquentielles cachées : un verrou unique, une ligne de base de données que tout le monde met à jour, un coordinateur central.

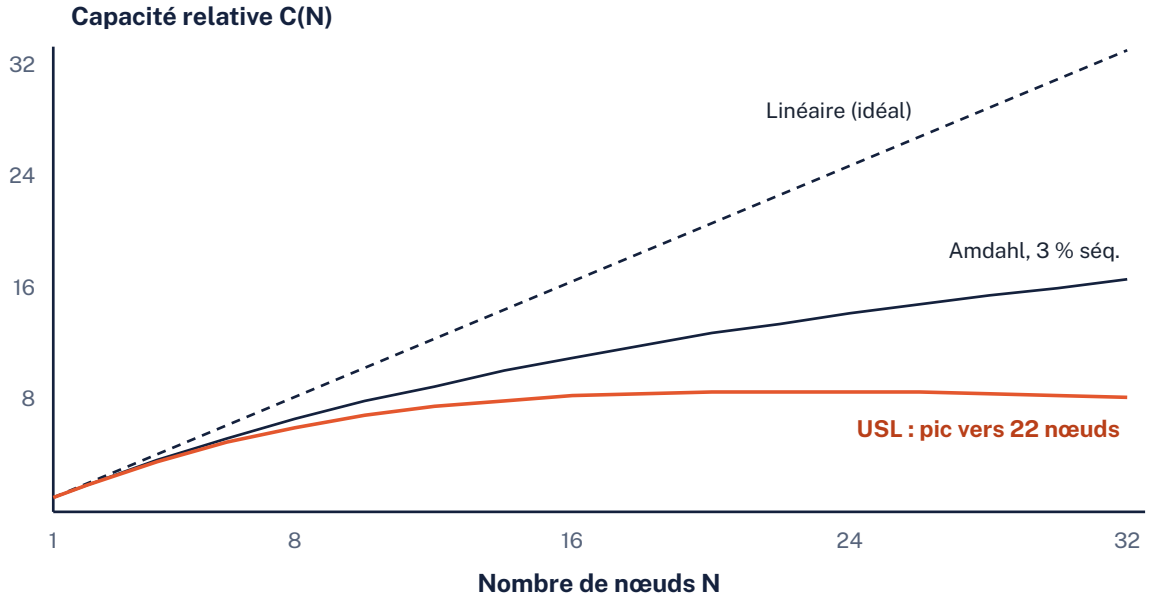
La loi universelle de scalabilité

La loi d'Amdahl suppose que des processeurs supplémentaires n'aggravent jamais les choses ; l'accélération se contente de plafonner. Les systèmes distribués réels, eux, se dégradent souvent : au-delà d'un certain point, ajouter des nœuds réduit le débit. La loi universelle de scalabilité (USL) de Neil Gunther modélise ce phénomène avec deux pénalités. La contention (α, alpha) est le coût de l'attente pour accéder aux ressources partagées, le même effet que la

fraction séquentielle d'Amdahl. La cohérence (β, bêta) est le coût du maintien de la cohérence des données entre les nœuds : chaque nœud doit échanger des informations avec tous les autres, si bien que ce coût croît à peu près comme le carré du nombre de nœuds.

La capacité relative est **$C(N) = N \div (1 + \alpha(N - 1) + \beta N(N - 1))$**. Avec $\beta = 0$, elle se ramène à la loi d'Amdahl. Avec tout β supérieur à zéro, la capacité atteint un pic vers $N = \sqrt{((1 - \alpha) \div \beta)}$, puis diminue. Pour $\alpha = 0{,}03$ et $\beta = 0{,}002$, le pic se situe vers $\sqrt{(0{,}97 \div 0{,}002)} \approx 22$ nœuds, pour une capacité d'environ 8,6 fois celle d'un nœud ; à 32 nœuds, la capacité est retombée à environ 8,2. Gunther décrit le modèle dans *Guerrilla Capacity Planning* (2007). Son intérêt pratique est que α et β peuvent être ajustés à partir d'une poignée de mesures de tests de charge, ce qui permet de prévoir à quel point le passage à l'échelle cessera d'être rentable avant d'acheter le matériel.

Scalabilité linéaire, d'Amdahl et USL



La scalabilité idéale est linéaire ; la loi d'Amdahl s'aplatit à cause du travail séquentiel ; l'USL intègre en plus les coûts de cohérence, si bien que la capacité culmine puis décline.

Tactiques de performances : vue d'ensemble

Bass, Clements et Kazman regroupent les tactiques de performances en deux familles. La première consiste à maîtriser la demande de ressources : réduire la quantité de travail qui arrive ou le coût de chaque unité. La seconde consiste à gérer les ressources : faire travailler davantage les ressources disponibles ou en ajouter. La plupart des conceptions réelles combinent plusieurs tactiques ; chacune a aussi des coûts ailleurs, ce qui explique qu'il s'agisse de décisions architecturales.

Les tactiques de performances et où le cours les traite

Tactique	Ce qu'il fait	Coût ou risque typique	Pour en savoir plus
Réduire la demande : limitation de débit (rate limiting), contrôle d'admission, salles d'attente virtuelles	Plafonne ou lisse la charge entrante pour que le cœur reste en deçà de son coude	Certains utilisateurs sont retardés ou refusés	Chapitre 9 : files d'attente et back pressure
Réduire le travail par requête : algorithmes efficaces, charges utiles plus petites, moins d'allers-retours	Réduit le temps de service	Effort de développement ; code parfois moins souple	Chapitre 0 : données et algorithmes
Prioriser et ordonnancer	Sert d'abord le travail important (les commandes payées avant les rapports)	Le travail peu prioritaire risque la famine	Chapitre 9 : patrons de résilience
Mise en cache	Garde les résultats près de l'utilisateur pour éviter de refaire le travail	Données périmées, complexité de l'invalidation, coût en mémoire	Chapitre 7 : mise en cache
Concurrence et asynchronisme	Traite en parallèle le travail indépendant ; retire le travail lent du chemin de la requête	Raisonnement plus difficile, problèmes d'ordre et de cohérence	Chapitre 6 : communication
Réplication	Ajoute des copies de services ou de données pour répartir les lectures et la charge	Cohérence entre les copies ; coût	Chapitre 7 : réplication et partitionnement
Partitionnement (sharding)	Divise les données et le travail pour que chaque nœud en traite une part	Les requêtes inter-partitions et le rééquilibrage sont difficiles	Chapitre 7 : réplication et partitionnement
Ajouter des ressources : scalabilité verticale ou horizontale	Augmente la capacité	Coût ; limité par le travail séquentiel et de cohérence	Chapitre 9 : stratégies de passage à l'échelle

ATTENTION

Erreurs courantes

- Spécifier ou surveiller des moyennes au lieu de percentiles, et faire la moyenne de percentiles entre serveurs.
- Faire des tests de charge à débit constant alors que le trafic réel arrive par rafales ; les files se forment pendant les rafales.
- Prévoir de faire tourner une ressource critique à 90 % d'utilisation ou plus, sans laisser de place aux pics.
- Supposer que plus de serveurs signifie toujours plus de capacité, sans mesurer les coûts séquentiels et de cohérence.
- Optimiser le temps de service en ignorant le temps d'attente, qui est généralement la plus grande part sous charge.
- Mesurer uniquement côté serveur, alors que les utilisateurs subissent aussi le temps réseau et le rendu côté client.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Un service reçoit 500 requêtes par seconde et chacune prend 80 ms. Combien sont en cours en moyenne ?

D'après la loi de Little, L = 500 × 0,08 = 40 requêtes en cours.

▼ Avec la formule M/M/1, quel est le temps de réponse moyen d'un serveur dont le temps de service est de 10 ms, à 75 % d'utilisation ?

W = 10 ÷ (1 − 0,75) = 40 ms, dont 30 ms d'attente.

▼ Pourquoi le p99 est-il généralement une meilleure mesure pour une exigence que la moyenne ?

Les distributions des temps de réponse sont asymétriques ; la moyenne peut paraître satisfaisante alors qu'une part non négligeable des utilisateurs subit des réponses très lentes, et le p99 met cette queue en évidence.

▼ Si 20 % d'une tâche est séquentielle, quelle est l'accélération maximale obtenue en ajoutant des processeurs ?

1 ÷ 0,2 = 5 fois, quel que soit le nombre de processeurs ajoutés.

▼ **À vous : Un service de commandes traite 2400 requêtes par seconde, chacune occupant une connexion à la base de données pendant 30 ms. Il compte 8 instances de 8 connexions chacune. Est-ce suffisant ? Quelle taille de pool fixeriez-vous par instance, en visant une utilisation de 70 % ?**

Selon la loi de Little, 2400 x 0,030 = 72 connexions occupées en moyenne, et il n'en existe que 8 x 8 = 64. Les requêtes font donc la queue pour obtenir des connexions et le temps d'attente grimpe en période de pointe. Pour une utilisation de 70 %, il faut 72 / 0,7 = environ 103 connexions, soit 103 / 8 = environ 13 par instance, 104 au total. Vérifiez que la base de données supporte ce nombre, et réduisez les 30 ms si ce n'est pas le cas.

SOURCES

- Dean and Barroso, The Tail at Scale (Google Research): https://research.google/pubs/pub40801/
- Neil Gunther: How to quantify scalability (the USL): https://www.perfdynamics.com/Manifesto/USLscalability.html
- Google SRE book: Monitoring distributed systems (latency and percentiles): https://sre.google/sre-book/monitoring-distributed-systems/
- Brendan Gregg: The USE method: https://brendangregg.com/usemethod.html

SECTION 2.5 SUR 2.8 · INTERMÉDIAIRE · 22 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Disponibilité et fiabilité : « neuf », MTBF, MTTR et tolérance aux pannes

AVANT DE LIRE

Cette section s'appuie sur :

- 0.7 Serveurs et cloud
- 2.2 Modèles de qualité
- 2.3 Scénarios d'attributs de qualité

Termes déjà vus qu'elle utilise : Serveur, Équilibreur de charge (load balancer), Composant, Base de données, Centre de données

L'IDÉE EN BREF

Pensez à l'épicerie du coin. La disponibilité, c'est la fréquence à laquelle la porte est ouverte quand vous arrivez. La fiabilité, c'est le fait que, une fois à l'intérieur, la caisse fonctionne du début à la fin de votre achat. Les magasins restent ouverts grâce à une caisse de secours, à un deuxième membre du personnel et à un moyen rapide de repérer et de régler les problèmes. Il en va de même pour les logiciels : les architectes décident de la capacité de réserve à conserver, de la façon de repérer vite les défaillances et de la rapidité du rétablissement, et ils expriment l'objectif sous forme de chiffre, par exemple 99,9 % du temps.

CE QU'IL FAUT SAVOIR

- La disponibilité est la fraction du temps ou des requêtes pendant laquelle le service est prêt ; la fiabilité est la probabilité de fonctionner sans défaillance pendant une période donnée ; ce sont deux notions différentes, et un système peut obtenir un bon score sur l'une et un mauvais sur l'autre.
- Une faute est une cause, une erreur est un état interne incorrect et une défaillance est un écart que les utilisateurs peuvent observer ; les tactiques préviennent les fautes, détectent les erreurs ou empêchent les erreurs de devenir des défaillances.
- Indisponibilité admise = (1 - disponibilité) x période : 99,9 % représente environ 8,76 heures par an ou 43,2 minutes par mois ; 99,99 % environ 52,6 minutes par an.
- A = MTBF / (MTBF + MTTR) : réduire le temps de réparation grâce à l'automatisation et à la redondance est donc souvent moins cher et plus efficace que de rendre les défaillances plus rares.

- En série, les disponibilités se multiplient ; en parallèle, ce sont les indisponibilités qui se multiplient, mais seulement si les copies tombent en panne indépendamment les unes des autres et si le failover fonctionne.
- Le RTO (le délai de restauration) et le RPO (la quantité de données que vous pouvez perdre) sont fixés par le métier et déterminent la conception et le coût de la reprise après sinistre.
- Les SLI mesurent, les SLO fixent des objectifs internes et les SLA sont des contrats ; un SLO transforme la disponibilité en budget d'erreur que les équipes peuvent dépenser en changements.

Quatre mots souvent confondus

Pour les utilisateurs, un système fonctionne ou ne fonctionne pas. Les architectes ont besoin de mots plus fins, car chaque mot appelle une réponse de conception différente. La disponibilité est la proportion du temps (ou des requêtes) pendant laquelle le système est prêt à fournir son service quand on le lui demande. La fiabilité est la probabilité que le système fonctionne correctement, sans défaillance, pendant une période donnée et dans des conditions données. La tolérance aux pannes est la capacité à continuer de fournir le service même lorsqu'une partie est défaillante. La résilience est la capacité plus large à absorber une perturbation, à maintenir le service essentiel (éventuellement dégradé) puis à revenir à la normale.

L'ISO/IEC 25010:2023 range la disponibilité, la tolérance aux pannes et la capacité de récupération, ainsi que l'absence de défauts (qui a remplacé le terme maturité de 2011), parmi les sous-caractéristiques de la fiabilité. Bass, Clements et Kazman, dans *Software Architecture in Practice*, traitent la disponibilité comme un attribut de qualité à part entière, parce qu'elle oriente un très grand nombre de décisions architecturales. Aucune de ces deux visions n'est fausse ; utilisez celle qui facilite la discussion, mais précisez bien de quel mot vous parlez.

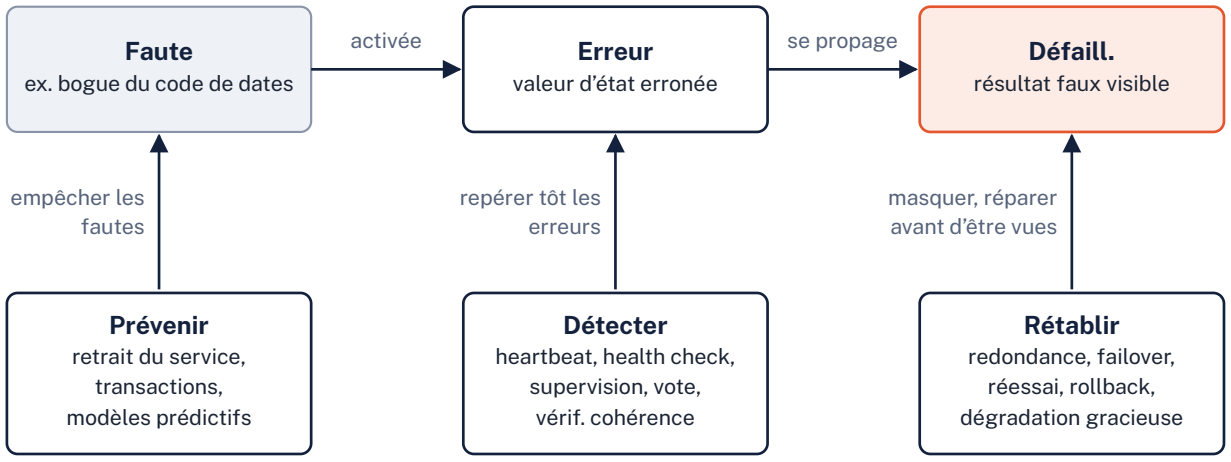
La différence entre disponibilité et fiabilité compte en pratique. Imaginez un service qui plante une fois par heure et redémarre en une seconde. Sa disponibilité est de 3599 / 3600 = 99,97 %, ce qui semble excellent. Sa fiabilité sur un traitement par lots de deux heures est médiocre : si les défaillances surviennent au hasard au rythme d'une par heure, la probabilité de tourner deux heures sans aucune est de e^{-2}, soit environ 13,5 %. Un terminal de paiement se soucie de la disponibilité ; un long calcul scientifique, un robot chirurgical ou un avion se soucient de la fiabilité sur toute la durée de la mission.

Faute, erreur et défaillance

La communauté de la sûreté de fonctionnement (Avižienis, Laprie, Randell et Landwehr, 2004) distingue trois idées que le langage courant confond. Une faute (fault) est la cause : un bogue, un disque en panne, une erreur de configuration, un certificat expiré. Une faute peut rester dormante pendant des années. Lorsqu'elle est activée, elle produit une erreur : une partie de l'état interne du système qui est incorrecte, comme un enregistrement corrompu ou une valeur erronée en mémoire. Si l'erreur atteint la frontière du service, il en résulte une défaillance (failure) : le service fourni aux utilisateurs s'écarte de ce qu'il devrait être.

Bass, Clements et Kazman reprennent la même idée plus simplement : une défaillance est un écart par rapport à la spécification que les utilisateurs du système peuvent observer, et la cause d'une défaillance est une faute. Pour l'architecte, l'essentiel est que chaque maillon de la chaîne est un point d'intervention. Vous pouvez empêcher les fautes d'apparaître (revue de code, retrait d'un composant du service avant son usure), détecter les erreurs avant qu'elles ne se propagent (validation, contrôles de santé) et contenir ou masquer les erreurs pour qu'elles ne deviennent jamais des défaillances (redondance, nouvelles tentatives, transactions).

La chaîne et les points d'action des tactiques



Familles de tactiques de Bass, Clements et Kazman, Software Architecture in Practice (4e édition, 2021)

Les tactiques de prévention empêchent les fautes d'apparaître, les tactiques de détection repèrent les erreurs et les tactiques de rétablissement empêchent les erreurs de devenir des défaillances visibles.

Compter les « neuf »

Les objectifs de disponibilité s'écrivent généralement sous forme de pourcentage avec un certain nombre de « neuf » : 99,9 %, ce sont « trois neuf ». Il est utile de convertir chaque objectif en temps d'indisponibilité admis, car un pourcentage masque le peu de temps dont vous disposez réellement. Le calcul est simple : indisponibilité = (1 - disponibilité) x période. Une

année de 365 jours compte 525 600 minutes, donc 99,9 % autorise 0,001 x 525 600 = 525,6 minutes, soit environ 8,76 heures. Pour un mois de 30 jours (43 200 minutes), cela autorise 43,2 minutes.

Chaque « neuf » supplémentaire divise la marge par dix, alors que le coût d'ingénierie augmente généralement bien plus vite. À 99,99 %, vous disposez d'environ 4,3 minutes par mois : moins que le temps nécessaire pour qu'une personne d'astreinte soit alertée, se réveille, se connecte et comprenne ce qui ne va pas. Au-delà de trois ou quatre « neuf », le rétablissement doit donc être automatique ; les humains sont trop lents pour être dans la boucle.

Indisponibilité admise pour les objectifs de disponibilité courants (année de 365 jours, mois de 30 jours)

Disponibilité	Par an	Par mois	Par semaine	Par jour
99 % (deux neuf)	3,65 jours (87,6 h)	7,2 heures	1,68 heure (100,8 min)	14,4 minutes
99,9 % (trois neuf)	8,76 heures	43,2 minutes	10,1 minutes	1,44 minute
99.95%	4,38 heures	21,6 minutes	5,04 minutes	43,2 secondes
99,99 % (quatre neuf)	52,6 minutes	4,32 minutes	1,01 minute	8,64 secondes
99,999 % (cinq neuf)	5,26 minutes	25,9 secondes	6,05 secondes	0,86 seconde

ATTENTION

Fondée sur le temps ou sur les requêtes ?

Le tableau suppose que la disponibilité se mesure comme le temps de fonctionnement rapporté au temps calendaire. Beaucoup de services en ligne la mesurent plutôt comme la proportion de requêtes réussies : 999 000 bonnes réponses sur 1 000 000, c'est 99,9 %. La vision fondée sur les requêtes gère naturellement les pannes partielles (une région hors service, ou seule l'étape de paiement en échec) et pondère davantage les heures chargées que les heures creuses. Quel que soit votre choix, notez exactement ce qui compte comme un succès (codes de statut, une limite de latence ?), où la mesure est faite (au niveau de l'équilibreur de charge ou dans le navigateur de l'utilisateur) et sur quelle fenêtre. Deux équipes peuvent toutes deux annoncer 99,9 % en voulant dire des choses très différentes.

MTBF, MTTF et MTTR

Trois moyennes relient les défaillances et les réparations. Le temps moyen jusqu'à la défaillance (MTTF) est la durée moyenne de fonctionnement d'un composant avant qu'il ne tombe en panne. Le temps moyen de réparation (MTTR) est la durée moyenne entre la défaillance et le rétablissement du service (certaines organisations lisent le R comme recover ou restore ; l'idée est la même, mais précisez quel chronomètre vous utilisez : à partir de la défaillance, de la détection ou de la première alerte). Le temps moyen entre défaillances (MTBF) est la durée moyenne entre une défaillance et la suivante.

La formule classique en régime permanent est **A = MTBF / (MTBF + MTTR)**. En toute rigueur, pour un système réparable, MTBF = MTTF + MTTR, et l'expression exacte est MTTF / (MTTF + MTTR) : le temps de fonctionnement divisé par le temps de fonctionnement plus le temps d'arrêt. Les manuels écrivent souvent MTBF là où ils veulent dire temps de fonctionnement moyen ; lorsque les réparations sont courtes par rapport au temps entre défaillances, ce qui est généralement le cas, la différence est négligeable. La formule suppose une moyenne sur le long terme et ne dit rien de la répartition des pannes : c'est pourquoi elle se combine bien avec une mesure de fiabilité et avec des percentiles du temps de réparation.

EXEMPLE COMMENTÉ

Exemple commenté : vaut-il mieux tomber en panne moins souvent ou se rétablir plus vite ?

Le service des commandes d'une app de livraison de repas tombe en panne en moyenne une fois tous les 30 jours (MTBF = 720 heures). Chaque défaillance demande 4 heures de réparation, car il faut alerter un ingénieur, qui doit diagnostiquer le problème et restaurer une base de données à la main. Le métier veut faire mieux que 99,9 %.

Disponibilité actuelle	720 / (720 + 4) = 720 / 724 = 0,99448, soit 99,45 %
Indisponibilité actuelle par an	(1 - 0,99448) x 8760 h = environ 48,4 heures
Option 1 : doubler le MTBF	Meilleur matériel et meilleurs tests ; une défaillance tous les 60 jours. A = 1440 / 1444 = 0,99723 (99,72 %), soit environ 24,3 heures par an
Option 2 : ramener le MTTR à 15 minutes	Détection automatisée, secours chaud (hot standby) et failover scripté. A = 720 / 720,25 = 0,99965 (99,97 %), soit environ 3,0 heures par an
Vérification de l'objectif	Seule l'option 2 dépasse 99,9 % (8,76 heures par an)

Diviser par deux la fréquence des défaillances a divisé par deux l'indisponibilité. Diviser le temps de réparation par 16 a divisé l'indisponibilité à peu près d'autant, et a permis d'atteindre l'objectif.

Augmenter le MTBF, c'est lutter contre la physique et la complexité : vous ne pouvez pas rendre parfaits les disques, les réseaux ou les personnes. Réduire le MTTR relève en grande partie de l'architecte : détection, automatisation, redondance, petits déploiements réversibles. C'est pourquoi les pratiques d'exploitation modernes se concentrent sur le rétablissement.

Le MTTR est une moyenne et masque les longues traînes. Une seule panne de huit heures parmi de nombreuses pannes de cinq minutes pèse plus que tout le reste de l'année : suivez donc la distribution des durées d'incident, pas seulement la moyenne.

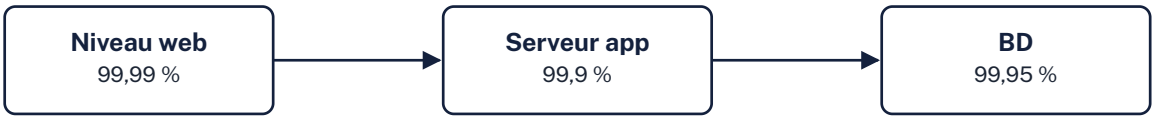
Composants en série et en parallèle

Les systèmes réels sont des chaînes de composants. Si une requête a besoin du niveau web, du serveur d'application et de la base de données, les trois doivent fonctionner : les composants sont *en série*, et leurs disponibilités se multiplient. Comme chaque facteur est inférieur à un, le produit est toujours plus faible que le maillon le plus faible. Dix composants à 99,9 % chacun donnent $0{,}999^{10}$ = 0,990, soit seulement 99,0 %.

Si, au contraire, une seule de plusieurs copies identiques suffit, les copies sont *en parallèle* (redondance). L'ensemble ne tombe en panne que lorsque toutes les copies tombent en panne en même temps, si bien qu'on multiplie les indisponibilités : $A_{parallèle} = 1 - (1 - A)^n$ pour n copies identiques. Deux serveurs à 99 % chacun donnent 1 - 0,01 x 0,01 = 99,99 %.

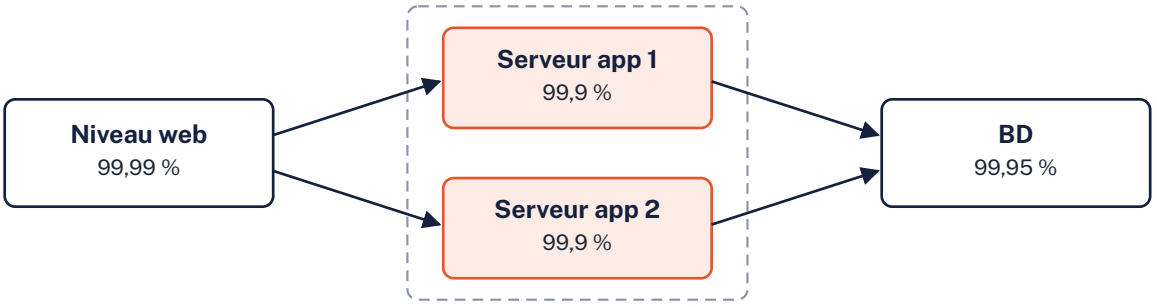
Série et parallèle, chiffres à l'appui

A. Trois composants en série : chacun doit fonctionner



0,9999 x 0,999 x 0,9995 = 0,9984 (99,84 %, environ 14 h d'arrêt par an)

B. Même chaîne avec deux serveurs app en parallèle : un seul suffit



paire : 1 - (0,001 x 0,001) = 0,999999 ; chaîne : 0,9994 (99,94 %, environ 5,3 h par an)

Dupliquer l'étage le plus faible fait passer toute la chaîne d'environ 14 heures d'indisponibilité par an à environ 5,3.

EXEMPLE COMMENTÉ

Exemple commenté : où dépenser la prochaine livre ?

Suite du schéma. La chaîne de la partie A atteint 99,84 %. Quel changement la rapproche le plus de 99,99 % ?

Ajouter un second serveur d'application	0,9999 x 0,999999 x 0,9995 = 0,99940 (99,94 %) : environ 5,3 heures par an
Puis ajouter une réplique de la base de données avec failover	Paire de bases de données : $1 - 0{,}0005^2 = 0{,}99999975$. Chaîne : 0,9999 x 0,999999 x 0,99999975 = 0,99990 (99,99 %) : environ 53 minutes par an
Ce qui limite désormais le système	Le niveau web unique (99,99 %) représente presque toute l'indisponibilité restante

Le calcul suppose que les copies tombent en panne indépendamment. Deux serveurs dans la même baie, sur la même alimentation électrique, exécutant la même version boguée ou dépendant tous deux d'un même fournisseur DNS partagent des fautes de mode commun et peuvent tomber en panne ensemble. Les gains réels sont toujours inférieurs à ce que suggère la formule.

Il suppose aussi que le failover est instantané et fonctionne toujours. Un secours qui n'a jamais été testé lors d'un vrai failover est un espoir, pas un composant.

Une règle utile des auteurs SRE de Google (Treynor, Dahlin, Rau et Beyer, 2017) : chaque dépendance critique devrait offrir à peu près un « neuf » de plus que le service qui s'appuie sur elle, sinon le service doit pouvoir s'en passer.

Tactiques de disponibilité : détecter, rétablir, prévenir

Bass, Clements et Kazman regroupent les tactiques de disponibilité en trois familles. Ce sont des options de conception, pas une liste de contrôle à appliquer intégralement ; chacune a un coût.

Les trois familles de tactiques de disponibilité

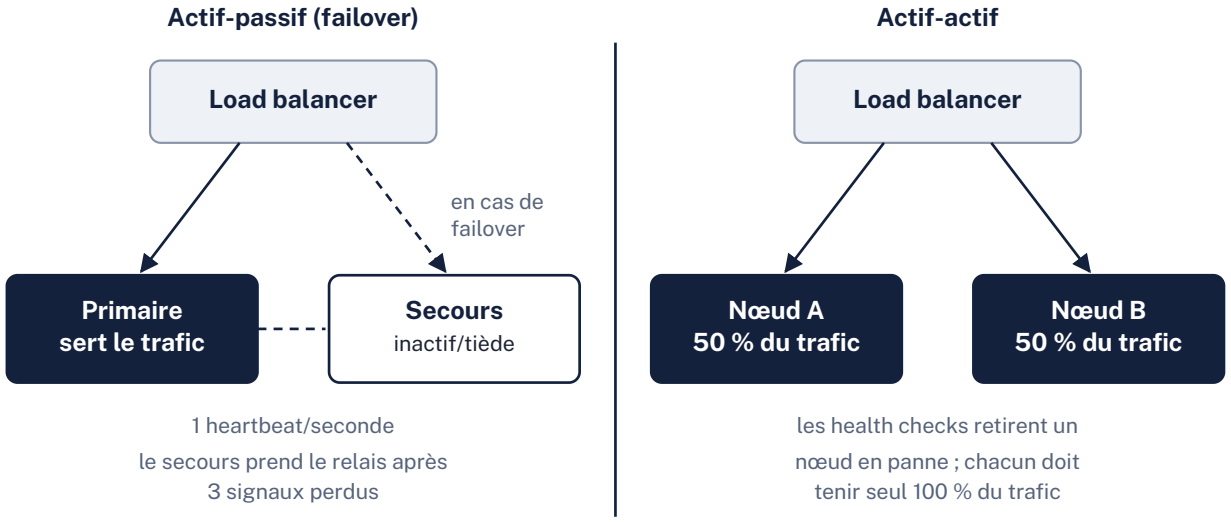
1. **Détecter les fautes**
   Heartbeat (battement de cœur) : un composant envoie périodiquement un message « je suis vivant », et le silence déclenche une action. Ping/écho : un moniteur interroge et attend une réponse. Contrôles de santé (health checks) : un endpoint comme `/health` qu'un équilibreur de charge ou un orchestrateur interroge régulièrement (un contrôle de vivacité, liveness, demande « le processus tourne-t-il ? » ; un contrôle de disponibilité, readiness, demande « peut-il servir du trafic maintenant ? »). La supervision des taux d'erreur et de la latence, les contrôles de vraisemblance des valeurs et le vote entre répliques qui calculent le même résultat.

2. **Rétablir : préparation et réparation**
   Des secours redondants à trois températures : *actif-actif* (toutes les copies servent le trafic réel), *secours chaud ou tiède* (une copie tenue à jour et prête) et *réserve froide* (démarrée seulement en cas de besoin). Le failover (basculement) fait passer le trafic vers une copie saine. Les nouvelles tentatives traitent les fautes transitoires (avec des limites et un backoff, sinon elles deviennent une surcharge auto-infligée). Le retour arrière (rollback) ramène à un état ou à une version connus comme bons. La dégradation gracieuse maintient le service essentiel sans une fonctionnalité défaillante : la boutique continue de prendre des commandes même si les recommandations sont hors service.

3. **Rétablir : réintroduction**
   Remettre en service un composant réparé en toute sécurité : le faire tourner en mode fantôme (shadow), resynchroniser son état, redémarrer par paliers croissants (un thread, puis un processus, puis une machine).

4. **Prévenir les fautes**
   Retrait du service (sortir un nœud de la rotation pour maintenance, ou planifier un redémarrage avant qu'une fuite mémoire connue ne frappe), transactions (mises à jour en tout ou rien pour qu'un plantage ne puisse pas laisser un état à moitié écrit), modèles prédictifs (remplacer un disque quand ses compteurs d'erreurs grimpent) et prévention des exceptions (typage plus strict, validation des entrées).

Deux formes de redondance



L'actif-passif paie une capacité inactive et un délai de failover ; l'actif-actif utilise toute la capacité, mais chaque nœud doit survivre à la perte de l'autre.

Choisir entre actif-passif et actif-actif

L'actif-passif est plus simple à raisonner, en particulier pour les composants avec état comme une base de données relationnelle avec un seul primaire accessible en écriture. Ses coûts sont le secours inactif, un délai de failover (détection plus promotion, souvent plusieurs dizaines de secondes) et le risque que le secours se soit discrètement désynchronisé. Un piège classique est le *split brain* (cerveau divisé) : le réseau entre les deux est coupé, chacun croit l'autre mort, et tous deux acceptent des écritures. Pour s'en prémunir, il faut un quorum ou un arbitre externe.

L'actif-actif utilise toute la capacité et n'a pas d'étape de promotion, mais chaque nœud doit être dimensionné pour supporter toute la charge si son partenaire tombe en panne. Deux nœuds fonctionnant chacun à 70 % d'utilisation seront sollicités à 140 % quand l'un d'eux tombe, et le survivant s'effondre : une conception redondante qui échoue quand même. L'actif-actif vous oblige aussi à résoudre la cohérence des données entre les copies, un vaste sujet en soi (voir cohérence, CAP et PACELC).

Reprise après sinistre : RTO et RPO

La redondance au sein d'un seul centre de données ne vous protège pas contre la perte du centre de données entier, d'une région, d'un compte cloud ou de vos données à cause d'un rançongiciel. La planification de la reprise après sinistre s'appuie sur deux chiffres qui devraient venir du métier, pas des ingénieurs. L'objectif de temps de reprise (RTO) est la durée

maximale acceptable pour restaurer le service après un sinistre. L'objectif de point de reprise (RPO) est la perte de données maximale acceptable, mesurée en temps : un RPO de 15 minutes signifie que vous pouvez perdre au plus les 15 dernières minutes de modifications.

Ces deux chiffres conduisent à des conceptions et à des coûts très différents. Des sauvegardes nocturnes copiées dans une autre région donnent un RPO pouvant aller jusqu'à 24 heures et un RTO de nombreuses heures (restaurer des téraoctets prend du temps). Une réplication asynchrone continue vers un secours tiède dans une autre région peut donner un RPO de quelques secondes et un RTO de quelques minutes, pour un coût d'infrastructure peut-être doublé. La réplication synchrone peut ramener le RPO près de zéro, mais ajoute à chaque écriture le temps d'aller-retour entre les sites. Les fournisseurs cloud décrivent une échelle de stratégies : sauvegarde et restauration, pilot light (veilleuse), warm standby (secours tiède) et actif-actif multisite, chacune plus rapide et plus coûteuse que la précédente. Les détails relèvent de multirégion et reprise après sinistre.

Quelle que soit la stratégie, une sauvegarde que vous n'avez jamais restaurée n'est pas une sauvegarde. Testez les restaurations à intervalles réguliers et chronométrez-les ; le temps de restauration mesuré est votre RTO réel.

Des objectifs aux SLI, SLO et SLA

Les pratiques d'exploitation modernes expriment la disponibilité avec trois termes liés, traités en détail dans SLI, SLO et budgets d'erreur. Un indicateur de niveau de service (SLI) est la mesure, par exemple la proportion de requêtes de paiement qui aboutissent en moins de 800 ms. Un objectif de niveau de service (SLO) est la cible interne pour cette mesure sur une fenêtre donnée, par exemple 99,9 % sur 30 jours. Un accord de niveau de service (SLA) est un contrat avec les clients qui prévoit des conséquences, comme des avoirs de service, si un niveau n'est pas atteint ; il est normalement fixé moins strictement que le SLO, pour que l'équipe soit avertie avant que le contrat ne soit violé.

Le SLO transforme la disponibilité en budget. Un SLO mensuel de 99,9 % autorise l'échec de 0,1 % des requêtes ; avec 2 millions de requêtes par mois, cela fait 2000 requêtes en échec, soit environ 43 minutes de panne totale. Les équipes dépensent ce budget d'erreur en prises de risque : mises en production, expériences, migrations. Quand il est épuisé, elles ralentissent et investissent dans la fiabilité. Cela relie les « neuf » abstraits de cette section aux décisions du quotidien.

SI VOUS DÉBUTEZ

Si vous débutez

Vous n'avez pas besoin de mémoriser le tableau des « neuf ». Retenez la méthode : un moins le pourcentage, multiplié par la période. 99,9 % laisse 0,1 % d'un mois de 30 jours, soit 0,001 x 43 200 minutes = 43,2 minutes. Toutes les autres lignes se calculent de la même façon.

Erreurs courantes

Promettre des « neuf » sans tenir compte des dépendances : un service à 99,99 % bâti sur une seule base de données à 99,9 % ne peut pas atteindre son objectif. Ajouter de la redondance sans supprimer les causes communes (la même version, le même déploiement de configuration, le même certificat qui expire sur tous les nœuds à minuit). Mesurer la disponibilité depuis l'intérieur du système, où tout semble au vert, alors que les utilisateurs ne peuvent pas l'atteindre à cause du DNS ou d'un réseau de diffusion de contenu. Prévoir un failover sans jamais le répéter ; l'ingénierie du chaos existe précisément pour exercer ces chemins délibérément (ingénierie du chaos). Enfin, courir après un chiffre pour lui-même : un système de planning du personnel utilisé aux heures de bureau n'a pas besoin de cinq neuf, et l'argent sera mieux dépensé ailleurs.

Pour rendre la disponibilité testable, écrivez-la sous forme de scénario (section 2.3) : «Lorsque le serveur de base de données primaire tombe en panne aux heures de pointe, le système le détecte en moins de 10 secondes et recommence à accepter des commandes en moins de 60 secondes, sans perdre aucune commande confirmée.» Chaque élément de cette phrase devient un choix de conception et un test.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Combien de minutes d'indisponibilité un objectif de 99,95 % autorise-t-il sur un mois de 30 jours ?**

0,0005 x 43 200 = 21,6 minutes.

▼ **Deux équilibreurs de charge indépendants ont chacun une disponibilité de 99,5 %, et chacun peut supporter tout le trafic. Quelle est la disponibilité de la paire ?**

1 - (0.005 x 0.005) = 1 - 0.000025 = 99.9975%.

▼ **Pourquoi l'ajout d'un second serveur identique peut-il apporter bien moins que ce que prédit la formule du parallèle ?**

Parce que les copies peuvent partager des causes communes de défaillance, comme la même version, la même configuration, la même alimentation, le même réseau ou la même dépendance, et ne tombent donc pas en panne indépendamment ; le failover lui-même peut aussi être lent ou défectueux.

▼ **Les sauvegardes d'une équipe ont lieu toutes les 6 heures et une restauration complète prend 3 heures. Quels sont, dans le pire des cas, leurs RPO et RTO effectifs ?**

Un RPO pouvant aller jusqu'à 6 heures de données perdues, et un RTO d'au moins 3 heures, auquel s'ajoute le temps de détecter le sinistre et de décider de restaurer.

▼ **À vous : Une chaîne comprend la passerelle à 99,95 %, le service de commandes à 99,9 % et le paiement à 99,9 %. Quelle est sa disponibilité, et son indisponibilité par an ? Ajoutez ensuite une seconde instance de commandes, indépendante, avec un failover opérationnel. Recalculez et nommez l'hypothèse qui pourrait fausser le résultat.**

En série : 0,9995 x 0,999 x 0,999 = 0,99750, soit environ 99,75 %, ou 21,9 heures par an. Deux instances de commandes donnent 1 - 0,001 x 0,001 = 0,999999 ; la chaîne vaut donc 0,9995 x 0,999999 x 0,999 = 0,99850, soit environ 13,1 heures. Le paiement et la passerelle dominent désormais. L'hypothèse est l'indépendance : une mauvaise version partagée, une alimentation électrique ou une base de données commune font tomber les deux copies ensemble, et le basculement doit réellement fonctionner.

SOURCES

- Google SRE book: Embracing Risk (time-based and request-based availability): https://sre.google/sre-book/embracing-risk/
- Google SRE book: Service Level Objectives: https://sre.google/sre-book/service-level-objectives/
- SEI blog: Tactics and Patterns for Software Robustness (availability tactics): https://www.sei.cmu.edu/blog/tactics-and-patterns-for-software-robustness/
- AWS whitepaper: Disaster recovery options in the cloud: https://docs.aws.amazon.com/whitepapers/latest/disaster-recovery-workloads-on-aws/disaster-recovery-options-in-the-cloud.html
- Avižienis, Laprie, Randell and Landwehr: Basic Concepts and Taxonomy of Dependable and Secure Computing (IEEE TDSC, 2004): https://doi.org/10.1109/TDSC.2004.2

SECTION 2.6 SUR 2.8 · INTERMÉDIAIRE · 22 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Modifiabilité, testabilité et déployabilité : concevoir pour le changement

AVANT DE LIRE

Cette section s'appuie sur :

- 0.3 Organiser le code
- 0.8 Comment les logiciels sont construits
- 2.3 Scénarios d'attributs de qualité

Termes déjà vus qu'elle utilise : Feature flag (indicateur de fonctionnalité), Interface, Module, Test unitaire, Production

L'IDÉE EN BREF

Pensez à une cuisine. Si la cuisinière est encastrée dans le mur, avec les tuyaux noyés dans le béton, la remplacer oblige à démolir le mur. Si elle se branche sur une prise standard, vous l'échangez en un après-midi. Il en va de même pour les logiciels : certaines parties changeront à coup sûr, alors les architectes leur donnent des connexions standard, permettent de vérifier facilement que tout fonctionne encore après un échange, et font de la livraison de la nouvelle version une routine plutôt qu'un événement risqué.

CE QU'IL FAUT SAVOIR

- La modifiabilité est le coût et le risque du changement ; spécifiez-la en indiquant ce qui est susceptible de changer, qui effectue le changement, à quel moment il est lié et ce qu'il peut coûter.
- Les tactiques de modifiabilité augmentent la cohésion, réduisent le couplage (encapsuler, utiliser un intermédiaire, restreindre les dépendances) et diffèrent la liaison, mais chaque moment de liaison plus tardif ajoute de la mécanique.
- La testabilité dépend de la contrôlabilité et de l'observabilité ; isolez les dépendances derrière des interfaces et injectez-les pour que les tests puissent utiliser des doublures de test.
- La déployabilité repose sur de petits lots, l'automatisation, des unités déployables indépendamment, des changements rétrocompatibles, des déploiements progressifs et un retour arrière rapide.

- Les métriques actuelles de performance de livraison logicielle de DORA sont le délai de mise en œuvre des changements, la fréquence de déploiement, le temps de rétablissement après un déploiement en échec, le taux d'échec des changements et le taux de reprise des déploiements.
- Dans l'ISO/IEC 25010:2023, la maintenabilité compte cinq sous-caractéristiques : modularité, réutilisabilité, analysabilité, modifiabilité et testabilité.

Trois qualités liées au changement

L'essentiel du coût d'un système qui réussit est engagé après sa première mise en production, à mesure qu'il est modifié encore et encore. Trois qualités liées décrivent dans quelle mesure une architecture s'y prête. La modifiabilité concerne le coût et le risque d'un changement. La testabilité concerne la facilité avec laquelle vous pouvez montrer, par des tests, que le système (ou une partie de celui-ci) se comporte comme prévu, et la facilité avec laquelle les fautes se révèlent. La déployabilité concerne la rapidité, la sûreté et la prévisibilité avec lesquelles un changement peut être mis en production, puis retiré.

Elles forment une boucle. Un changement n'est bon marché que si vous pouvez le faire à un seul endroit (modifiabilité), savoir rapidement que vous n'avez rien cassé (testabilité) et le livrer sans cérémonie (déployabilité). Un système modulaire mais sans tests rapides évoluera quand même lentement, car chaque changement exige des semaines de tests de régression manuels. Un système doté de bons tests mais d'une fenêtre de mise en production trimestrielle livrera quand même de la valeur lentement.

Modifiabilité : quatre questions à poser sur tout changement

Bass, Clements et Kazman estiment que la modifiabilité n'a pas de sens tant que vous n'avez pas dit ce qui est censé changer. Prévoir tous les changements possibles produit un système surdimensionné, plein de points d'extension inutilisés ; n'en prévoir aucun produit un système où chaque changement est une réécriture. Quatre questions précisent l'exigence.

Qu'est-ce qui peut changer ? Les fonctions, les plateformes (une nouvelle version de base de données, le passage à un autre cloud), l'environnement (une nouvelle API partenaire, une nouvelle réglementation), les qualités (deux fois plus de charge) ou la capacité. **Quelle est la probabilité du changement ?** Consacrez l'effort de conception là où le changement est probable : les prestataires de paiement, les règles fiscales et les interfaces utilisateur changent souvent ; la notion de commande, rarement. **Qui fait le changement, et quand ?** Un développeur au moment du codage, un ingénieur de mise en production au déploiement, un administrateur par la configuration ou un utilisateur final par les paramètres. Le moment où

une décision est fixée s'appelle son moment de liaison (binding time). **Que coûte le changement ?** L'effort, le délai et l'argent nécessaires pour le réaliser, le tester et le déployer, plus le nombre d'autres éléments touchés et le risque d'introduire des défauts.

Répondre à ces questions donne un scénario du type présenté dans la section 2.3 : «Un développeur ajoute la prise en charge d'un nouveau prestataire de paiement, sans modifier aucun module existant autre que la configuration, en 3 jours-personne tests compris, sans augmentation des erreurs à l'étape de paiement.»

Scénarios de modifiabilité avec différentes sources et différents moments de liaison

Qui change	Ce qui change	Moment de liaison	Mesure de la réponse
Développeur	Ajouter un nouveau partenaire de livraison à une plateforme de livraison de repas	Codage	Seul l'adaptateur du partenaire change ; développé, testé et déployé en 5 jours
Chef de produit	Activer un nouveau parcours de paiement pour 10 % des utilisateurs	Exécution (feature flag)	Prend effet en moins de 1 minute, sans déploiement
Ingénieur d'exploitation	Faire pointer le système vers un nouvel hôte de base de données	Déploiement (configuration)	Aucune modification du code ; une exécution du pipeline de moins de 15 minutes
Administrateur côté client	Ajouter un champ personnalisé à une facture	Exécution (paramètres)	Sans intervention du fournisseur ; disponible immédiatement
Équipe plateforme	Passer le framework web à une nouvelle version majeure	Build	Moins de 20 fichiers modifiés hors d'une couche anticorruption ; moins de 2 semaines

Tactiques de modifiabilité

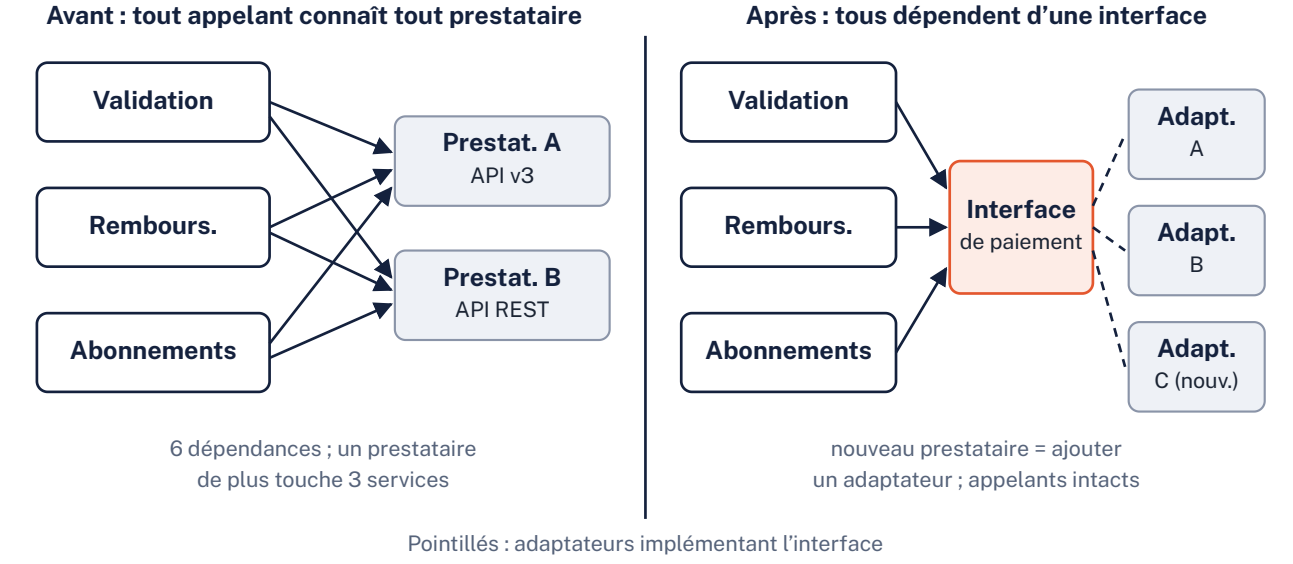
Les tactiques de *Software Architecture in Practice* réduisent le coût du changement en limitant la propagation d'un changement. Elles reposent sur deux idées anciennes, traitées en profondeur au chapitre 3. La cohésion mesure à quel point les responsabilités d'un module vont ensemble ; le couplage mesure à quel point un module dépend d'un autre. Un changement probable devrait toucher un seul module cohésif, et ce module devrait être faiblement couplé au reste.

Augmenter la cohésion. Scindez un module qui fait deux tâches sans rapport (un `CustomerService` qui gère à la fois la connexion et les points de fidélité) et déplacez les responsabilités là où elles ont leur place, pour qu'un changement probable n'atterrisse qu'à un seul endroit.

Réduire le couplage. *Encapsuler* : cachez l'intérieur d'un module derrière une interface explicite, selon le principe de masquage de l'information de David Parnas (1972), pour que les appelants ne puissent pas dépendre de ce qui peut changer. *Utiliser un intermédiaire* : placez quelque chose entre deux parties pour qu'aucune ne connaisse directement l'autre : un adaptateur, un courtier de messages (message broker), un canal publication-abonnement, un repository devant une base de données ou une API gateway. *Restreindre les dépendances* : décidez quels modules peuvent appeler lesquels (règles de couches, module visible uniquement dans son propre package) et faites-le respecter par l'outillage. *Abstraire les services communs* : fournissez une seule fois un comportement partagé, derrière une interface, plutôt que de le copier.

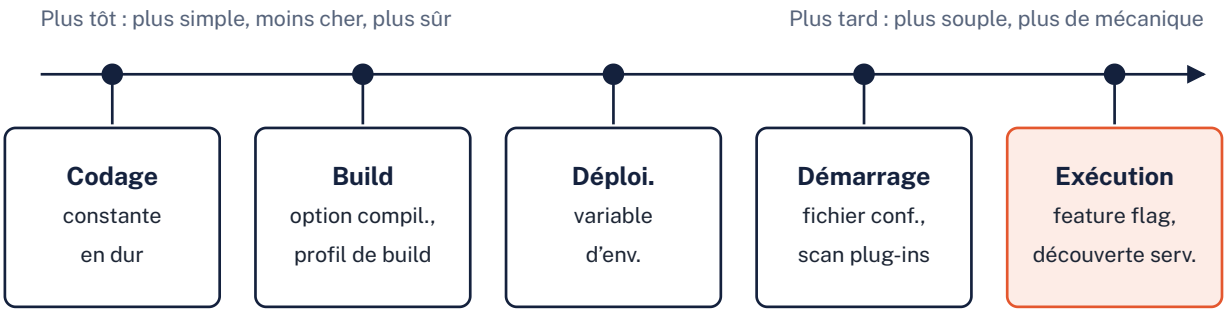
Différer la liaison. Fixez une décision aussi tard que c'est utile, pour que le changement puisse se faire plus tard et par quelqu'un de plus éloigné du code : fichiers de configuration, variables d'environnement, plug-ins découverts au démarrage, découverte de services et feature flags (indicateurs de fonctionnalité). Chaque moment de liaison plus tardif ajoute de la mécanique (configuration à valider, flags à supprimer, interfaces de plug-in versionnées) : ne différez donc que les décisions qui varient réellement.

Utiliser un intermédiaire pour contenir le changement



Une interface avec un adaptateur par prestataire transforme un changement qui touchait trois services en l'ajout d'un seul adaptateur.

Moment de liaison (binding time)



Chaque pas vers la droite permet de changer plus tard, par quelqu'un de plus loin du code, sans nouvelle version

Déplacer une décision vers la droite lui permet de changer plus tard et sans mise en production, mais chaque pas ajoute de la mécanique à gérer.

GUIDE DE DÉCISION

Guide de décision : à quel moment cette décision doit-elle être liée ?

Utilisez-le quand une partie prenante dit « cela pourrait changer ». Commencez en haut et arrêtez-vous au premier oui.

Q1 **Le changement est-il improbable, ou nécessiterait-il de toute façon du nouveau code et de nouveaux tests ?**

- **Oui** Liez la décision au moment du codage. Gardez-la simple et bien encapsulée pour que le changement reste local le jour où il arrive.
- **Non** Passez à la question suivante.

Q2 **La valeur ne diffère-t-elle qu'entre environnements (test, staging, production) ou entre installations de différents clients ?**

- **Oui** Liez-la au déploiement : variables d'environnement ou configuration injectées par le pipeline, conservées sous gestion de versions.
- **Non** Passez à la question suivante.

Q3 **Des tiers ou d'autres équipes doivent-ils ajouter du comportement sans modifier votre code ?**

- **Oui** Liez-la au démarrage ou à l'exécution avec une interface de plug-in ou d'extension, et versionnez cette interface avec soin.
- **Non** Passez à la question suivante.

Q4 **Des personnes qui ne sont pas développeurs doivent-elles activer ou désactiver un comportement rapidement, ou devez-vous d'abord livrer à une fraction des utilisateurs ?**

- **Oui** Liez-la à l'exécution avec un feature flag, et donnez à chaque flag un responsable et une date de suppression.
- **Non** Liez-la au build ou au déploiement ; une liaison plus tardive ajouterait une mécanique dont personne n'a besoin.

EXEMPLE COMMENTÉ

Exemple commenté : que vaut le scénario de modifiabilité ?

Une société de livraison de repas intègre les systèmes de point de vente (caisses) des restaurants. Aujourd'hui, chaque nouvel éditeur de point de vente exige des modifications dans les services de carte, de commandes et de remboursements : environ 15 jours-personne de travail, plus un cycle de tests de régression de 5 jours. Les commerciaux prévoient 8 nouveaux éditeurs au cours des 12 prochains mois. L'architecte propose une interface d'adaptateurs d'éditeurs (comme dans le schéma ci-dessus). On suppose un coût complet de £600 par jour-personne.

Coût par éditeur aujourd'hui	15 + 5 = 20 jours-personne, soit 20 x £600 = £12 000
Coût pour 8 éditeurs aujourd'hui	8 x £12,000 = £96,000
Coût ponctuel du refactoring	Concevoir et construire l'interface, et migrer les 3 éditeurs existants : 30 jours-personne = £18 000
Coût par éditeur ensuite	Écrire un adaptateur (3 jours), plus des tests de contrat sur l'interface (1 jour) : 4 jours-personne = £2400
Coût pour 8 éditeurs ensuite	£18,000 + 8 x £2,400 = £37,200
Économie la première année	£96,000 - £37,200 = £58,800
Seuil de rentabilité	Chaque éditeur fait économiser £12 000 - £2400 = £9600, donc les £18 000 sont remboursées après £18 000 / £9600 = 1,9, c'est-à-dire dès le deuxième nouvel éditeur

Le calcul ne vaut que ce que vaut la prévision du changement. Si les commerciaux ne signent aucun nouvel éditeur, le refactoring est un pur coût. C'est pourquoi les exigences de modifiabilité doivent dire ce qui est susceptible de changer.

Le délai compte souvent plus que le coût : ramener le lancement d'un éditeur d'environ 4 semaines à moins de 1 semaine peut faire gagner des contrats, ce qu'aucun calcul en jours-personne ne reflète.

La nouvelle interface d'adaptateurs est elle-même un engagement. La modifier plus tard touche tous les adaptateurs : elle mérite donc une conception et un versionnement soignés.

SI VOUS DÉBUTEZ

Pas à pas : décider quand un choix est fixé

L'idée la plus difficile de cette section est le moment de liaison (binding time) : le moment où une décision est fixée, qui détermine qui pourra la changer plus tard et avec quelle facilité. Suivez une décision à travers trois moments de liaison, à l'aide du schéma de la section, dans lequel l'étape de paiement atteint les prestataires de paiement via une seule interface de paiement, avec un adaptateur par prestataire.

- L'adaptateur A et l'adaptateur B relient chacun l'interface de paiement à un prestataire. Supposons que la décision porte sur l'adaptateur qu'utilise l'étape de paiement.
- Moment du codage : un développeur écrit « utiliser l'adaptateur A » dans le code. C'est le choix le plus simple, mais passer à B implique une modification du code, des tests et une nouvelle version, et seul un développeur peut le faire.
- Moment du déploiement : le choix passe dans la configuration, par exemple une variable d'environnement fournie par le pipeline de déploiement. Le test et la production peuvent désormais utiliser des prestataires différents, et un ingénieur d'exploitation peut changer de prestataire en une seule exécution du pipeline, sans modifier le code.
- Moment de l'exécution : un feature flag fait passer 10 % des paiements par l'adaptateur B, et un chef de produit peut augmenter ou réduire cette part en moins d'une minute, sans déploiement.
- Chaque étape plus tardive ajoute de la mécanique. La configuration doit être vérifiée pour qu'une valeur mal saisie ne puisse pas bloquer les paiements ; le flag a besoin d'un responsable et d'une date de suppression ; et tant que l'un ou l'autre adaptateur peut être actif, les deux doivent rester testés.
- Ajouter un nouvel adaptateur C implique toujours du nouveau code et de nouveaux tests, quel que soit le moment de liaison. Différer la liaison change le moment où un choix entre des options existantes est fait, et la personne qui le fait.
- **La leçon :** demandez-vous qui doit pouvoir changer une décision et à quelle fréquence, puis choisissez le moment de liaison le plus précoce qui le lui permet. Toute liaison plus tardive ne fait qu'ajouter une mécanique que personne n'utilise.

Testabilité : contrôlabilité et observabilité

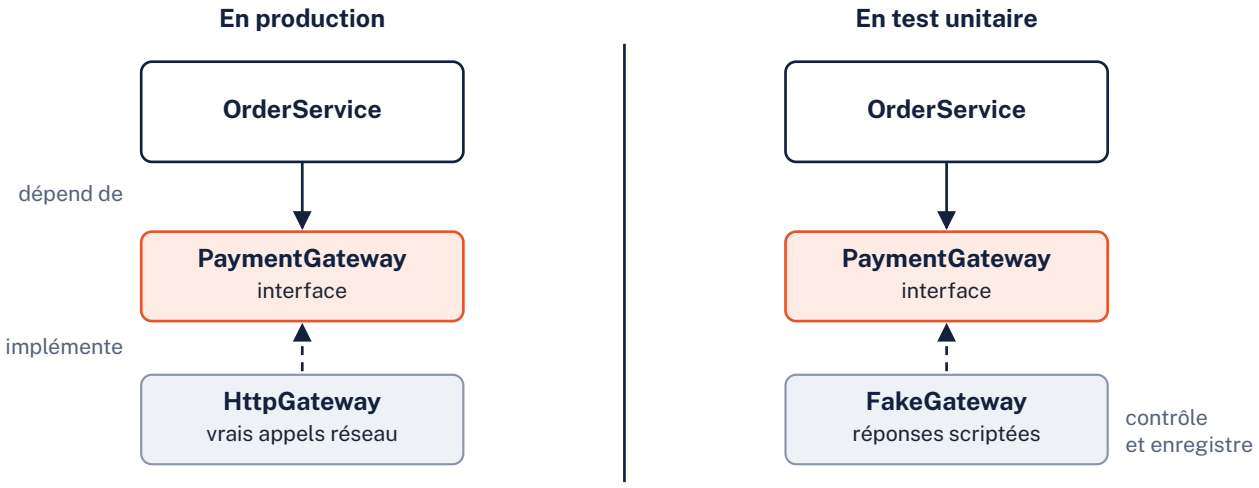
Un composant est testable lorsqu'un test peut le mettre dans un état connu, lui fournir des entrées choisies et voir ce qu'il a fait. Deux propriétés le traduisent. La **contrôlabilité** est la capacité à définir depuis un test les entrées et l'état interne d'un composant. L'**observabilité** (au sens des tests) est la capacité à voir ses sorties et son état interne pertinent. Un code qui lit

directement l'horloge système, appelle un vrai service de paiement, tire des nombres aléatoires ou écrit dans une base de données partagée est difficile à contrôler ; un code qui cache ses résultats dans un état privé ou uniquement dans le texte des journaux est difficile à observer.

Les tactiques architecturales de testabilité visent donc à donner aux tests contrôle et visibilité, et à limiter la complexité. **Isolez les dépendances** derrière des interfaces, puis utilisez l'injection de dépendances : le composant reçoit ses collaborateurs (une horloge, une passerelle de paiement, un repository) de l'extérieur au lieu de les créer, de sorte qu'un test peut lui passer un substitut. Ces substituts sont des doublures de test (test doubles), un terme popularisé par Gerard Meszaros : un *stub* renvoie des réponses préparées à l'avance, un *fake* est une implémentation légère qui fonctionne (un repository en mémoire), un *mock* vérifie que les appels attendus ont bien été faits, et un *spy* enregistre les appels pour des assertions ultérieures. **Fournissez des interfaces de test** : des points d'accroche pour définir l'état, enregistrer et rejouer du trafic, ou s'exécuter dans un environnement sandbox. **Limitez le non-déterminisme** : évitez la concurrence cachée et l'état global, et rendez le temps et l'aléatoire injectables. **Limitez la complexité structurelle** : moins de dépendances et aucun cycle signifient des mises en place de tests plus petites.

L'architecture façonne aussi la stratégie de test aux niveaux supérieurs. Dans un système de services, chaque frontière a besoin de tests de contrat qui vérifient que les deux côtés s'accordent sur l'API, et de quelques tests de bout en bout qui exercent le vrai chemin. Une conception aux frontières nettes permet à la plupart des tests de s'exécuter vite et isolément, tandis qu'une conception enchevêtrée impose des tests de bout en bout lents et fragiles pour tout.

L'injection de dépendances rend un composant contrôlable



Le même OrderService fonctionne avec la vraie passerelle en production et avec un fake scripté en test, parce qu'il ne dépend que de l'interface.

```
class OrderService:
    def __init__(self, gateway, clock):
        self.gateway = gateway   # anything with charge(amount, card)
        self.clock = clock       # anything with now()

    def place(self, order):
        if self.clock.now().hour ⩾ 23:
            return "kitchen closed"
        return self.gateway.charge(order.total, order.card)

# In a test: full control, no network, no waiting until 11 pm
class FakeClock:
    def __init__(self, t): self.t = t
    def now(self): return self.t

class FakeGateway:
    def __init__(self): self.calls = []
    def charge(self, amount, card):
        self.calls.append(amount)
        return "approved"
```

Injection par le constructeur : le service reçoit son horloge et sa passerelle, si bien qu'un test peut fournir des fakes.

Déployabilité

La déployabilité mesure avec quelle prévisibilité et à quel coût un changement passe du commit d'un développeur à la production, et à quelle vitesse il peut être annulé. Elle n'est devenue une qualité de premier plan qu'au cours de la dernière décennie, lorsque la livraison continue a montré que fréquence des mises en production et stabilité ne s'opposent pas. La quatrième édition de *Software Architecture in Practice* (2021) lui a ajouté un chapitre.

L'idée centrale est celle des **petits lots**. Une mise en production qui contient un seul changement est facile à tester, facile à comprendre et facile à annuler ; une mise en production de trois mois de travail contient des centaines de changements qui interagissent, si bien que les défaillances sont probables et difficiles à diagnostiquer. Les petits lots exigent de l'**automatisation** : un pipeline de déploiement qui construit, teste et déploie chaque changement de la même façon, une infrastructure scriptée et aucune étape manuelle que quelqu'un pourrait oublier. Les tactiques architecturales comprennent des unités déployables indépendamment (pour que la mise en production d'une équipe n'attende pas celle d'une autre), des interfaces et des migrations de base de données rétrocompatibles, des mécanismes de retour arrière (rollback) et de correction vers l'avant (roll forward), des déploiements progressifs (canary releases vers une petite part du trafic d'abord, bascule bleu-vert entre deux

environnements) et des feature flags, qui séparent le fait de *déployer* du code de celui de *mettre à disposition* une fonctionnalité pour les utilisateurs. Les pipelines et les stratégies sont traités dans pipelines CI/CD et stratégies de déploiement.

Ignorer la déployabilité peut coûter très cher. En août 2012, la société de trading Knight Capital a déployé manuellement du nouveau code sur ses huit serveurs de routage d'ordres, mais en a oublié un. Un flag de configuration réutilisé a alors activé du code obsolète sur ce serveur, qui a inondé le marché d'ordres, produisant environ 4 millions d'exécutions en quelque 45 minutes, et a coûté à l'entreprise environ 460 millions de dollars américains. Un déploiement automatisé et vérifié, tout comme la suppression du code mort, aurait suffi à l'éviter.

Mesurer la livraison : les métriques DORA

Le programme de recherche DORA (DevOps Research and Assessment, décrit dans *Accelerate* de Nicole Forsgren, Jez Humble et Gene Kim, 2018, et poursuivi dans des rapports annuels désormais réalisés au sein de Google Cloud) a montré qu'un petit ensemble de mesures prédit la performance de livraison logicielle, et que les meilleurs sont à la fois rapides et stables. Le modèle actuel compte cinq métriques de performance de livraison logicielle, regroupées en débit et instabilité. Leurs noms ont évolué : l'ancien «temps moyen de rétablissement» (mean time to restore) est devenu le temps de rétablissement après un déploiement en échec en 2023, et le taux de reprise des déploiements a été ajouté en 2024.

Deux mises en garde. D'abord, ces métriques décrivent le système de livraison d'une équipe, pas des individus ; s'en servir pour classer les personnes incite à les manipuler. Ensuite, ce sont des résultats. L'architecture les influence fortement : les équipes qui peuvent déployer leur partie indépendamment, la tester isolément et l'annuler rapidement obtiennent de meilleurs scores, ce qui explique en partie pourquoi la recherche DORA associe une architecture faiblement couplée à une meilleure performance de livraison.

Métriques DORA de performance de livraison logicielle (dora.dev, modèle actuel)

Métrique	Groupe	Ce qu'elle mesure	Leviers architecturaux
Délai de mise en œuvre des changements	Débit (throughput)	Temps écoulé entre le commit d'un changement dans la gestion de versions et son exécution en production	Tests automatisés rapides, petites unités déployables indépendamment, aucun passage de relais
Fréquence de déploiement	Débit (throughput)	Fréquence à laquelle les changements sont déployés en production (ou temps entre deux déploiements)	Déployabilité indépendante, pipeline automatisé, feature flags
Temps de rétablissement après un déploiement en échec	Débit (throughput)	Temps nécessaire pour se rétablir d'un déploiement qui échoue et exige une intervention immédiate	Retour arrière automatisé, canary releases, bonne supervision, petits changements
Taux d'échec des changements	Instabilité	Proportion des déploiements qui exigent une intervention immédiate, comme un retour arrière ou un correctif urgent (hotfix)	Testabilité, tests de contrat, changements rétrocompatibles
Taux de reprise des déploiements	Instabilité	Proportion des déploiements non planifiés, effectués à cause d'un incident en production	Qualité des tests et des revues ; observabilité qui repère tôt les problèmes

La maintenabilité dans l'ISO/IEC 25010

L'ISO/IEC 25010:2023 regroupe ces préoccupations sous la caractéristique de maintenabilité, avec cinq sous-caractéristiques : la **modularité** (le système est fait de composants distincts, si bien que la modification de l'un a un impact minimal sur les autres), la **réutilisabilité** (un actif peut être utilisé dans plusieurs systèmes ou pour construire d'autres actifs), l'**analysabilité** (la facilité avec laquelle on peut évaluer l'impact d'un changement, diagnostiquer une déficience ou trouver les parties à modifier), la **modifiabilité** (le système peut être modifié sans introduire de défauts ni dégrader la qualité) et la **testabilité** (des critères de test peuvent être établis et des tests exécutés pour déterminer s'ils sont respectés). La déployabilité n'a pas de sous-caractéristique propre dans la norme ; l'installabilité et la remplaçabilité relèvent de la flexibilité. Le modèle est une liste de contrôle, comme l'explique la section 2.2 ; ce sont les scénarios qui rendent chaque élément concret.

ATTENTION

Erreurs courantes

- Construire pour un changement imaginaire : frameworks de plug-ins et couches d'abstraction pour des variantes que personne n'a demandées. Chaque point d'extension a un coût en complexité et en tests. Demandez-vous quelle est la probabilité du changement avant de différer la liaison.
- Des feature flags sans porte de sortie : des centaines de flags périmés multiplient les chemins dans le code que personne ne teste. Donnez à chaque flag un responsable et une date de suppression.
- Les microservices comme remède à une livraison lente : découper un système en services qui doivent encore être mis en production ensemble (un monolithe distribué) dégrade la déployabilité au lieu de l'améliorer.
- Des tests chargés de mocks qui vérifient des détails d'implémentation : ils passent alors que le système est cassé et échouent à chaque refactoring du code. Préférez les fakes et les tests de contrat aux frontières architecturales.
- Mesurer le délai à partir de l'ouverture d'un ticket dans une équipe et à partir du commit dans une autre, puis comparer les deux. Définissez chaque métrique une seule fois et appliquez-la de façon cohérente.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Pourquoi « le système doit être facile à modifier » n'est-il pas une exigence utile ?**

Elle ne dit ni ce qui changera, ni qui le changera, ni quand, ni quel coût est acceptable : personne ne peut donc concevoir en conséquence ni vérifier par des tests qu'elle est satisfaite.

▼ **Quelle est la différence entre un stub et un mock ?**

Un stub renvoie des réponses préparées à l'avance pour que le test puisse contrôler les entrées ; un mock vérifie en plus que les appels attendus ont été faits, si bien que le test vérifie les interactions.

▼ **Pourquoi les feature flags favorisent-ils la déployabilité ?**

Ils séparent le déploiement du code de la mise à disposition d'une fonctionnalité : des fonctionnalités incomplètes ou risquées peuvent ainsi être déployées désactivées, activées progressivement et désactivées à nouveau sans nouveau déploiement.

▼ **Une équipe déploie tous les mois et chaque mise en production contient environ 200 changements. Quelles métriques DORA risquent d'en souffrir, et pourquoi ?**

Le délai de mise en œuvre des changements (les changements attendent la mise en production pendant des semaines), la fréquence de déploiement, et probablement le taux d'échec des changements et le temps de rétablissement, car les gros lots échouent plus souvent et sont plus difficiles à diagnostiquer et à annuler.

▼ **À vous : Chaque nouveau partenaire de livraison coûte aujourd'hui 10 jours-personne. Un refactoring de 15 jours ramènerait ce chiffre à 4 jours. Cinq partenaires sont attendus cette année, à £600 par jour. Le refactoring en vaut-il la peine, et quand est-il rentabilisé ?**

Sans : 5 x 10 = 50 jours, £30 000. Avec : 15 + (5 x 4) = 35 jours, £21 000, soit une économie de 15 jours ou £9000. Chaque partenaire fait gagner 6 jours : le seuil de rentabilité est donc à 15 / 6 = 2,5, c'est-à-dire au troisième partenaire. Cela vaut la peine si au moins trois partenaires semblent probables. Cela rend aussi chaque intégration moins risquée, mais vérifiez d'abord la prévision, car l'économie en dépend.

SOURCES

- DORA: software delivery performance metrics: https://dora.dev/guides/dora-metrics/
- DORA: history of the software delivery metrics: https://dora.dev/guides/dora-metrics/history/
- Martin Fowler: Test Double: https://martinfowler.com/bliki/TestDouble.html
- Pete Hodgson: Feature Toggles (aka Feature Flags), martinfowler.com: https://martinfowler.com/articles/feature-toggles.html
- Martin Fowler: Inversion of Control Containers and the Dependency Injection pattern: https://martinfowler.com/articles/injection.html

Sécurité, utilisabilité, interopérabilité, coût et durabilité en tant que qualités

AVANT DE LIRE

Cette section s'appuie sur :

- 2.2 Modèles de qualité
- 2.3 Scénarios d'attributs de qualité
- 2.5 Disponibilité et fiabilité

Termes déjà vus qu'elle utilise : ISO/IEC 25010, Scénario d'attribut de qualité, Tactique, Contrainte, API (interface de programmation d'application)

L'IDÉE EN BREF

Quand vous choisissez une banque, vous tenez à ce qu'elle protège votre argent, que son app soit facile à utiliser, qu'elle fonctionne avec les autres banques, que ses frais soient raisonnables et, peut-être, qu'elle ne gaspille pas d'énergie. Aucune de ces qualités n'est une fonctionnalité que l'on peut montrer du doigt à l'écran ; chacune est une qualité du service dans son ensemble. Les architectes doivent transformer chacune d'elles en quelque chose d'assez précis pour concevoir en conséquence et le vérifier, et ils doivent souvent les mettre en balance les unes avec les autres.

CE QU'IL FAUT SAVOIR

- La sécurité part de la confidentialité, de l'intégrité et de la disponibilité ; l'ISO/IEC 25010:2023 énumère la confidentialité, l'intégrité, la non-répudiation, l'imputabilité, l'authenticité et la résistance, la disponibilité étant rangée sous la fiabilité.
- Les scénarios de sécurité nomment l'attaquant, l'attaque, la cible et une réponse mesurable, comme le délai de détection ou les données exposées ; les tactiques consistent à résister aux attaques, à les détecter, à y réagir et à s'en remettre.
- La capacité d'interaction (anciennement utilisabilité) influe sur l'architecture par la réactivité, l'annulation, l'interruption d'une opération et le fonctionnement hors ligne ; l'accessibilité selon les WCAG 2.2, niveau AA, est de plus en plus une contrainte légale.
- L'interopérabilité exige un accord sur le sens autant que sur le format ; les normes sectorielles et les adaptateurs en réduisent le coût.
- Le coût et le carbone s'expriment de préférence par unité de travail ; la spécification SCI (ISO/IEC 21031:2024) calcule ((E x I) + M) par R et ne compte que les réductions réelles.

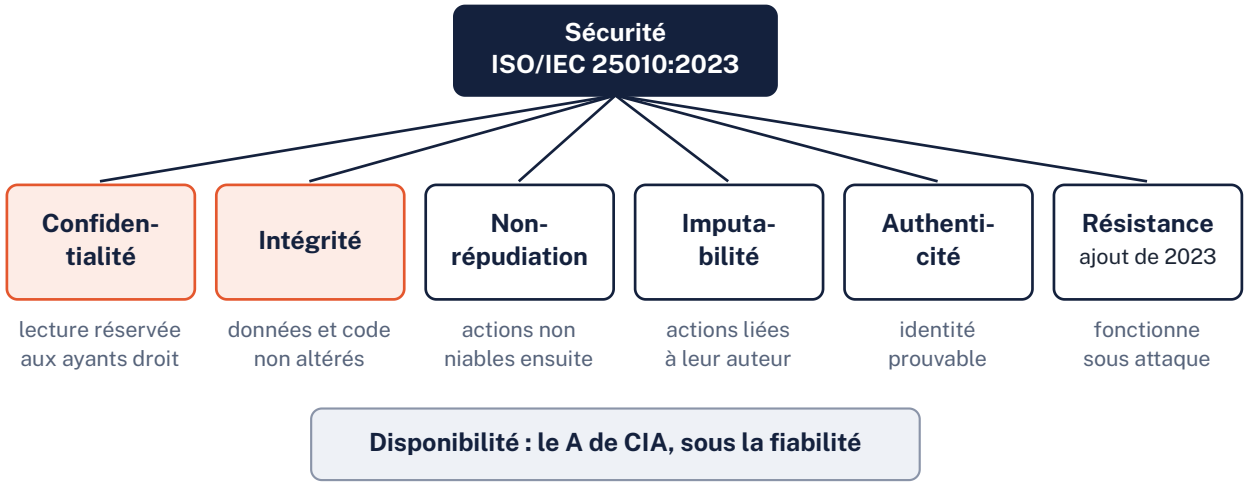
- La sûreté, l'observabilité, la résidence des données, l'auditabilité et la conservation sont aussi des qualités, et doivent être recueillies tôt sous forme de contraintes testables.

La sécurité en tant qu'attribut de qualité

La sécurité est la capacité du système à protéger les données et les fonctions contre l'accès et l'utilisation non autorisés, tout en continuant à servir les personnes autorisées. Les architectes partent souvent de la triade CIA : la **confidentialité** (seules les parties autorisées peuvent lire les données), l'**intégrité** (les données et le code ne sont pas modifiés sans autorisation, et toute modification est détectable) et la **disponibilité** (les utilisateurs autorisés peuvent utiliser le service quand ils en ont besoin). Les attaques se rattachent à ces trois propriétés : voler des fiches clients porte atteinte à la confidentialité, modifier les coordonnées bancaires d'un bénéficiaire porte atteinte à l'intégrité, et une attaque par déni de service porte atteinte à la disponibilité.

L'ISO/IEC 25010:2023 affine ce tableau. Sa caractéristique de sécurité comporte six sous-caractéristiques : la confidentialité, l'intégrité, la **non-répudiation** (on peut prouver que des actions ou des événements ont eu lieu, de sorte qu'une partie ne puisse pas les nier ensuite), l'**imputabilité** (les actions peuvent être attribuées de manière univoque à l'entité qui les a effectuées), l'**authenticité** (on peut prouver que l'identité d'un sujet ou d'une ressource est bien celle qui est revendiquée) et, nouveauté de l'édition 2023, la **résistance** (le produit continue de fonctionner comme prévu lorsqu'il est attaqué par un acteur malveillant). Dans ce modèle, la disponibilité relève de la fiabilité, c'est pourquoi elle a été traitée à la section 2.5.

Les sous-caractéristiques de la sécurité



La confidentialité et l'intégrité viennent de la triade CIA ; les quatre autres rendent explicites l'imputabilité, la preuve d'identité et le comportement sous attaque.

Rédiger des scénarios de sécurité

« Le système doit être sécurisé » ne permet ni de concevoir ni de tester. Un scénario de sécurité nomme une source (qui attaque, avec quel accès), un stimulus (ce qu'elle tente), un artefact (ce qui est visé), un environnement (en ligne, pendant un incident, avec l'attaquant déjà à l'intérieur du réseau), une réponse et une mesure. En sécurité, les mesures portent souvent sur le délai de détection, l'ampleur des dégâts ou le délai de rétablissement, car « aucune attaque ne réussit jamais » est invérifiable.

Trois exemples pour une banque en ligne. *Bourrage d'identifiants (credential stuffing) :* un attaquant externe essaie des paires nom d'utilisateur et mot de passe ayant fuité contre l'API de connexion, à raison de 50 tentatives par seconde depuis de nombreuses adresses ; le système détecte le schéma en moins de 2 minutes, soumet les adresses sources à une vérification supplémentaire ou les bloque, et l'attaquant ne prend le contrôle d'aucun compte protégé par l'authentification multifacteur. *Falsification interne :* un employé du support disposant d'un accès légitime modifie les coordonnées d'un bénéficiaire d'un client ; la modification requiert l'approbation d'une deuxième personne, et le journal d'audit enregistre qui, quoi et quand sous une forme que l'employé ne peut pas modifier. *Sauvegarde volée :* un attaquant obtient une copie de la sauvegarde de la base de données de la semaine dernière ; toutes les données personnelles qu'elle contient sont chiffrées avec des clés stockées séparément, si bien qu'aucune donnée client n'est lisible.

Notez que chaque scénario implique une architecture différente : limitation de débit (rate limiting) et détection d'anomalies en périphérie, circuit d'approbation à quatre yeux et journaux révélant toute altération, et chiffrement avec une gestion des clés séparée. Trouver ces scénarios de manière systématique est le rôle de la modélisation des menaces, traitée au chapitre 8.

SI VOUS DÉBUTEZ

Pas à pas : distinguer les termes de la sécurité

L'idée qui demande le plus d'attention ici consiste à distinguer les sous-caractéristiques de la sécurité qui se ressemblent, en particulier l'authenticité, l'imputabilité et la non-répudiation. Reprenez le scénario de falsification interne ci-dessus, dans lequel un employé du support modifie les coordonnées d'un bénéficiaire d'un client, et associez chacune de ses protections à une sous-caractéristique.

- Les coordonnées du bénéficiaire ne doivent pas changer sans autorisation, et toute modification doit être détectable ; dans le scénario, une deuxième personne doit approuver la modification. Cela protège l'**intégrité**.
- Le journal d'audit enregistre qui a fait la modification, en quoi elle consistait et quand, si bien qu'on peut la rattacher à cet employé précis. C'est l'**imputabilité** : savoir qui l'a fait.
- Le journal ne peut désigner la bonne personne que si la banque est sûre que la personne connectée sous l'identité de l'employé est bien cet employé, et non quelqu'un qui utilise un mot de passe volé. Pouvoir prouver qu'une personne ou un système est bien celui ou ce qu'il prétend être, c'est l'**authenticité**.
- Le journal est conservé sous une forme que l'employé ne peut pas modifier : s'il affirme plus tard « ce n'était pas moi », la banque peut prouver le contraire. C'est la **non-répudiation** : la preuve que l'action a eu lieu, si bien qu'elle ne peut pas être niée.
- Les deux autres relèvent d'autres scénarios. La **confidentialité** (seules les parties autorisées peuvent lire les données) est ce que le scénario de la sauvegarde volée protège par le chiffrement. La **résistance**, le fait de continuer à fonctionner comme prévu sous attaque, est ce que met à l'épreuve le scénario du bourrage d'identifiants.
- **La leçon :** la confidentialité et l'intégrité protègent les données ; l'authenticité, l'imputabilité et la non-répudiation concernent les personnes qui agissent sur elles : prouver qui elles sont, retracer ce qu'elles ont fait et le prouver ensuite. Chacune appelle un mécanisme différent : nommez donc celle que vous visez dans chaque scénario.

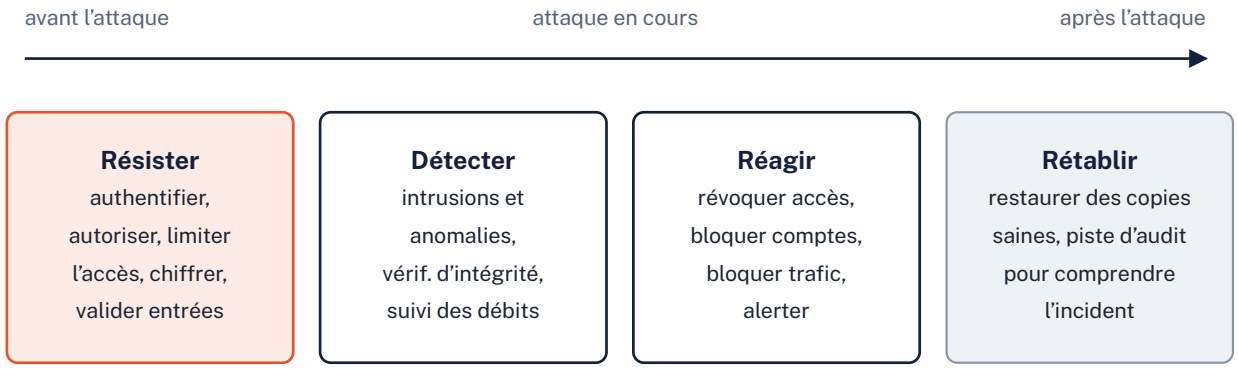
Tactiques de sécurité : vue d'ensemble

Bass, Clements et Kazman regroupent les tactiques de sécurité selon le moment où elles agissent. **Résister aux attaques** : identifier, authentifier et autoriser les acteurs, limiter l'accès et l'exposition (moins de ports ouverts, moins d'endpoints publics, moindre privilège), chiffrer les données en transit et au repos, séparer les entités (données sensibles dans leur propre stockage ou segment de réseau), modifier les paramètres par défaut et valider les entrées. **Détecter les attaques** : détection d'intrusion, détection du déni de service, vérification de l'intégrité des messages et repérage des délais ou des schémas inhabituels. **Réagir aux**

attaques : révoquer l'accès, verrouiller les comptes après des échecs de connexion, informer les opérateurs et les utilisateurs. **Se rétablir après une attaque** : restaurer un état sain connu (à nouveau les tactiques de disponibilité) et conserver une piste d'audit pour comprendre ce qui s'est passé et étayer la non-répudiation.

Cette section ne fait que présenter ces familles. Les principes comme la défense en profondeur et le moindre privilège, les protocoles d'identité, le chiffrement, le zero trust et la sécurité de la chaîne d'approvisionnement sont l'objet du chapitre 8. Le point architectural ici est que la sécurité est une propriété de toute la structure, et non un composant que l'on ajoute à la fin : l'emplacement des frontières de confiance, le composant qui détient telle ou telle donnée et la manière dont les services s'authentifient mutuellement sont des décisions précoces et coûteuses à changer.

Tactiques avant, pendant et après une attaque



Regroupement de Bass, Clements et Kazman ; détails au chapitre 8

Les tactiques de résistance réduisent les chances de succès ; les tactiques de détection, de réaction et de rétablissement limitent les dégâts quand la résistance échoue.

Utilisabilité, capacité d'interaction et accessibilité

L'ISO/IEC 25010:2023 a renommé l'utilisabilité en *capacité d'interaction* : le degré auquel des utilisateurs spécifiés peuvent interagir avec un produit pour échanger des informations via l'interface utilisateur afin d'accomplir des tâches spécifiques. Ses sous-caractéristiques sont la reconnaissabilité de l'adéquation, la facilité d'apprentissage, l'opérabilité, la protection contre les erreurs de l'utilisateur, l'engagement de l'utilisateur (qui a remplacé l'esthétique de l'interface utilisateur), l'inclusivité, l'assistance à l'utilisateur et l'autodescription. Une grande partie de tout cela relève du design d'interaction plutôt que de l'architecture, mais plusieurs préoccupations d'utilisabilité touchent profondément la structure.

La réactivité est la plus évidente. Des recommandations de longue date en ergonomie, popularisées par Jakob Nielsen, fixent trois limites : environ 0,1 seconde paraît instantané, environ 1 seconde préserve le fil de la pensée de l'utilisateur, et environ 10 secondes

constituent la limite de l'attention. Les respecter peut exiger du cache, un traitement asynchrone avec indication de progression, ou des mises à jour optimistes. Annuler et interrompre sont des questions d'architecture : prendre en charge «annuler les cinq dernières actions» exige des commandes réversibles, et interrompre un long envoi de fichier exige une opération qu'on peut arrêter proprement. Le fonctionnement hors ligne d'une app mobile exige un stockage local et une synchronisation avec gestion des conflits. Il est bien moins coûteux de prévoir tout cela dès la conception que de l'ajouter après coup.

L'accessibilité, c'est-à-dire la possibilité pour les personnes handicapées d'utiliser le produit, est de plus en plus une contrainte légale plutôt qu'une préférence. Les Web Content Accessibility Guidelines version 2.2 (WCAG 2.2), recommandation du W3C depuis octobre 2023, définissent des critères de succès testables aux niveaux A, AA et AAA ; le niveau AA est la référence habituelle sur le plan légal et dans les marchés publics. Les WCAG 2.2 ont ajouté neuf critères, dont des tailles minimales de cible pour les dispositifs de pointage et l'authentification accessible (aucune étape de la connexion ne peut dépendre d'un test de fonction cognitive, comme mémoriser un mot de passe ou recopier un code, sauf si l'utilisateur se voit proposer une autre méthode ou une aide, comme le collage depuis un gestionnaire de mots de passe ; quelques autres exceptions limitées s'appliquent). Dans l'Union européenne, l'acte européen sur l'accessibilité (directive (UE) 2019/882) s'applique depuis le 28 juin 2025 à de nombreux produits et services destinés aux consommateurs, dont le commerce électronique, les services bancaires, les livres numériques et les services de transport de voyageurs. Pour un architecte, l'accessibilité est donc une contrainte à recueillir tôt : elle influe sur le choix du framework frontend, le design system, le parcours d'authentification et le pipeline de test (vérifications automatisées de l'accessibilité en intégration continue, plus des tests manuels avec des technologies d'assistance).

Interopérabilité

L'interopérabilité est le degré auquel deux systèmes ou plus peuvent échanger des informations et utiliser les informations échangées. L'ISO/IEC 25010 la range sous la compatibilité, à côté de la coexistence. Elle comporte deux niveaux. L'interopérabilité *syntaxique* signifie que les systèmes s'accordent sur le format et le protocole : tous deux parlent HTTP et JSON. L'interopérabilité *sémantique* signifie qu'ils s'accordent sur le sens : `"amount": 1050` est exprimé en pence, pas en livres ; `status: "closed"` signifie payé, pas annulé. La plupart des échecs d'intégration sont d'ordre sémantique.

Les secteurs d'activité réduisent le coût de l'accord sémantique grâce à des normes communes : HL7 FHIR pour l'échange de dossiers de santé, ISO 20022 pour les messages financiers, GTFS pour les horaires de transports publics. Les tactiques architecturales comprennent la découverte de services via un registre, l'adaptation des interfaces au moyen d'adaptateurs et de traducteurs, et une gestion explicite des versions pour que les partenaires

plus anciens continuent de fonctionner. Un scénario d'interopérabilité pourrait s'énoncer ainsi : «Un nouvel hôpital rejoint le réseau et échange des résumés de dossiers patients à l'aide de ressources FHIR R4 en moins de 4 semaines, sans modification du service central de dossiers.»

Le coût en tant que qualité

Le coût figure rarement dans les modèles de qualité, et pourtant il contraint chaque décision d'architecture. Distinguez le **coût de construction** (l'effort nécessaire pour créer le système), le **coût d'exploitation** (infrastructure, licences, services tiers, support) et le **coût du changement** (l'effort nécessaire pour chaque modification future, ce dont traite la modifiabilité). Dans le cloud, le coût d'exploitation est variable et directement façonné par l'architecture : des services trop bavards paient le transfert réseau, des clusters surdimensionnés paient de la capacité inutilisée, et un schéma de requêtes négligent peut multiplier la facture de base de données.

Une manière utile de faire du coût un attribut de qualité est le coût unitaire : coût par transaction, par utilisateur actif ou par commande. Contrairement à une facture mensuelle totale, il montre si l'architecture devient plus ou moins efficiente à mesure que l'activité croît, et il peut s'écrire sous forme de scénario : «À 3 millions de commandes par mois, le coût d'infrastructure reste inférieur à 1 penny par commande.» Les pratiques FinOps pour le gérer sont traitées au chapitre 10.

EXEMPLE COMMENTÉ

Exemple commenté : coût unitaire et carbone par requête pour une API de livraison de repas

Une API de commandes tourne sur un cluster qui coûte 24 000 £ par mois et traite 3 000 000 de commandes par mois. Une analyse estime sa consommation d'énergie à 1200 kWh par mois, sur un réseau électrique d'une moyenne de 250 g CO_2e par kWh, avec 60 kg CO_2e d'émissions incorporées du matériel qui lui sont imputées chaque mois. La formule de la Software Carbon Intensity est SCI = ((E x I) + M) par R.

Coût par commande	24 000 £ / 3 000 000 = 0,008 £, soit 0,8 pence par commande
Émissions opérationnelles (E x I)	1200 kWh x 250 g = 300 000 g = 300 kg CO_2e
Ajouter les émissions incorporées (M)	300 + 60 = 360 kg CO_2e
Diviser par l'unité fonctionnelle (R = une commande)	360 000 g / 3 000 000 = 0,12 g CO_2e par commande
Changement : mettre les cartes en cache, réduire l'énergie de 25 %	E = 900 kWh : (900 x 250 + 60 000) / 3 000 000 = 0,095 g par commande
Changement : exécuter les traitements par lots dans une région à 50 g par kWh	Réduit I pour cette part de la charge ; une planification consciente du carbone (carbon-aware) réduit le score sans modifier le code
Croissance à 6 000 000 de commandes sur le même cluster	Le coût par commande est divisé par deux, à 0,4 pence ; le carbone par commande baisse aussi, tant que l'énergie augmente moins que proportionnellement (les serveurs inactifs consomment quand même de l'électricité, donc un taux d'utilisation plus élevé est plus efficient)

Les deux indicateurs sont des taux par unité de travail utile : ils récompensent donc l'efficience au lieu de pénaliser la croissance.

La spécification SCI ne compte que les réductions réelles : acheter des compensations carbone ou des certificats d'énergie renouvelable ne fait pas baisser le score.

Les chiffres d'énergie et de réseau électrique sont des estimations ; l'important est de les mesurer de la même manière dans le temps pour pouvoir comparer les changements.

Efficacité énergétique et durabilité

Le logiciel n'émet pas de carbone lui-même, mais le matériel sur lequel il tourne consomme de l'électricité et porte en lui les émissions de sa fabrication. La spécification Software Carbon Intensity (SCI) de la Green Software Foundation, publiée par l'ISO sous la référence ISO/IEC 21031:2024, définit un taux : les émissions opérationnelles (l'énergie E multipliée par l'intensité carbone I de l'électricité) plus les émissions incorporées M, divisées par une unité fonctionnelle R, comme un utilisateur, un appel d'API ou une commande. Elle identifie trois leviers : l'**efficacité énergétique** (consommer moins d'électricité pour le même travail), l'**efficacité matérielle** (utiliser moins d'appareils, et les utiliser plus pleinement et plus longtemps) et la **conscience carbone** (exécuter le travail flexible quand et là où l'électricité est plus propre).

L'architecture influe sur les trois. Le bon dimensionnement et un taux d'utilisation plus élevé améliorent l'efficacité matérielle ; le cache, des formats de données efficaces et l'absence de travail inutile améliorent l'efficacité énergétique ; les files d'attente et le traitement par lots permettent de décaler le travail dans le temps ou dans l'espace. La durabilité va souvent dans le même sens que le coût, mais pas toujours : maintenir un warm standby (secours tiède) dans une deuxième région améliore la disponibilité et double les émissions incorporées. Les grands modèles d'IA rendent la question plus pressante, comme l'explique le chapitre 12.

Sûreté (safety)

La sûreté est l'absence de risque inacceptable de dommage aux personnes, aux biens ou à l'environnement. Elle diffère de la sécurité : la sécurité protège le système contre des personnes mal intentionnées ; la sûreté protège les personnes contre le système, y compris quand personne n'est malveillant. L'ISO/IEC 25010:2023 a ajouté la sûreté comme neuvième caractéristique, avec pour sous-caractéristiques la contrainte opérationnelle, l'identification des risques, la sécurité intrinsèque en cas de défaillance (fail safe), l'avertissement de danger et l'intégration sûre.

Les domaines critiques pour la sûreté ont leurs propres normes, qui imposent une architecture et un processus : IEC 61508 pour la sécurité fonctionnelle en général, ISO 26262 pour les véhicules routiers, IEC 62304 pour les logiciels de dispositifs médicaux et DO-178C pour les logiciels embarqués aéronautiques. Les tactiques typiques sont un comportement par défaut sûr (un train s'arrête quand les signaux sont perdus), des canaux de surveillance indépendants capables de prendre le pas sur le contrôleur principal, la redondance avec vote, et une séparation stricte des composants critiques pour la sûreté de tout le reste, afin qu'une faute (fault) dans le système d'infodivertissement ne puisse pas atteindre les freins. Notez que sécurité intrinsèque et haute disponibilité peuvent entrer en conflit : un système à sécurité intrinsèque s'arrête souvent, ce qui est sûr mais indisponible.

L'observabilité, une qualité émergente

L'observabilité est le degré auquel vous pouvez comprendre l'état interne d'un système depuis l'extérieur, à partir des données qu'il émet, suffisamment bien pour répondre à des questions que vous n'aviez pas anticipées lors de sa construction. Elle ne figure pas dans l'ISO/IEC 25010, même si l'analysabilité et l'imputabilité s'en approchent. Dans les systèmes distribués, elle est devenue une qualité à part entière, car le délai de rétablissement, et donc la disponibilité, dépend de la rapidité avec laquelle on parvient à trouver ce qui ne va pas. Sur le plan architectural, elle exige des journaux structurés, des métriques et des traces distribuées avec des identifiants de corrélation qui suivent une requête d'un service à l'autre, prévus dès le départ. Un scénario pourrait être : «Pour toute étape de paiement en échec, un ingénieur d'astreinte peut identifier le service défaillant et le chemin de la requête en moins de 5 minutes.» Voir l'observabilité au chapitre 9.

Qualités juridiques et réglementaires

Les lois et règlements créent des qualités faciles à négliger jusqu'à un audit. La résidence des données exige que certaines données soient stockées ou traitées dans un pays ou une région donnés ; elle peut découler de règles sectorielles, de contrats du secteur public ou d'exigences de clients. Le RGPD lui-même n'exige pas que les données restent dans l'UE, mais les transferts hors de l'UE nécessitent un fondement juridique et des garanties, ce qui, en pratique, détermine où s'exécutent les services et les sauvegardes. L'**auditabilité** consiste à pouvoir montrer, après coup, qui a fait quoi et quand : des journaux révélant toute altération, conservés pendant une durée définie, avec des horloges synchronisées. Les règles de **conservation et de suppression** exigent que les données soient conservées pendant une durée minimale (registres financiers) et supprimées après une durée maximale ou sur demande (données personnelles), ce qui est difficile dans des systèmes comportant de nombreuses copies, caches et sauvegardes. L'**explicabilité** et le contrôle humain sont exigés pour certaines décisions automatisées, comme l'explique le chapitre sur le règlement européen sur l'IA.

Traitez-les comme des contraintes assorties de critères d'acceptation : « les données clients des clients allemands du secteur public sont stockées et traitées uniquement dans des régions de l'UE, sauvegardes et journaux compris » est testable, et exclut d'emblée certains services managés. Les exigences de protection de la vie privée sont traitées dans la protection de la vie privée dès la conception et le RGPD.

Synthèse : les qualités de cette section et des exemples de mesures

Qualité	Exemple de mesure de la réponse	Levier architectural typique
Sécurité (confidentialité)	Aucune donnée personnelle lisible depuis une sauvegarde volée	Chiffrement au repos avec des clés gérées séparément
Sécurité (imputabilité)	Chaque modification des coordonnées d'un bénéficiaire imputable à une seule personne, journaux révélant toute altération	Journal d'audit central, stockage à écriture unique
Capacité d'interaction	95 % des nouveaux utilisateurs terminent le paiement sans aide en moins de 3 minutes	Parcours simples, réponses rapides, annulation
Accessibilité	Tous les parcours clients satisfont aux WCAG 2.2 niveau AA	Design system accessible, vérifications automatisées en CI
Interopérabilité	Nouveau partenaire intégré avec la norme d'API publiée en moins de 4 semaines	Formats standard, adaptateurs, API versionnées
Coût	Infrastructure à moins de 1 penny par commande pour 3 millions de commandes par mois	Bon dimensionnement, cache, autoscaling
Durabilité	SCI inférieur à 0,1 g CO_2e par commande	Efficacité, taux d'utilisation, planification consciente du carbone
Observabilité	Service défaillant identifié en moins de 5 minutes pour toute requête en échec	Journaux structurés, métriques, traçage distribué
Résidence des données	Toutes les données des clients de l'UE, sauvegardes comprises, restent dans des régions de l'UE	Stockage et traitement rattachés à une région

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Quelle est la différence entre l'imputabilité et la non-répudiation ?**

L'imputabilité signifie que les actions peuvent être rattachées à l'entité qui les a effectuées ; la non-répudiation signifie qu'il existe une preuve suffisamment solide pour que l'entité ne puisse pas nier l'action par la suite.

▼ **Pourquoi « annuler les cinq dernières actions » compte-t-il comme une exigence architecturale, et pas seulement comme une exigence d'interface utilisateur ?**

Parce qu'il faut modéliser les opérations comme des commandes réversibles avec un historique stocké, ce qui influe sur la façon dont les changements d'état sont structurés dans tout le système.

▼ **Une API consomme 800 kWh par mois sur un réseau électrique à 200 g CO2e par kWh, se voit imputer 40 kg d'émissions incorporées et traite 2 000 000 de requêtes. Quel est son SCI par requête ?**

(800 x 200 g + 40 000 g) / 2 000 000 = 200 000 / 2 000 000 = 0,1 g CO2e par requête.

▼ **En quoi la sûreté diffère-t-elle de la sécurité ?**

La sécurité protège le système et ses données contre des personnes mal intentionnées ; la sûreté protège les personnes et l'environnement contre les dommages que le système pourrait causer, même en l'absence de tout acteur malveillant.

▼ **À vous : Une API de billetterie coûte £18 000 par mois pour 1 500 000 billets et consomme 800 kWh à 300 g CO2e par kWh, avec 40 kg d'émissions incorporées. Calculez le coût et le SCI par billet, puis le SCI si le cache réduit l'énergie de 25 %.**

Coût : £18 000 / 1 500 000 = 1,2 pence par billet. Émissions d'exploitation : 800 x 300 g = 240 kg ; en ajoutant 40 kg, on obtient 280 kg, et 280 000 g / 1 500 000 = environ 0,187 g par billet. Avec le cache : 600 kWh donnent 180 kg, plus 40 kg font 220 kg, soit environ 0,147 g par billet. Les chiffres par unité gardent leur sens quand le volume augmente : suivez-les donc plutôt que la facture totale.

SOURCES

- W3C: Web Content Accessibility Guidelines (WCAG) 2.2: https://www.w3.org/TR/WCAG22/
- W3C WAI: What's new in WCAG 2.2: https://www.w3.org/WAI/standards-guidelines/wcag/new-in-22/
- Green Software Foundation: Software Carbon Intensity (SCI) specification: https://sci.greensoftware.foundation/
- Nielsen Norman Group: Response Times, the 3 Important Limits: https://www.nngroup.com/articles/response-times-3-important-limits/
- OWASP Application Security Verification Standard: https://owasp.org/www-project-application-security-verification-standard/

SECTION 2.8 SUR 2.8 · INTERMÉDIAIRE · 22 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Éliciter, hiérarchiser et arbitrer les attributs de qualité

AVANT DE LIRE

Cette section s'appuie sur :

- 1.5 Parties prenantes et préoccupations
- 1.6 Décisions et compromis
- 2.3 Scénarios d'attributs de qualité

Termes déjà vus qu'elle utilise : Partie prenante, Préoccupation, Scénario d'attribut de qualité, Percentile (p95, p99), Enregistrement de décision d'architecture (ADR)

L'IDÉE EN BREF

Quand on prépare des vacances en famille, chacun veut quelque chose de différent : l'un veut que ce soit bon marché, un autre veut une piscine, un autre veut des trajets courts. Vous ne pouvez pas tout avoir, alors vous vous asseyez ensemble, vous listez ce dont chacun a vraiment besoin, vous votez sur ce qui compte le plus et vous convenez de ce à quoi renoncer. Les architectes font de même avec les qualités d'un système : ils réunissent toutes les personnes concernées, transforment les souhaits en objectifs précis et mesurables, les classent et consignent les compromis qu'ils choisissent.

CE QU'IL FAUT SAVOIR

- Les exigences de qualité doivent être élicitées auprès des parties prenantes et à partir des objectifs métier, de la réglementation, des incidents passés, des concurrents et des mesures du système existant, car elles sont rarement énoncées spontanément.
- Le Quality Attribute Workshop du SEI comporte huit étapes ; il produit des facteurs architecturaux et des scénarios affinés et hiérarchisés, classés par un vote avec un nombre de voix fixe.
- Un arbre d'utilité décompose l'utilité en attributs de qualité, en raffinements et en scénarios concrets, chacun étant noté (importance métier, difficulté technique) ; les scénarios (H,H) reçoivent en premier l'attention de la conception.
- Les qualités entrent en conflit (performances et sécurité, disponibilité et cohérence, modifiabilité et performances, coût et presque tout le reste, utilisabilité et sécurité), et la bonne résolution dépend du contexte et de la valeur métier.

- Consignez les compromis importants dans des enregistrements de décision et rendez chaque cible mesurable avec une échelle, un instrument de mesure et des niveaux, puis suivez-la dans le temps.
- Les fonctions d'aptitude transforment les mesures de la réponse des scénarios en vérifications automatisées qui s'exécutent à mesure que le système évolue.

D'où viennent les exigences de qualité

Les exigences fonctionnelles arrivent généralement dans un backlog. Les exigences de qualité, rarement : les parties prenantes les tiennent pour acquises (« bien sûr que ce doit être rapide et sécurisé ») ou ne savent pas qu'elles doivent les énoncer. Les architectes doivent donc les éliciter activement, à partir de plusieurs sources.

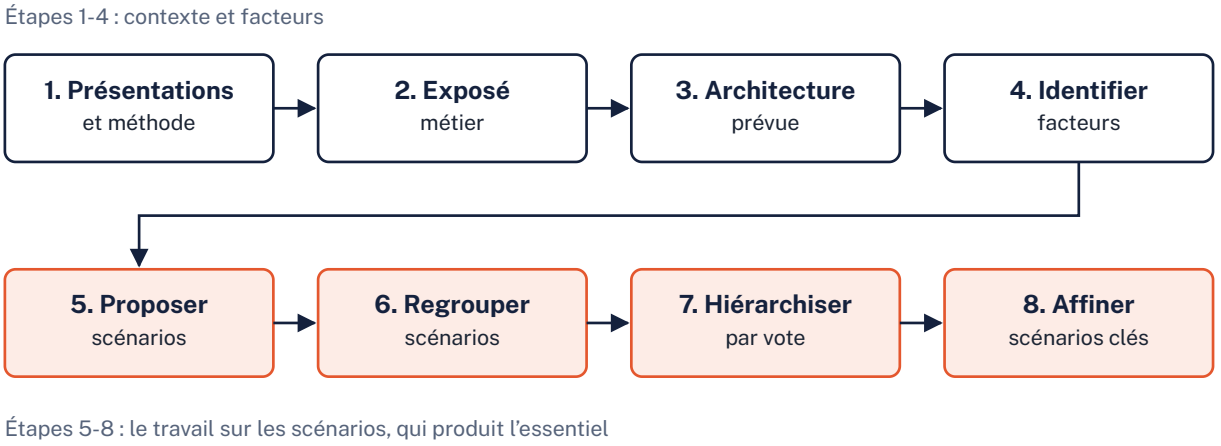
Les parties prenantes. Chaque groupe a des préoccupations différentes : les clients tiennent à la rapidité et à la facilité d'utilisation, l'exploitation à la supervision et au rétablissement, la finance au coût d'exploitation, le support au diagnostic des problèmes, la conformité aux pistes d'audit. La cartographie des parties prenantes du chapitre 1 est le point de départ. **Les objectifs métier.** « S'implanter dans trois nouveaux pays l'an prochain » implique la localisation, la résidence des données et la scalabilité ; « réduire de 20 % les coûts du support client » implique l'utilisabilité et le libre-service. **La réglementation et les contrats.** La protection des données, le droit de l'accessibilité, les règles sectorielles et les SLA clients créent des contraintes non négociables. **Les incidents.** Chaque panne, violation ou journée de lenteur passée révèle une qualité qui était sous-spécifiée ; les revues post-incident sont une source précieuse. **Les concurrents et le marché.** Si les concurrents chargent en une seconde, deux secondes sont un problème de qualité. **Le système existant.** Son comportement mesuré aujourd'hui (charge réelle, percentiles de latence réels, délai réel de mise en œuvre des changements) fournit une base de référence qui transforme « plus rapide » en chiffre.

Les entretiens et les questionnaires fonctionnent, mais les attributs de qualité émergent mieux lorsque les parties prenantes s'écoutent les unes les autres. Un product owner qui apprend du responsable de l'exploitation ce que coûte une heure d'indisponibilité le vendredi à l'heure du dîner établira d'autres priorités. C'est l'idée qui sous-tend les ateliers structurés.

Le Quality Attribute Workshop

Le Quality Attribute Workshop (QAW), mis au point au Software Engineering Institute (Barbacci et ses collègues ; la troisième édition du rapport de méthode est parue en 2003), est une séance animée, qui dure généralement une journée, organisée tôt, avant que l'architecture soit figée. Il réunit de 5 à 30 parties prenantes pour produire des scénarios d'attributs de qualité affinés et hiérarchisés. Il comporte huit étapes.

Les huit étapes du QAW



Les quatre premières étapes établissent le contexte et les facteurs ; les quatre dernières produisent, fusionnent, classent et affinent les scénarios.

Ce qui se passe à chaque étape

1. **1. Présentation et tour de table**
   Les animateurs expliquent l'objectif et la méthode ; les participants se présentent et exposent leurs préoccupations.

2. **2. Présentation métier ou de la mission**
   Un représentant du métier décrit le contexte, les objectifs et les exigences de haut niveau : pourquoi le système existe et ce qui compte le plus.

3. **3. Présentation du plan d'architecture**
   Un responsable technique présente les plans actuels, les systèmes existants, les contraintes connues et les engagements technologiques, aussi préliminaires soient-ils.

4. **4. Identification des facteurs architecturaux**
   Les animateurs dégagent les principaux facteurs architecturaux entendus jusque-là (objectifs, qualités, contraintes) et les valident avec le groupe.

5. **5. Proposition de scénarios (brainstorming)**
   Les participants proposent des scénarios à tour de rôle ; les animateurs veillent à ce que chaque facteur en ait au moins un, et à ce que les scénarios d'usage normal, de croissance et exploratoires (de stress) soient tous représentés.

6 **6. Regroupement des scénarios**
Les quasi-doublons sont fusionnés, mais seulement avec l'accord de leurs auteurs, afin que les voix ne soient pas dispersées.

7 **7. Hiérarchisation des scénarios**
Chaque participant reçoit un nombre de voix égal à environ 30 % du nombre de scénarios et les répartit en deux tours, avec autant de voix qu'il le souhaite par scénario. Le décompte classe les scénarios.

8 **8. Affinage des scénarios**
Les quatre ou cinq premiers scénarios sont rédigés en entier : source, stimulus, artefact, environnement, réponse et mesure de la réponse, avec les objectifs métier qu'ils servent et les questions ouvertes.

Les équipes qui ne peuvent pas y consacrer une journée entière organisent souvent un « mini-QAW » plus court, une variante plus légère décrite par des praticiens comme Michael Keeling, Will Chaparro et Thijmen de Gooijer, qui utilise une taxonomie des attributs de qualité comme support et quelques heures avec un groupe plus restreint. L'essentiel demeure : les parties prenantes réunies, des scénarios plutôt que des adjectifs, un vote explicite et l'affinage des scénarios retenus.

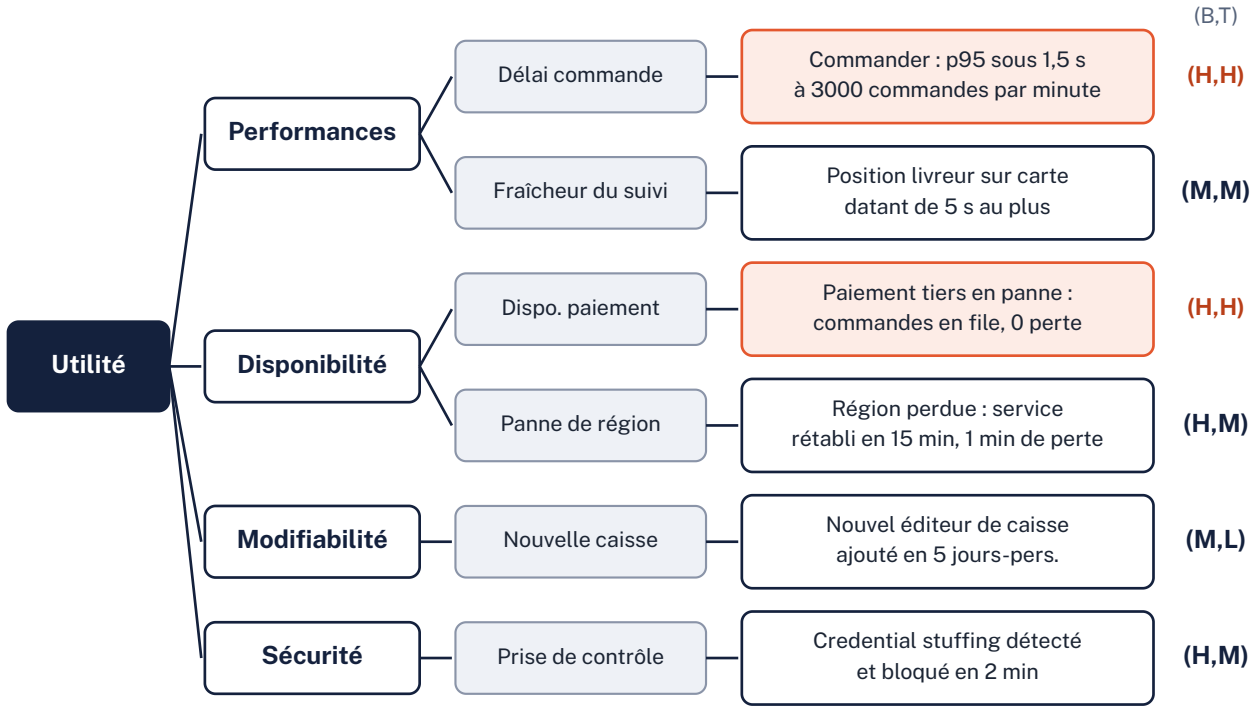
L'arbre d'utilité

Un arbre d'utilité est un moyen d'organiser et de hiérarchiser les exigences de qualité, introduit dans le cadre de l'Architecture Tradeoff Analysis Method (ATAM), que Rick Kazman, Mark Klein et Paul Clements ont mise au point au SEI ; la méthode d'évaluation elle-même est traitée au chapitre 11. La racine est l'*utilité*, la qualité globale du système. Le deuxième niveau contient les attributs de qualité (performances, disponibilité, etc.). Le troisième niveau décompose chacun d'eux en préoccupations plus précises (pour les performances : latence des commandes, fraîcheur du suivi). Les feuilles sont des scénarios concrets et mesurables.

Chaque feuille est notée par une paire : l'**importance métier** (à quel point le scénario compte pour l'entreprise : élevée (H), moyenne (M) ou faible (L)) et la **difficulté technique** ou le risque (à quel point il est difficile à atteindre avec l'architecture : élevée, moyenne ou faible). La paire s'écrit (B,T), par exemple (H,M). Les scénarios notés (H,H) sont les priorités absolues de l'architecture : ce sont les plus importants et les plus difficiles, ils méritent donc un effort de conception précoce, des prototypes et des analyses. Les scénarios (H,L) comptent mais sont faciles, donc une approche standard suffira ; les scénarios (L,H) sont candidats à l'abandon ou au report, car ils sont difficiles mais ne valent pas grand-chose.

L’arbre est utile même en dehors d’une évaluation formelle. Il rend les compromis visibles sur une seule page, montre les lacunes (une qualité sans scénario) et empêche la partie prenante la plus bruyante de fixer seule les priorités.

Un arbre d’utilité pour une plateforme de livraison de repas



Deux scénarios notés (H,H), mis en évidence, sont les priorités absolues de l’architecture ; le scénario de modifiabilité (M,L) peut être satisfait par une bonne conception ordinaire.

EXEMPLE COMMENTÉ

Exemple commenté : construire l'arbre d'utilité d'une plateforme de livraison de repas

Une plateforme de livraison de repas met en relation des clients, 4000 restaurants et 9000 livreurs dans un pays. L'entreprise prévoit de doubler le volume de commandes d'ici 18 mois. Le pic du vendredi soir atteint aujourd'hui 3000 commandes par minute. Un QAW de deux heures a produit 26 scénarios ; avec 12 participants, chacun disposait de 8 voix (30 % de 26 font 7,8, arrondi au nombre pair supérieur, 8). Les scénarios ayant reçu le plus de voix sont devenus les feuilles de l'arbre.

Performances : latence des commandes (H,H)	Les clients passent commande au rythme de pointe de 3000 par minute ; 95 % des confirmations de commande reviennent en moins de 1,5 s et 99 % en moins de 3 s. Importance élevée : les paniers abandonnés sont du chiffre d'affaires perdu. Difficulté élevée : la requête traverse les services du panier, de la tarification, des paiements et des restaurants
Vérification de capacité pour ce scénario	3000 commandes/min = 50 commandes/s. Doubler le volume donne 100 commandes/s. La loi de Little exige le temps moyen, et non un percentile (section 2.4) ; la cible de 1,5 s ne donne donc qu'un chiffre de planification prudent : si la moyenne atteignait 1,5 s, il y aurait environ 100 x 1,5 = 150 commandes en cours simultanément ; la moyenne réelle est plus basse, car le temps moyen se situe bien en dessous du p95
Disponibilité : disponibilité du paiement (H,H)	Lorsque le prestataire de paiement dépasse son délai d'expiration pendant le pic, les commandes sont acceptées et mises en file d'attente pour paiement, aucune commande confirmée n'est perdue, et les clients sont informés en moins de 10 s. Difficile, car cela exige un paiement asynchrone, des nouvelles tentatives idempotentes et une expérience utilisateur pour l'état « en attente »
Disponibilité : panne de région (H,M)	Si la région cloud tombe en panne, le service reprend dans une deuxième région en moins de 15 minutes (RTO) avec au plus 1 minute de données de commande perdues (RPO). Important, mais techniquement bien maîtrisé
Sécurité : prise de contrôle de compte (H,M)	Une attaque par bourrage d'identifiants (credential stuffing) est détectée et bloquée en moins de 2 minutes ; l'attaquant ne prend le contrôle d'aucun compte protégé par l'authentification multifacteur
Performances : fraîcheur du suivi (M,M)	La position d'un livreur sur la carte du client n'a jamais plus de 5 s de retard sous la charge de pointe
Modifiabilité : nouvel éditeur de caisse (M,L)	Un nouvel éditeur de logiciel de caisse pour restaurants est intégré en 5 jours-personne par l'ajout d'un adaptateur (section 2.6)
Priorités qui en résultent	Prototyper et tester en charge d'abord le parcours de commande et la conception du paiement mis en file ; utiliser des patrons multirégions standard pour la panne de région ; garder l'interface de l'adaptateur simple

Les notes sont des jugements convenus par le groupe, pas des calculs. Consignez qui a noté quoi et pourquoi, afin que les notes puissent être contestées plus tard.

L'arbre pilote le plan de travail : les deux scénarios (H,H) font l'objet de spikes architecturaux dès les premiers sprints, tandis que le scénario (M,L) ne nécessite aucune attention particulière.

Le suivi des livreurs n'a été noté que d'importance moyenne, ce qui a surpris les ingénieurs ; le product owner a expliqué que les clients regardent surtout l'heure d'arrivée estimée, pas la carte.

Hiérarchiser quand tout est « critique »

Chaque partie prenante notera sa propre préoccupation comme élevée. Les techniques qui obligent à faire de vrais choix sont utiles. Un nombre de voix fixe (comme dans le QAW) oblige chacun à dépenser des voix rares. La comparaison par paires (« si vous ne pouviez en avoir qu'une, une recherche en moins d'une seconde ou aucune perte de données lors d'un basculement ? ») révèle le véritable ordre. Placer un montant ou un délai à côté de chaque cible (« quatre neuf coûteront 9000 £ de plus par mois et deux mois de travail ») fait changer bien des avis. Enfin, distinguer les *contraintes* (exigences légales d'accessibilité, résidence des données) des *préférences* évite que des éléments non négociables soient sacrifiés par accident.

Les priorités ne sont pas figées. Une start-up qui cherche ses premiers clients place à juste titre le délai de mise sur le marché et la modifiabilité au-dessus d'une disponibilité à cinq neuf ; le même produit, trois ans plus tard, avec des clients grands comptes et des SLA contractuels, doit revoir son classement. Revisitez l'arbre d'utilité quand l'activité change.

Les conflits entre qualités

Les qualités interagissent. Certaines se renforcent mutuellement : la modularité aide la testabilité, qui aide la déployabilité. Beaucoup entrent en conflit, parce que la tactique qui favorise une qualité en pénalise une autre. L'ATAM, décrite par Bass, Clements et Kazman, appelle points de compromis les points où une décision d'architecture affecte plusieurs qualités. Connaître les conflits habituels vous permet de les soulever avant qu'ils ne surprennent qui que ce soit.

Conflits courants entre qualités, et comment le contexte les résout

Conflit	Pourquoi elles s'opposent	Comment le résoudre selon le contexte
Performances contre sécurité	Le chiffrement, les contrôles d'authentification, la validation des entrées et la journalisation d'audit ajoutent du travail à chaque requête	Mesurer le surcoût réel (souvent faible avec le TLS et le matériel modernes) ; mettre brièvement en cache les décisions d'autorisation ; repousser les contrôles coûteux en périphérie ; ne jamais sacrifier la sécurité de données dont l'exposition causerait un préjudice grave
Disponibilité contre cohérence	Pendant une partition réseau, un système répliqué doit choisir entre répondre (éventuellement avec des données périmées) et refuser	Décider opération par opération : une carte ou la position d'un livreur peut être légèrement périmée ; un paiement ou une réservation de stock, non. Voir CAP et PACELC au chapitre 6
Modifiabilité contre performances	Les couches, les interfaces, les intermédiaires et la liaison tardive ajoutent de l'indirection et des sauts	Garder l'indirection là où le changement est probable ; optimiser le chemin critique mesuré ; accepter quelques raccourcis délibérés, documentés dans un enregistrement de décision
Coût contre presque tout le reste	La redondance, les régions supplémentaires, davantage de tests et du matériel plus rapide coûtent tous de l'argent	Rattacher chaque cible de qualité à sa valeur métier ; utiliser des budgets d'erreur et le coût unitaire ; n'acheter des « neuf » que là où une panne coûte plus cher que sa prévention
Utilisabilité contre sécurité	L'authentification forte, les délais d'expiration et les confirmations ajoutent de la friction	Authentification fondée sur le risque (renforcée seulement pour les actions à risque), passkeys et authentification unique (single sign-on), durées de session raisonnables ; associer les utilisateurs aux tests
Délai de mise sur le marché contre maintenabilité	Les raccourcis accélèrent la première version et ralentissent les suivantes	Contracter de la dette technique délibérément, la consigner et planifier son remboursement (chapitre 1)

La bonne résolution dépend toujours du contexte. « La sécurité prime sur les performances » est un slogan, pas une décision : 2 ms de plus de négociation TLS sur un site web public sont sans importance, alors que des contrôles antifraude synchrones à chaque passage de carte dans un stade bondé peuvent poser un vrai problème qui mérite une réponse architecturale (un modèle « approuver d'abord, vérifier ensuite » avec des plafonds). Demandez-vous : quel objectif métier chaque camp sert-il, combien coûte chaque option, et qui doit donner son accord ?

SI VOUS DÉBUTEZ

Pas à pas : un conflit, deux réponses

Le point qui fait le plus souvent trébucher ici est qu'un conflit entre deux qualités ne se règle pas une fois pour toutes pour l'ensemble du système. Il se règle opération par opération, selon ce que chacune vaut pour l'entreprise. Suivez la ligne « disponibilité contre cohérence » du tableau à travers la plateforme de livraison de repas.

- Supposons que la plateforme conserve des copies de certaines de ses données sur plusieurs machines, afin de survivre à une panne, et transmette chaque modification d'une copie aux autres.
- Supposons maintenant que la liaison réseau entre deux copies soit coupée pendant un certain temps : c'est une partition réseau. Une copie qui ne peut pas être informée des dernières modifications peut soit continuer à répondre avec ce qu'elle a, qui peut être périmé, soit refuser jusqu'au rétablissement de la liaison. Elle ne peut pas promettre les deux. Le chapitre 6 revient sur ce choix.
- Position du livreur : un point sur la carte du client vieux de quelques secondes fait peu de dégâts, et les clients regardent de toute façon surtout l'heure d'arrivée estimée. La disponibilité l'emporte : on continue à répondre.
- Paiement : agir sur des informations périmées pourrait conduire à débiter un client deux fois, ou à considérer une commande comme payée alors que le paiement n'est jamais passé. La cohérence l'emporte : on retient le paiement et on affiche la commande comme en attente plutôt que de deviner.
- Une même plateforme donne donc des réponses opposées au même conflit. Aucune ne vient d'un slogan comme « la cohérence prime sur la disponibilité » ; chacune découle de ce que coûterait une mauvaise réponse pour cette opération.
- **La leçon :** quand deux qualités entrent en conflit, demandez-vous quel objectif métier chaque camp sert pour cette opération précise, et décidez à ce niveau. Consignez ensuite le choix et ses raisons, comme l'explique la partie suivante de cette section, pour que personne ne revienne plus tard dessus sans savoir pourquoi.

Consigner les décisions et suivre les qualités

Un compromis qui n'est pas consigné par écrit sera de nouveau débattu, souvent par des personnes qui n'étaient pas dans la salle. Consignez les compromis importants sous forme d'enregistrements de décision d'architecture (chapitre 4) : le contexte, les qualités en tension, les options envisagées, la décision et ses conséquences, y compris ce à quoi vous avez sciemment renoncé.

Rendez ensuite chaque cible de qualité mesurable et visible. Le Planguage de Tom Gilb propose un format rigoureux : une *échelle* (ce qui est mesuré, dans quelle unité), un *instrument de mesure* (comment c'est mesuré), et des niveaux comme *échec* (le seuil en deçà duquel l'exigence n'est pas satisfaite), *objectif* (la cible) et *cible ambitieuse* ou *souhait* (des niveaux plus ambitieux). Appliqué au scénario de latence des commandes : échelle, secondes entre l'envoi et la confirmation au 95e percentile ; instrument de mesure, supervision des utilisateurs réels pendant le pic du vendredi ; échec au-delà de 3 s ; objectif 1,5 s ; cible ambitieuse 1 s.

Enfin, suivez les mesures dans le temps sur un tableau de bord visible par les parties prenantes, à côté des SLO (chapitre 9), et revoyez-les chaque fois que l'activité, la charge ou l'équipe change.

Des scénarios aux fonctions d'aptitude

Le moyen le plus sûr de garantir qu'une exigence de qualité reste satisfaite est de la vérifier automatiquement, à chaque modification du système. Neal Ford, Rebecca Parsons et Patrick Kua, dans *Building Evolutionary Architectures* (2017 ; deuxième édition avec Pramod Sadalage, 2022), appellent une telle vérification une fonction d'aptitude architecturale : une évaluation objective de la mesure dans laquelle le système satisfait une caractéristique d'architecture. Un test de charge dans le pipeline qui fait échouer le build si la latence p95 des commandes dépasse 1,5 s à 100 commandes par seconde est une fonction d'aptitude. Il en va de même d'un test qui échoue si un module extérieur à `payments` importe la bibliothèque du prestataire de paiement (ce qui protège le scénario de modifiabilité), d'un exercice de basculement planifié qui mesure le délai de rétablissement, ou d'une analyse des dépendances qui bloque les bibliothèques vulnérables connues.

Les fonctions d'aptitude sont l'aboutissement naturel du travail de ce chapitre : un souhait vague devient un scénario (section 2.3), le scénario reçoit une priorité dans l'arbre d'utilité, et la mesure de la réponse du scénario devient une vérification automatisée. Toutes les qualités ne peuvent pas être automatisées à peu de frais (l'utilisabilité a toujours besoin de personnes), mais chacune de celles que vous automatisez cesse de s'éroder en silence. Le chapitre 11 traite en profondeur de leur conception : fonctions d'aptitude et architecture évolutive.

Liste de contrôle : animer un atelier sur les attributs de qualité

Adaptée à une séance d'une demi-journée ou d'une journée ; un « mini-QAW » plus court peut couvrir le même terrain avec un groupe plus restreint.

- [ ] **Inviter le bon panel**
  De 5 à 30 personnes : responsables métier, utilisateurs ou leurs représentants, exploitation, sécurité, support, développeurs, et toute personne qui valide la conformité. L'absence de l'exploitation est la lacune la plus courante.
- [ ] **Envoyer une courte lecture préalable**
  Les objectifs métier, les contraintes connues et une liste provisoire d'attributs de qualité (l'ISO/IEC 25010 comme support), pour que chacun arrive en ayant déjà réfléchi.
- [ ] **Commencer par les objectifs métier, pas par la technologie**
  Une partie prenante de haut niveau explique pourquoi le système existe, à quoi ressemblerait la réussite et ce que coûterait un échec.
- [ ] **S'accorder sur les facteurs architecturaux**
  Une courte liste des objectifs, qualités et contraintes qui façonneront l'architecture, validée avec le groupe avant le début des scénarios.
- [ ] **Proposer des scénarios par tours**
  Chacun en apporte au moins deux, à tour de rôle, pour que les voix discrètes soient entendues. Incluez l'usage normal, la croissance (« dans deux ans ») et le stress (« quand le prestataire de paiement est en panne »).
- [ ] **Ne fusionner les doublons qu'avec accord**
  Le regroupement évite la dispersion des voix, mais l'auteur doit convenir que deux scénarios signifient vraiment la même chose.
- [ ] **Voter avec un nombre de voix fixe**
  Chaque personne reçoit un nombre de voix égal à environ 30 % du nombre de scénarios, réparties en deux tours. Cela révèle les vraies priorités et évite le « tout est critique ».
- [ ] **Affiner entièrement les premiers scénarios**
  Rédigez les quatre ou cinq premiers sous forme de scénarios en six parties avec des mesures de la réponse mesurables, et notez l'objectif métier que chacun sert.
- [ ] **Consigner les questions ouvertes et les conflits**
  Les désaccords sur la priorité ou la faisabilité sont des résultats précieux ; listez-les avec un responsable.

- [ ] **Publier le résultat sous quelques jours**
  Partagez les scénarios hiérarchisés, ajoutez-les à la documentation d'architecture et au backlog, et planifiez le moment où chacun sera vérifié.

ATTENTION

Erreurs courantes

- Recueillir les qualités uniquement auprès du commanditaire métier : les préoccupations d'exploitation, de sécurité et de support apparaissent alors après la mise en service, quand elles coûtent cher.
- Accepter des adjectifs : « scalable » et « robuste » sur une liste d'exigences sans scénario derrière.
- Noter chaque feuille (H,H) : si tout est prioritaire, rien ne l'est. Insistez jusqu'à ce que les notes fassent vraiment la différence.
- Traiter l'arbre d'utilité comme un artefact ponctuel : il doit être revu quand les objectifs ou la charge changent.
- Résoudre les conflits en silence dans le code : un développeur qui supprime discrètement la journalisation d'audit pour atteindre une cible de latence a pris une décision d'architecture que personne n'a consignée.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Dans un QAW avec 20 scénarios regroupés, combien de voix environ chaque participant reçoit-il ?

Environ 30 % de 20, soit 6 voix, réparties en deux tours de 3.

▼ Une feuille de l'arbre d'utilité est notée (L,H). Que cela suggère-t-il ?

Elle compte peu pour l'entreprise mais est difficile à atteindre : c'est donc une candidate à l'abandon, au report ou à la renégociation, plutôt qu'un élément qui pilote l'architecture.

▼ Pourquoi « la sécurité prime toujours sur les performances » est-il une mauvaise manière de résoudre un conflit ?

Parce que la bonne réponse dépend du surcoût réel, du préjudice que le contrôle de sécurité évite et de la valeur métier des performances ; le conflit doit être résolu au cas par cas, chiffres à l'appui, et consigné.

▼ **Donnez un exemple de fonction d'aptitude pour une exigence de modifiabilité.**

Un test automatisé dans le build qui échoue si un module autre que l'adaptateur de paiement importe la bibliothèque du prestataire de paiement, ce qui garde le prestataire remplaçable.

▼ **À vous : Les parties prenantes jugent tout critique. Évaluez ces éléments (importance, difficulté) pour un arbre d'utilité de livraison de repas : confirmation de commande en moins de 1,5 s au pic ; export du rapport mensuel en moins d'une heure ; ajout d'un prestataire de paiement en moins d'un mois. Lequel reçoit le plus tôt l'attention de la conception ?**

La confirmation de commande est (H,H) : les paniers perdus sont du chiffre d'affaires perdu, et la requête traverse plusieurs services sous charge de pointe ; elle reçoit donc en premier l'attention de la conception. Un nouveau prestataire de paiement pourrait être (M,M) : utile, et moyennement difficile à moins qu'un adaptateur n'existe déjà. L'export du rapport est (L,L) ou (M,L) : il peut facilement s'exécuter la nuit. Ces notes imposent un classement sans prétendre qu'une qualité vaut toujours plus qu'une autre. Consignez les raisons et vérifiez-les avec les parties prenantes par un vote.

SOURCES

- SEI: Quality Attribute Workshop collection: https://www.sei.cmu.edu/library/quality-attribute-workshop-collection/
- SEI: Architecture Tradeoff Analysis Method (ATAM) collection: https://www.sei.cmu.edu/library/architecture-tradeoff-analysis-method-collection/
- IREB RE Magazine: Discover Quality Requirements with the Mini-QAW: https://re-magazine.ireb.org/articles/discover-quality-requirements-with-the-mini-qaw
- Thoughtworks Technology Radar: Architectural fitness function: https://www.thoughtworks.com/radar/techniques/architectural-fitness-function

Lectures complémentaires

- *Software Architecture in Practice (4th edition)*, Len Bass, Paul Clements and Rick Kazman (2021), Addison-Wesley. La source des scénarios d'attributs de qualité et des tactiques, avec un chapitre sur chaque qualité majeure et un autre sur les exigences architecturalement significatives.
  https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/
- *ISO/IEC 25010:2023 Systems and software engineering: Product quality model*, ISO/IEC JTC 1/SC 7 (2023), International Organization for Standardization. Le modèle normalisé de qualité des produits, avec neuf caractéristiques ; sa norme compagne, l'ISO/IEC 25019:2023, couvre la qualité en utilisation.
  https://www.iso.org/standard/78176.html
- *Characterizing Architecturally Significant Requirements*, Lianping Chen, Muhammad Ali Babar and Bashar Nuseibeh (2013), IEEE Software, volume 30, issue 2. Une étude empirique sur la façon dont les praticiens reconnaissent les exigences qui façonnent une architecture.
  https://pure.itu.dk/en/publications/characterizing-architecturally-significant-requirements/
- *The Tail at Scale*, Jeffrey Dean and Luiz André Barroso (2013), Communications of the ACM, volume 56, issue 2. Un article court et facile à lire sur les raisons pour lesquelles de rares réponses lentes dominent les grands systèmes à fan-out, et sur les techniques qu'utilise Google pour les maîtriser.
  https://research.google/pubs/pub40801/
- *Designing Data-Intensive Applications (2nd edition)*, Martin Kleppmann and Chris Riccomini (2026), O'Reilly Media. Un traitement clair de la charge, des percentiles, de la fiabilité et de la scalabilité dans les systèmes de données, avec de nombreux exemples réels.
- *Fundamentals of Software Architecture (2nd edition)*, Mark Richards and Neal Ford (2025), O'Reilly Media. Le point de vue d'un praticien sur l'identification et la hiérarchisation des caractéristiques d'architecture, et sur leur protection par des fonctions d'aptitude.
- *Site Reliability Engineering: How Google Runs Production Systems*, Betsy Beyer, Chris Jones, Jennifer Petoff and Niall Richard Murphy (eds) (2016), O'Reilly Media. Gratuit en ligne ; les chapitres sur les objectifs de niveau de service et la supervision montrent comment les qualités sont mesurées en production.
  https://sre.google/books/
- *Systems Performance: Enterprise and the Cloud (2nd edition)*, Brendan Gregg (2020), Addison-Wesley. La référence en matière d'analyse des performances, des files d'attente et du taux d'utilisation jusqu'à la méthode USE.

Glossaire

Accessibilité
Mesure dans laquelle les personnes en situation de handicap, y compris celles qui utilisent des technologies d'assistance comme les lecteurs d'écran, peuvent utiliser un produit.

Accord de niveau de service (SLA)
Contrat passé avec les clients qui fixe un niveau de service et les conséquences, comme des crédits de service, s'il n'est pas atteint.

Arbre d'utilité
Arbre, issu de l'ATAM, qui décompose l'utilité globale en attributs de qualité, puis en scénarios concrets, chacun évalué selon son importance métier et sa difficulté technique.

Contrainte
Décision de conception déjà prise pour vous, ne laissant aucune ou très peu de liberté, comme un langage de programmation imposé, une réglementation, une échéance fixe ou un budget.

Coût unitaire
Le coût d'exploitation d'un système divisé par une unité de travail utile, comme le coût par commande ou par utilisateur actif, afin de pouvoir suivre l'efficacité à mesure que l'activité croît.

Débit (throughput)
La quantité de travail qu'un système accomplit par unité de temps, par exemple 1200 requêtes par seconde ou 40 000 commandes par heure.

Défaillance (failure)
Événement au cours duquel le service rendu par un système s'écarte de ce qu'il devrait être, d'une manière que ses utilisateurs peuvent observer.

Dégradation gracieuse
Maintien des fonctions les plus importantes lorsqu'une partie d'un système tombe en panne, au lieu d'une panne totale, par exemple une boutique qui continue de prendre des commandes pendant que les recommandations sont hors service.

Déployabilité
Rapidité, sûreté et prévisibilité avec lesquelles une modification peut être mise en production, puis retirée si nécessaire.

Disponibilité
Proportion du temps, ou des requêtes, pendant laquelle un système est prêt à fournir son service lorsqu'on le sollicite, généralement exprimée en pourcentage, par exemple 99,9 %.

Doublure de test (test double)
Substitut d'un collaborateur réel utilisé dans un test, comme un stub aux réponses préparées à l'avance, un fake doté d'une implémentation simple qui fonctionne, ou un mock qui vérifie les appels qu'il reçoit.

Élasticité
Capacité à ajouter et à retirer des ressources automatiquement à mesure que la charge monte et descend, afin que la capacité suive la demande et que vous ne payiez que ce que vous utilisez.

Élicitation
Fait d'extraire activement les exigences des parties prenantes, des documents, des réglementations et des systèmes existants, plutôt que d'attendre qu'elles soient énoncées.

Erreur
Partie de l'état interne d'un système qui est incorrecte parce qu'une faute a été activée, comme un enregistrement corrompu. Une erreur peut entraîner ou non une défaillance.

Exigence architecturalement significative (ASR)
Exigence qui a un effet mesurable sur l'architecture d'un système : si elle changeait, la structure devrait changer. La plupart des ASR sont des attributs de qualité ou des contraintes.

Exigence fonctionnelle
Énoncé de ce qu'un système doit faire : un comportement, un calcul ou une information qu'il doit fournir, comme « un client peut annuler une commande avant qu'elle ne soit expédiée ».

Exigence implicite
Qualité que chaque partie prenante tient pour acquise mais que personne ne consigne par écrit, comme « les commandes ne sont jamais perdues » ou « le système continue de fonctionner lors d'un changement d'heure ».

Exigence non fonctionnelle (NFR)
Nom très répandu mais imprécis pour les exigences de qualité et les contraintes. De nombreux architectes l'évitent, car ces exigences qualifient les fonctions au lieu d'exister à part.

Facteur architectural
Objectif, exigence d'attribut de qualité ou contrainte qui façonne de manière significative l'architecture.

Failover (basculement)
Transfert du travail d'un composant défaillant vers un composant de secours ou une réplique en bon état, automatiquement ou manuellement.

Fan-out (diffusion en éventail)
Patron dans lequel une requête entrante déclenche plusieurs requêtes parallèles vers d'autres services ou serveurs, dont les réponses sont ensuite combinées.

Faute (fault)
Cause d'un problème, comme un bug, un disque défaillant ou une mauvaise configuration. Une faute peut rester latente jusqu'à ce que quelque chose l'active.

Fiabilité
La probabilité qu'un système fonctionne correctement, sans défaillance, pendant une période donnée et dans des conditions données. Un système peut être très disponible tout en étant peu fiable s'il tombe souvent en défaillance mais se rétablit rapidement.

FURPS+
Classification de la qualité née chez Hewlett-Packard sous le nom de FURPS (fonctionnalité, utilisabilité, fiabilité, performances et supportabilité), étendue ensuite par un « + » pour des contraintes comme la conception, l'implémentation, les interfaces et les limites physiques.

Heartbeat (battement de cœur)
Message périodique « je suis en vie » envoyé par un composant ; si les messages cessent, un moniteur considère que le composant est défaillant et agit.

Indicateur de niveau de service (SLI)
Mesure soigneusement définie du comportement d'un service, comme la proportion de requêtes qui réussissent en moins de 300 ms.

Injection de dépendances
Fourniture à un composant, depuis l'extérieur, des collaborateurs dont il a besoin (une base de données, une horloge, une passerelle de paiement) au lieu de le laisser les créer, afin de pouvoir les remplacer, par exemple par des imitations dans les tests.

Interopérabilité
Degré auquel des systèmes peuvent échanger des informations et utiliser correctement ce qu'ils échangent, ce qui exige un accord sur le sens autant que sur le format.

ISO/IEC 25010
Norme internationale de modèle de qualité des produits. L'édition 2023 définit neuf caractéristiques, de l'adéquation fonctionnelle à la sûreté, chacune divisée en sous-caractéristiques.

Latence de queue (tail latency)
Les temps de réponse des requêtes les plus lentes, généralement décrits par des percentiles élevés comme p99 ou p99.9. À grande échelle, c'est souvent la queue qui détermine ce que vivent les utilisateurs.

Loi d'Amdahl
Formule (Gene Amdahl, 1967) montrant que l'accélération obtenue en ajoutant des processeurs est limitée par la part du travail qui doit s'exécuter en série.

Loi de Little
La règle L = λ × W : le nombre moyen d'éléments dans un système stable est égal au taux d'arrivée moyen multiplié par le temps moyen que chaque élément y passe.

Loi universelle de scalabilité (USL)
Le modèle de Neil Gunther décrivant comment la capacité augmente avec l'ajout de nœuds, en tenant compte de la contention pour les ressources partagées et du coût du maintien de la cohérence des données. Il prédit un pic au-delà duquel ajouter des nœuds réduit la capacité.

Maintenabilité
Dans l'ISO/IEC 25010, l'efficacité et l'efficience avec lesquelles un produit peut être modifié, ce qui couvre la modularité, la réutilisabilité, l'analysabilité, la modifiabilité et la testabilité.

Mesure de la réponse
Le chiffre qui rend un scénario testable, par exemple « 99 % en moins de 800 ms », « service rétabli en moins de 2 minutes » ou « changement réalisé en 3 jours-personne ».

Métriques DORA
Mesures de la performance de livraison logicielle issues du programme de recherche DORA : délai de mise en œuvre des changements, fréquence de déploiement, temps de rétablissement après un déploiement en échec, taux d'échec des changements et taux de reprise des déploiements.

Mise en file d'attente (queueing)
Attendre une ressource occupée. La théorie des files d'attente étudie comment le temps d'attente augmente avec le taux d'utilisation ; elle explique pourquoi le temps de réponse grimpe brusquement lorsqu'une ressource approche 100 % d'occupation.

Modèle de qualité
Liste structurée de caractéristiques de qualité et de leurs sous-caractéristiques, utilisée comme vocabulaire commun et comme liste de contrôle pour spécifier ou évaluer la qualité.

Modifiabilité
La facilité et la sûreté avec lesquelles un système peut être modifié à moindre coût : le coût, l'effort et le risque d'un changement, et la mesure dans laquelle il se propage au-delà de l'endroit où il est fait.

Moment de liaison (binding time)
Moment où une décision est figée : à l'écriture du code, au build, au déploiement, au démarrage ou pendant l'exécution du système.

Objectif de niveau de service (SLO)
Cible interne pour un SLI sur une fenêtre de temps, par exemple 99,9 % de requêtes réussies sur 30 jours.

Objectif de point de reprise (RPO)
La plus grande quantité de données, mesurée en durée, que l'entreprise peut accepter de perdre en cas de sinistre, par exemple les 15 dernières minutes de modifications.

Objectif de temps de reprise (RTO)
Le délai le plus long que l'entreprise peut accepter entre un sinistre et le rétablissement du service.

Observabilité
La capacité des personnes à comprendre ce qui se passe à l'intérieur d'un système en fonctionnement à partir des données qu'il émet (journaux, métriques, traces), y compris pour des problèmes que personne n'avait anticipés.

Percentile (p95, p99)
La valeur sous laquelle se situe une proportion donnée des mesures. Un p99 de 400 ms signifie que 99 % des requêtes ont pris 400 ms ou moins et que 1 % ont pris plus longtemps.

Point de compromis
Décision architecturale qui affecte plus d'un attribut de qualité, en améliorant l'un tout en nuisant à un autre.

Qualité en utilisation
La qualité qu'un produit apporte aux personnes dans un contexte d'utilisation donné (s'il est bénéfique, exempt de risque et acceptable). Depuis 2023, elle a sa propre norme, l'ISO/IEC 25019.

Quality Attribute Workshop (QAW)
Méthode animée du SEI dans laquelle les parties prenantes proposent, regroupent, votent et affinent des scénarios d'attributs de qualité au début d'un projet.

Redondance
Disposer de plus d'un exemplaire d'un composant afin qu'un autre puisse prendre le relais en cas de défaillance de l'un d'eux.

Résidence des données
Exigence selon laquelle certaines données doivent être stockées, et parfois traitées, uniquement dans un pays ou une région donnés.

Résilience
La capacité plus large d'un système à absorber une perturbation, à maintenir son service essentiel (éventuellement sous une forme dégradée) et à revenir ensuite à la normale.

Retour arrière (rollback)
Ramener un système à un état ou à une version antérieure connue comme saine après qu'un changement a causé des problèmes.

Scalabilité
La capacité d'un système à absorber davantage de charge en ajoutant des ressources, idéalement avec une augmentation prévisible et à peu près proportionnelle de la capacité et du coût.

Scénario concret
Scénario rédigé pour un système particulier, avec des acteurs, des événements, des conditions et des chiffres précis, prêt à servir de base à la conception et aux tests.

Scénario d'attribut de qualité
Énoncé court et testable d'une exigence de qualité en six parties : source du stimulus, stimulus, artefact, environnement, réponse et mesure de la réponse.

Scénario général
Modèle indépendant de tout système qui recense les sources, stimuli, environnements, réponses et mesures typiques pour un attribut de qualité.

Software Carbon Intensity (SCI)
Spécification de la Green Software Foundation, publiée sous la référence ISO/IEC 21031:2024, qui exprime les émissions de carbone d'un logiciel sous forme de taux : ((énergie x intensité carbone du réseau électrique) + émissions incorporées) par unité fonctionnelle.

Sous-caractéristique
Qualité plus spécifique au sein d'une caractéristique d'un modèle de qualité ; par exemple, la tolérance aux pannes et la capacité de récupération sont des sous-caractéristiques de la fiabilité dans l'ISO/IEC 25010.

Stimulus
L'événement sur lequel porte un scénario d'attribut de qualité, comme une rafale de requêtes, la panne d'un serveur, une demande de changement ou une attaque.

Sûreté (safety)
Absence de risque inacceptable de dommage pour les personnes, les biens ou l'environnement causé par le système, que quelqu'un agisse ou non avec malveillance.

Tactique
Décision de conception qui influe sur la manière dont un système répond à un stimulus de qualité, comme la mise en cache pour réduire la latence ou la redondance pour survivre à une panne. Les tactiques sont les blocs de construction des patrons.

Taux d'utilisation
La fraction du temps pendant laquelle une ressource est occupée, généralement notée ρ (rho). Un serveur qui traite 40 requêtes par seconde nécessitant chacune 20 ms de travail est utilisé à 80 %.

Temps de réponse
Le temps total entre l'envoi d'une requête par un client et la réception de la réponse : le temps de service, plus tout temps d'attente dans des files, plus les délais réseau.

Temps de service

Le temps qu'une ressource passe réellement à travailler sur une requête, sans compter le temps pendant lequel la requête attend cette ressource.

Temps moyen de réparation (MTTR)

Le temps moyen entre une défaillance et le rétablissement du service. Certaines organisations comprennent le R comme recover ou restore (rétablir) ; précisez toujours quels points de départ et d'arrivée sont mesurés.

Temps moyen entre défaillances (MTBF)

Le temps moyen qui sépare une défaillance d'un composant ou d'un système réparable de la suivante.

Temps moyen jusqu'à la défaillance (MTTF)

Le temps moyen pendant lequel un composant fonctionne correctement avant de tomber en défaillance.

Testabilité

La facilité avec laquelle on peut montrer, par des tests, qu'un système ou un composant se comporte comme prévu, ce qui suppose de pouvoir contrôler ses entrées et son état et observer ses sorties.

Tolérance aux pannes

Capacité d'un système à continuer de rendre son service même si l'une de ses parties est défaillante, généralement grâce à la redondance.

Triade CIA

Les trois objectifs classiques de la sécurité de l'information : confidentialité (seules les parties autorisées peuvent lire), intégrité (aucune modification non autorisée) et disponibilité (utilisable quand on en a besoin).

WCAG

Les Web Content Accessibility Guidelines (règles pour l'accessibilité des contenus web) du W3C. La version 2.2 (2023) fixe des critères de succès testables aux niveaux A, AA et AAA ; le niveau AA est la référence juridique habituelle.