BE SOFTWARE ARCHITECTURE WISE · CHAPITRE 4

Documenter et communiquer l’architecture

Ce chapitre montre comment décrire et communiquer une architecture logicielle pour que les bonnes personnes la comprennent et que la description reste fidèle. Il traite des destinataires de la documentation et de la quantité à écrire, des vues et des points de vue de l’ISO/IEC/IEEE 42010, du modèle 4+1 et de Rozanski et Woods, du modèle C4, des diagrammes UML que les architectes utilisent encore, des enregistrements de décision d’architecture (ADR), d’arc42, des diagrammes en tant que code et de la présentation de l’architecture à différents publics. Il est pratique de bout en bout, s’appuie sur une même plateforme de livraison de repas comme exemple fil rouge, et s’adresse aux développeurs, aux architectes et aux responsables techniques.

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

EN UN COUP D'ŒIL

Vous apprendrez à

- Expliquer qui lit la documentation d'architecture, ce dont chaque lecteur a besoin et comment décider de ce qui mérite d'être consigné par écrit.
- Distinguer les parties prenantes, les préoccupations, les points de vue, les vues et les types de modèles tels que les définit l'ISO/IEC/IEEE 42010:2022.
- Comparer le modèle de vues 4+1, les points de vue et perspectives de Rozanski et Woods, et les trois catégories de vues du SEI.
- Dessiner des diagrammes de contexte système, de conteneurs et de composants qui respectent les règles de notation du C4.
- Lire et dessiner les diagrammes UML de séquence, d'états-transitions, de composants, de déploiement, d'activité et de classes qu'utilisent les architectes.
- Rédiger des enregistrements de décision d'architecture et gérer leur cycle de vie dans un journal des décisions.
- Appliquer arc42 et les pratiques de diagrammes en tant que code pour que la documentation reste utile et à jour.
- Adapter une explication de l'architecture aux dirigeants, aux développeurs, aux équipes d'exploitation, aux équipes de sécurité et aux auditeurs.

Avant de commencer

Les chapitres 1 à 3, en particulier les parties prenantes et les préoccupations (section 1.5) et les décisions et compromis (section 1.6).

Durée

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

Termes clés

Description d'architecture · Vue · Point de vue · Préoccupation · Modèle de vues 4+1 · Modèle C4 · Conteneur (C4) · UML · Diagramme de séquence · Diagramme d'états-transitions · Enregistrement de décision d'architecture · arc42 · Diagrammes en tant que code · Documentation en tant que code (docs as code)

Pourquoi documenter l'architecture fait partie du métier

Une architecture qui n'existe que dans la tête de quelques personnes est fragile. Personne en dehors de la salle ne peut la revoir, le prochain ingénieur qui arrive ne peut pas l'apprendre, un auditeur ne peut pas la vérifier, et elle part avec les personnes qui la détiennent. Le chapitre 1 a distingué une architecture (ce qu'est réellement le système) d'une description d'architecture (les documents, diagrammes et enregistrements par lesquels les gens la partagent). Ce chapitre porte sur la seconde : comment produire des descriptions que les gens lisent vraiment, qui restent fidèles à mesure que le système évolue, et qui aident à prendre des décisions.

C'est une compétence pratique plus que théorique, et il est facile de se tromper dans les deux sens. Les équipes qui écrivent trop peu perdent les raisons de leur structure et refont sans cesse les mêmes débats. Les équipes qui écrivent trop produisent des wikis auxquels personne ne se fie. Le principe directeur du chapitre vient de l'ouvrage du SEI *Documenting Software Architectures: Views and Beyond* : écrire pour le lecteur. Décidez qui utilisera chaque élément de documentation et ce dont il a besoin, écrivez cela de façon brève et exacte, et n'écrivez rien d'autre.

Un même exemple fil rouge revient tout au long du chapitre : une plateforme de livraison de repas avec une app client, une app restaurant et une app livreur, des services de commandes, d'affectation des livreurs et de notifications, et des paiements par carte via un prestataire externe. Vous la verrez sous forme de diagramme de contexte, de diagramme de conteneurs, de diagramme de composants, de diagrammes UML de séquence et d'états, d'enregistrement de décision et de synthèse arc42 d'une page, afin de comparer chaque technique sur le même système.

Décrire et dessiner : sections 4.1 à 4.4

Les quatre premières sections traitent de *ce qu'il faut décrire* et de *la façon de le dessiner*. La section 4.1 se demande à qui s'adresse la documentation, ce que le Manifeste agile dit et ne dit pas à son sujet, comment décider de ce qui mérite d'être écrit et quels anti-patrons éviter. La réponse courte : écrire à la main ce que le code ne peut pas vous dire (le contexte, les objectifs de qualité, les décisions et leurs raisons) et générer ce que le code indique déjà.

La section 4.2 présente l'idée centrale selon laquelle un seul diagramme ne suffit jamais. Une architecture se décrit au moyen de plusieurs vues, chacune répondant aux préoccupations de parties prenantes précises et chacune construite selon un point de vue. La section couvre le modèle conceptuel de l'ISO/IEC/IEEE 42010:2022, le modèle de vues 4+1 de Kruchten, les points de vue et perspectives de Rozanski et Woods, et les catégories module, composants et connecteurs et allocation du SEI, et montre comment choisir l'ensemble minimal de vues dont un système a besoin.

La section 4.3 transforme cette théorie en l'approche pratique la plus utilisée, le modèle C4 de Simon Brown : quatre niveaux de zoom (contexte système, conteneur, composant, code), quelques diagrammes complémentaires et un petit ensemble de règles de notation qui rendent les diagrammes compréhensibles par eux-mêmes. La section 4.4 couvre les parties d'UML que les architectes utilisent encore, principalement les diagrammes de séquence, d'états-transitions, de composants, de déploiement, d'activité et de petits diagrammes de classes, et explique quand la précision d'UML vaut l'effort.

Décisions, modèles, outils et personnes : sections 4.5 à 4.8

Les diagrammes montrent la structure ; ils montrent rarement le *pourquoi*. La section 4.5 traite des enregistrements de décision d'architecture : le format de Michael Nygard de 2011, des alternatives comme MADR et les Y-statements, le cycle de vie d'un enregistrement, de proposé à remplacé, et un enregistrement complet commenté pour la plateforme de livraison. La section 4.6 montre comment un modèle de documentation comme arc42 donne à l'ensemble de la description une organisation standard, pour que les auteurs sachent quoi couvrir et les lecteurs où chercher, et comment l'utiliser de façon légère pour un petit système.

La section 4.7 s'attaque au problème qui met en échec la plupart des documentations : l'obsolescence. Elle couvre l'écriture de diagrammes sous forme de texte avec PlantUML, Mermaid, Structurizr DSL et D2, la différence entre outils de dessin et outils de modélisation, la publication de la documentation depuis le pipeline de build, ainsi que les habitudes de responsabilité et de revue qui maintiennent la documentation fidèle. La section 4.8 élargit le regard des documents aux personnes : adapter le message aux dirigeants, aux développeurs, aux équipes d'exploitation, aux équipes de sécurité et aux auditeurs, expliquer les compromis, suivre une requête à travers un système, rédiger des documents de conception et mener des processus de RFC, présenter lors d'une revue et construire un consensus quand les avis divergent.

L'idée à retenir

Chaque technique de ce chapitre sert un seul objectif : une compréhension partagée et exacte du système entre les personnes qui le construisent, l'exploitent, le financent et en dépendent. Un diagramme, un enregistrement de décision ou un modèle de documentation ne vaut la peine d'être produit que si un vrai lecteur en a besoin, et seulement tant que quelqu'un le maintient fidèle. Une documentation réduite, à jour, dotée d'un responsable et placée à côté du code vaut mieux qu'un document exhaustif auquel personne ne se fie. Si vous ne retenez qu'une question de ce chapitre, que ce soit celle-ci : *qui va lire ceci, et qu'en fera-t-il ?*

SECTION 4.1 SUR 4.8 · INTERMÉDIAIRE · 28 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Pourquoi documenter, et dans quelle mesure

AVANT DE LIRE

Cette section s'appuie sur :

- 1.1 Définir l'architecture
- 1.5 Parties prenantes et préoccupations

Termes déjà vus qu'elle utilise : Description d'architecture, Partie prenante, Gestion de versions, Pull request

L'IDÉE EN BREF

Pensez aux documents d'une maison. Un acheteur veut un plan des pièces, un électricien veut le schéma de câblage, un assureur veut le rapport d'expertise, et la personne qui la rénovera dans vingt ans veut savoir pourquoi ce mur n'a jamais été déplacé. Personne ne veut la liste de chaque brique, et un plan qui montre un mur abattu depuis longtemps est pire que pas de plan du tout. La documentation d'architecture logicielle, c'est pareil : un petit ensemble de documents exacts, chacun écrit pour quelqu'un qui en a réellement besoin, qui expliquent ce qu'on ne peut pas voir simplement en regardant le code.

CE QU'IL FAUT SAVOIR

- Une description d'architecture est l'ensemble des documents, diagrammes et enregistrements par lesquels les gens partagent une architecture ; elle sert à la formation, à la communication entre parties prenantes et à l'analyse.
- Des lecteurs différents (nouveaux venus, développeurs, exploitation, sécurité, auditeurs, futurs mainteneurs et décideurs) arrivent avec des questions différentes ; la documentation doit donc s'organiser autour des lecteurs et de leurs tâches.
- Le Manifeste agile privilégie les logiciels opérationnels plutôt qu'une documentation exhaustive, mais dit explicitement que la documentation garde de la valeur ; il rejette la documentation pour elle-même, pas la documentation.
- Écrivez à la main ce qui ne peut pas être retrouvé à partir du code (le contexte, les objectifs de qualité, les décisions et leurs raisons, les principes) ; générez ce que le code indique déjà et qui change souvent.
- Les sept règles de Views and Beyond : écrire pour le lecteur, éviter les répétitions inutiles, éviter l'ambiguïté, utiliser une organisation standard, consigner la justification, garder la documentation à jour mais pas trop, et la revoir pour vérifier qu'elle remplit son objectif.

- La documentation en tant que code (docs as code) et la documentation vivante conservent la documentation en gestion de versions, la revoient avec les changements de code et la tirent du système quand c'est possible, ce qui est la défense la plus fiable contre l'obsolescence.
- Les anti-patrons courants sont le wiki périmé, le document monolithique que personne ne lit, les diagrammes sans légende, les niveaux d'abstraction mélangés et les descriptions qui omettent le pourquoi.

Pourquoi une architecture doit être mise par écrit

Tout système a une architecture, que quelqu'un la décrive ou non (voir la section 1.1). La description d'architecture est autre chose : les documents, diagrammes, modèles et enregistrements de décision par lesquels les gens partagent une compréhension de cette architecture. Dans une équipe de trois personnes qui travaillent côte à côte et ont écrit chaque ligne, une grande partie de la description peut rester dans leur tête. Dès que l'équipe grandit, que des personnes partent, qu'une deuxième équipe arrive, qu'un auditeur se présente ou que le système survit à ses auteurs, un savoir qui n'existe que dans les têtes devient un risque. La définition de l'architecture donnée par Ralph Johnson, la compréhension partagée des développeurs experts, contient un avertissement : une compréhension qui n'est pas écrite ne peut se partager qu'en parlant, et la parole ne passe pas à l'échelle.

Documenting Software Architectures: Views and Beyond (Paul Clements et ses collègues du Software Engineering Institute, 2e édition, 2010) cite trois usages de la documentation d'architecture, et ils restent la façon la plus claire de comprendre pourquoi elle vaut l'effort. Elle **forme** : elle fait découvrir le système aux nouveaux membres de l'équipe, aux relecteurs externes et aux nouveaux architectes. Elle est le principal vecteur de **communication entre parties prenantes** : développeurs, exploitants, équipes de sécurité et managers débattent du système à travers elle, et les décisions prises dans une réunion parviennent à ceux qui n'y étaient pas. Enfin, elle est la **base de l'analyse et de la construction** : elle indique à ceux qui implémentent ce qu'ils peuvent faire et ne pas faire, et c'est ce qu'examine un évaluateur qui se demande si le système atteindra ses objectifs de qualité (les méthodes d'évaluation du chapitre 11 ont besoin d'une description pour travailler).

Deux conséquences en découlent. D'abord, la documentation est un moyen, pas une fin : un document qui ne sert aucun de ces usages est du gaspillage, aussi bien mis en forme soit-il. Ensuite, c'est le lecteur qui juge si elle a fonctionné. Une description que son auteur trouve complète mais dont un nouveau venu ne peut rien faire a échoué sur le plan de la formation, et un diagramme que l'équipe d'exploitation ne peut pas faire correspondre à de vrais serveurs a échoué sur le plan de l'analyse.

Qui lit la documentation d'architecture, et ce dont chacun a besoin

Lecteur	Ses questions	Ce qui le sert le mieux
Nouveaux venus (nouveaux développeurs, prestataires, nouvel architecte)	Que fait ce système, pour qui, et par où commencer ? Quelles sont les principales parties et les mots qu'on emploie ?	Une vue d'ensemble d'une page, un diagramme de contexte système, un diagramme de conteneurs, un glossaire et un court ordre de lecture
Développeurs actuels	Où placer cette nouvelle fonctionnalité ? De quoi puis-je dépendre ? Pourquoi est-ce construit ainsi, et puis-je le changer ?	Structure des modules ou des composants, conventions, concepts transversaux (gestion des erreurs, sécurité, journalisation) et enregistrements de décision
Équipes d'exploitation et SRE	Où cela tourne-t-il, de quoi cela dépend-il, comment cela tombe-t-il en panne et comment rétablir le service ?	Vue de déploiement par environnement, dépendances externes, runbooks (procédures d'exploitation), supervision et objectifs de reprise
Équipes de sécurité et de protection des données	Où sont les frontières de confiance ? Où les données à caractère personnel circulent-elles et où sont-elles stockées ? Qui peut appeler quoi ?	Vues de contexte et de flux de données, concept d'identité et d'accès, liste des services tiers
Auditeurs et régulateurs	Pouvez-vous montrer que les contrôles existent, que les décisions ont été prises délibérément et que les changements sont traçables ?	Enregistrements de décision avec dates et responsables, traçabilité des exigences vers la conception et les tests, historique des versions
Futurs mainteneurs	Mais pourquoi diable a-t-on fait cela ? Quelles contraintes s'appliquent encore ? Que peut-on supprimer sans risque ?	La justification avant tout : les décisions avec leur contexte, les options rejetées et les hypothèses retenues
Décideurs (sponsors, responsables produit et responsables de l'ingénierie)	Quels sont les risques, les coûts et les options ? Que dois-je décider, et pour quand ?	Une synthèse d'une page, les objectifs de qualité, les principaux risques et la dette technique, les décisions en attente d'un responsable

Des logiciels opérationnels plus qu'une documentation exhaustive

Le Manifeste agile (2001) énonce quatre paires de valeurs, dont l'une est «Des logiciels opérationnels plus qu'une documentation exhaustive». On le cite souvent comme une permission de ne rien documenter du tout. Lu en entier, le manifeste dit quelque chose de plus

restreint. Ses auteurs précisent que, si les éléments de droite ont de la valeur, ils valorisent davantage ceux de gauche. Cette valeur fixe une priorité pour les cas où les deux entrent en concurrence ; elle ne rejette pas la documentation.

Ce que cette valeur rejette, c'est le projet piloté par la documentation des années 1990 : des centaines de pages de spécifications validées avant qu'une ligne de code soit écrite, considérées comme un progrès en soi, puis abandonnées dès que le code s'en écarte. Le mot qui compte est *exhaustive* : une documentation qui cherche à tout décrire, pour elle-même, et à laquelle on accorde plus de valeur qu'au système en fonctionnement. Rien dans le manifeste ne s'oppose à un diagramme de contexte, à un court enregistrement expliquant pourquoi l'équipe a choisi la messagerie asynchrone, ou à un runbook qui permet à l'ingénieur d'astreinte de rétablir le service à trois heures du matin.

Les praticiens ne s'accordent pas sur jusqu'où aller. Certains coachs agiles soutiennent que la conversation et les tests suffisent presque toujours, et que la documentation d'architecture écrite devient obsolète plus vite qu'elle ne rapporte. D'autres, en particulier ceux qui maintiennent des systèmes à longue durée de vie ou travaillent dans des secteurs réglementés, soulignent que les personnes qui ont le plus besoin de la documentation (les successeurs, les auditeurs, l'équipe d'exploitation) sont justement celles qui sont absentes des conversations. Les deux positions s'accordent sur l'économie sous-jacente : la documentation a un coût d'écriture, un coût de mise à jour et un coût de lecture, et elle ne vaut la peine d'être produite que là où sa valeur pour les lecteurs dépasse ces coûts.

Juste assez de documentation

L'expression de Scott Ambler « just barely good enough » (tout juste suffisant) décrit la cible : un modèle ou un document est suffisant lorsqu'il remplit son objectif pour son public, et tout effort au-delà coûte plus qu'il ne rapporte. Son idée que c'est le public, et non l'auteur, qui décide si un artefact est suffisant est le même principe que celui d'écrire pour le lecteur. George Fairbanks défend la même idée pour le travail d'architecture dans son ensemble dans *Just Enough Software Architecture* (2010) : en faire autant que les risques le justifient (voir la section 1.8).

La documentation juste suffisante n'est donc pas une quantité fixe. Elle croît avec la taille et la durée de vie du système, le nombre d'équipes, la rotation du personnel, le coût d'une défaillance et les exigences des régulateurs. Un prototype à deux personnes a besoin d'un README et peut-être d'un diagramme. Une plateforme de paiement exploitée par huit équipes sous réglementation financière a besoin de vues de contexte, de conteneurs et de déploiement, d'un journal des décisions remontant sur des années, de concepts de sécurité documentés et de preuves que les documents sont revus.

Un signal pratique est la question qui revient. Si les nouveaux venus posent la même question dès leur première semaine, ou si le même débat de conception ressurgit tous les quelques mois, la réponse doit être écrite. Un autre est l'erreur coûteuse : si une équipe a enfreint une règle non écrite (« ne jamais appeler le prestataire de paiement depuis l'app restaurant ») parce que personne ne la lui avait dite, cette règle relevait de l'architecture et aurait dû être écrite.

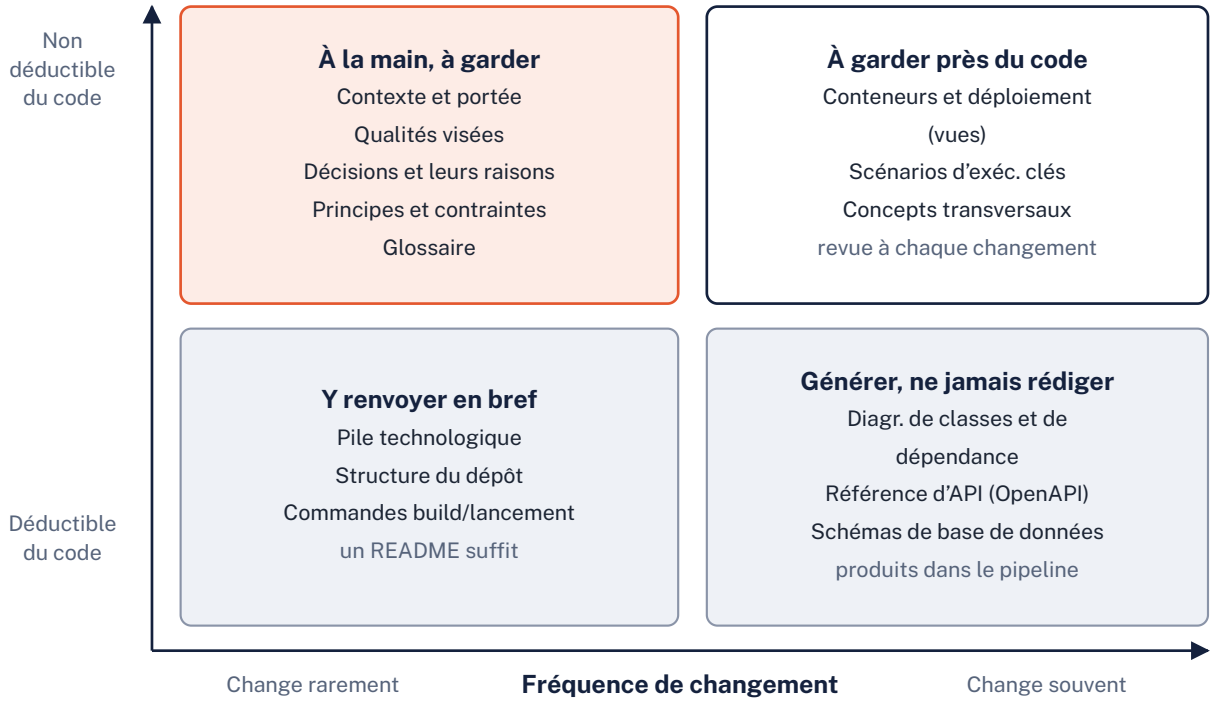
Ce qui vaut la peine d'être documenté, et ce qui ne le vaut pas

Les informations diffèrent de deux façons qui déterminent comment les documenter. La première est de savoir si elles peuvent être **retrouvées à partir du code**. Les noms de classes, les signatures de méthodes et les tables de base de données se lisent dans le code source à tout moment ; les raisons d'une décision, les objectifs de qualité que sert le système et les options rejetées ne le peuvent pas, car elles n'ont jamais été dans le code. La seconde est **leur fréquence de changement**. La finalité du système et ses principaux partenaires externes changent rarement ; la liste des classes d'un module change chaque semaine.

La documentation d'architecture écrite à la main doit se concentrer sur ce qui est à la fois important et impossible à retrouver dans le code : le **contexte et la portée** du système (qui l'utilise et de quels autres systèmes il dépend), ses **objectifs de qualité** et ses contraintes, les **décisions clés et leurs raisons**, la **structure** à gros grain (les principales parties déployables et leurs responsabilités), le **déploiement** sur l'infrastructure, les **scénarios d'exécution** importants, et les **concepts transversaux** que chaque partie doit suivre, comme la façon de gérer les erreurs, d'authentifier les utilisateurs et de protéger les données à caractère personnel. Cette liste est proche de la structure du modèle arc42, traité à la section 4.6.

Ce qui ne vaut pas la peine d'être écrit à la main, c'est tout ce que le code ou les outils indiquent déjà précisément et qui change souvent : les diagrammes de classes d'une base de code entière, les listes d'endpoints que contient déjà un fichier OpenAPI, ou les descriptions de chaque fonction. Lorsque de telles vues sont utiles, générez-les à partir du code source dans le pipeline de build pour qu'elles ne puissent pas dériver. Omettez aussi le contenu aspirationnel (des diagrammes de ce à quoi le système devrait ressembler, présentés comme s'il y ressemblait déjà), les comptes rendus de réunions dont le résultat est consigné ailleurs, et le texte dupliqué, qui finira par contredire sa copie.

Décider comment documenter une information



Écrivez à la main ce que le code ne peut pas vous dire ; générez ce que le code dit déjà et qui change souvent.

SI VOUS DÉBUTEZ

Pas à pas : l'écrire, le générer ou y renvoyer

L'idée qui demande le plus d'attention ici consiste à décider, pour chaque élément de connaissance sur un système, si une personne doit l'écrire, si un outil doit le générer ou si un court renvoi suffit. Deux questions tranchent : peut-on le lire dans le code, et à quelle fréquence change-t-il ? Essayez avec cinq faits sur la plateforme de livraison de repas.

- Pourquoi les événements de commande sont envoyés de façon asynchrone plutôt que par appels directs. Le code ne dit pas pourquoi, ni ce qui a été rejeté, et les raisons changent rarement. Une personne les écrit une fois, sous forme d'enregistrement de décision d'architecture.
- La règle « ne jamais appeler le prestataire de paiement depuis l'app restaurant ». Le code montre que l'app n'appelle pas le prestataire aujourd'hui, pas qu'elle ne doit jamais le faire. Une équipe qui n'a jamais entendu parler de la règle pourrait l'enfreindre ; elle est donc écrite elle aussi.
- La liste des endpoints du service de commandes. Son fichier OpenAPI les indique déjà précisément, et ils changent souvent ; le pipeline génère donc la référence à partir de ce fichier.
- Le diagramme des principales apps, services et bases de données de la plateforme et de leurs connexions. Aucune base de code ne montre à elle seule cette vue d'ensemble, et pourtant elle change chaque fois qu'une partie est ajoutée ou supprimée. On la conserve donc dans le dépôt, idéalement générée à partir d'un modèle qui s'y trouve, et la pull request qui ajoute ou supprime une partie la modifie aussi.
- Les commandes pour construire et lancer chaque service. Elles figurent dans les fichiers de build et changent rarement ; un court README qui y renvoie suffit donc.
- **La leçon :** consacrez l'effort d'écriture manuelle là où le code est muet : les raisons, les règles, les objectifs et la vue d'ensemble. Générez ce que le code indique déjà, ou renvoyez-y : les copies faites à la main dérivent.

Sept règles pour une bonne documentation

Views and Beyond énonce sept règles pour une bonne documentation. Elles valent la peine d'être connues par cœur, car presque tous les échecs de documentation enfreignent l'une d'elles.

1

1. Écrire pour le lecteur

Une documentation s'écrit une fois et se lit de nombreuses fois ; optimisez-la donc pour le lecteur. Décidez qui sont les lecteurs et ce qu'ils chercheront à faire avec le document, organisez-le pour la consultation plutôt que pour une lecture linéaire, et évitez le jargon interne. Signe que cette règle est enfreinte : le document est organisé dans l'ordre où l'auteur a pensé aux choses.

2

2. Éviter les répétitions inutiles

Chaque information doit avoir un seul emplacement, avec des liens depuis ailleurs. La répétition allonge les documents et, pire, laisse les copies diverger jusqu'à ce que les lecteurs ne sachent plus laquelle est vraie. Une répétition délibérée pour le confort du lecteur, comme une synthèse, est acceptable si elle est clairement signalée.

3

3. Éviter l'ambiguïté

Un diagramme de boîtes et de traits sans légende peut se lire de plusieurs façons : un trait est-il un appel, un flux de données ou une dépendance de déploiement ? Définissez la notation dans une légende, utilisez un seul nom pour une seule chose, et précisez ce que signifie chaque élément et chaque relation.

4

4. Utiliser une organisation standard

Une structure connue (un modèle comme arc42 ou le modèle de Views and Beyond) indique aux lecteurs où chercher et montre aux auteurs ce qui manque. Une section vide marquée « sans objet, parce que… » est plus informative qu'un silence.

5

5. Consigner la justification

Écrivez le pourquoi, y compris les options rejetées. Les futurs mainteneurs peuvent généralement comprendre ce qui a été construit ; ils ne peuvent pas comprendre pourquoi, et sans le pourquoi, soit ils craignent de changer quoi que ce soit, soit ils défont une décision dont les raisons tiennent toujours. Les enregistrements de décision existent pour cela.

6

6. La tenir à jour, mais pas trop

Une documentation périmée induit en erreur. Mais mettre à jour chaque document pour chaque idée à moitié formée crée de l'agitation inutile ; consignez les décisions une fois qu'elles sont prises et mettez à jour la description à un rythme prévisible, par exemple à chaque version ou à chaque changement important.

7. **La revoir pour vérifier qu'elle remplit son objectif**
   Seuls les lecteurs visés peuvent dire si un document fonctionne. Demandez à un nouveau venu d'installer et d'expliquer le système à partir des documents, ou à un ingénieur d'astreinte d'utiliser la vue de déploiement lors d'un exercice, et corrigez ce sur quoi ils butent.

Documentation vivante et documentation en tant que code

L'échec traditionnel de la documentation d'architecture est l'obsolescence : la description est écrite au début, le système change, et personne ne met à jour le document parce qu'il vit à l'écart du travail. Deux idées liées s'attaquent à ce problème à la racine.

La documentation en tant que code (docs as code), une pratique promue par la communauté Write the Docs, consiste à traiter la documentation comme du code source : écrite dans des formats texte comme Markdown ou AsciiDoc, conservée en gestion de versions (généralement dans le même dépôt que le code qu'elle décrit), modifiée par les mêmes pull requests, revue par les mêmes personnes, et construite et publiée par le même pipeline. On peut alors exiger qu'une pull request qui modifie la façon dont les commandes sont affectées aux livreurs mette à jour la vue d'exécution dans le même commit, et le relecteur voit les deux.

La documentation vivante, terme popularisé par le livre de Cyrille Martraire qui porte ce nom (2019), va plus loin : chaque fois que c'est possible, la documentation doit être tirée du système lui-même pour ne pas pouvoir dériver. Des diagrammes de dépendances générés à partir du code, des références d'API générées à partir de fichiers OpenAPI, des règles d'architecture vérifiées par des tests automatisés et des diagrammes générés à partir d'un modèle conservé à côté du code en sont des exemples. Le texte écrit à la main est réservé à ce qui ne peut pas être dérivé, principalement l'intention et la justification.

Aucune de ces idées ne dispense de juger de ce qu'il faut écrire. La section 4.7, Diagrammes en tant que code, présente en détail les outils et les habitudes.

GUIDE DE DÉCISION

Guide de décision : faut-il écrire ceci ?

Posez ces questions dans l'ordre à propos d'un élément de connaissance architecturale.

Q1 **Quelqu'un commettrait-il une erreur coûteuse, ou poserait-il sans cesse la même question, si ceci n'était pas écrit ?**

- **Oui** Continuez : c'est un candidat à la documentation.
- **Non** Laissez-le à la conversation et au code pour l'instant ; revenez-y si la question commence à revenir.

Q2 **Peut-on le lire de façon fiable dans le code, la configuration ou un outil ?**

- **Oui** Générez-le, ou renvoyez à l'endroit où il se trouve ; ne le recopiez pas à la main.
- **Non** Continuez : il doit être écrit par une personne.

Q3 **S'agit-il d'une décision avec des raisons et des options rejetées ?**

- **Oui** Rédigez un enregistrement de décision (voir la section 4.5).
- **Non** Continuez.

Q4 **Changera-t-il avec la plupart des versions ?**

- **Oui** Conservez-le dans le dépôt à côté du code qu'il décrit et faites de sa mise à jour une partie de la pull request.
- **Non** Placez-le dans la vue d'ensemble de l'architecture ou dans la section de modèle appropriée, et revoyez-le à un rythme fixe.

EXEMPLE COMMENTÉ

Exemple commenté : une documentation juste suffisante pour la plateforme de livraison de repas

Une entreprise de livraison de repas exploite une plateforme comprenant une app client, une app restaurant, une app livreur, des services de commandes, d'affectation des livreurs et de notifications, et un prestataire de paiement externe. Elle compte 30 ingénieurs répartis en cinq équipes et se prépare à un audit par un évaluateur de la sécurité des paiements par carte. L'architecte propose l'ensemble documentaire suivant, entièrement conservé dans un dépôt de documentation unique ou à côté du code, et dont aucun élément n'est plus long que nécessaire.

Vue d'ensemble de l'architecture (2 à 3 pages)	Finalité, principaux utilisateurs, objectifs de qualité par ordre de priorité (par exemple : ne jamais perdre ni facturer deux fois une commande ; confirmation de commande en moins de 1,5 seconde au p99 en période de pointe ; app restaurant utilisable avec une mauvaise connexion), contraintes clés, et un ordre de lecture pour les nouveaux venus. Responsable : l'architecte. Revue chaque trimestre.
Diagrammes de contexte système et de conteneurs	Deux diagrammes C4 (section 4.3) avec légende, générés à partir d'un modèle du dépôt. Mis à jour dans la pull request qui ajoute ou supprime un conteneur ou une dépendance externe.
Journal des décisions (decision log)	Environ 40 enregistrements de décision courts, numérotés, dans le dépôt : événements de commande asynchrones, choix du prestataire de paiement, une base de données par service, etc. Jamais réécrits ; remplacés lorsqu'une décision change.
Vue de déploiement	Un diagramme et un tableau par environnement : régions, clusters, bases de données managées, courtier de messages (message broker) et frontière réseau autour des données de carte. Sous la responsabilité de l'équipe plateforme.
Scénarios d'exécution	Des diagrammes de séquence pour les quatre flux les plus importants : passer commande, refus par le restaurant, affectation d'un livreur, remboursement. Chacun tient sur un écran.
Concepts transversaux	Une page pour chacun : authentification, idempotence et nouvelles tentatives, journalisation et données à caractère personnel. Ce sont les règles que chaque équipe doit suivre.
Runbooks	Un par alerte susceptible de réveiller quelqu'un, avec un lien depuis l'alerte elle-même.

Non écrit à la main	Diagrammes de classes, listes d’endpoints et schémas de base de données (générés dans le pipeline) ; comptes rendus de réunion ; l’histoire de la croissance du système (le journal des décisions et l’historique des versions la racontent déjà).

Chaque élément a un lecteur identifié, un responsable et un déclencheur de mise à jour. La documentation sans responsable est la première à se dégrader.

L’audit s’appuie sur le journal des décisions, la vue de déploiement et l’historique des versions, et non sur un document d’audit séparé qu’il faudrait maintenir cohérent avec eux.

L’ensemble se lit en deux heures environ, ce qui rend plausible que les nouveaux venus le lisent vraiment.

ATTENTION

Anti-patrons de documentation

Voici les échecs les plus fréquents en pratique, et la règle que chacun enfreint.

- **Le wiki périmé.** Des centaines de pages, modifiées pour la dernière fois il y a des années, en partie vraies, sans qu'on puisse dire quelle partie. Les lecteurs apprennent à se méfier de tout. Remède : moins de pages, chacune avec un responsable et une date de « dernière revue », et supprimer ce que personne ne maintient.
- **Le document monolithique.** Une spécification d'architecture de 120 pages rédigée en amont, approuvée une fois et jamais relue. Il enfreint la règle d'écrire pour le lecteur et devient obsolète en quelques mois.
- **Les diagrammes sans légende.** Des boîtes de différentes couleurs et des traits de différents styles sans aucune légende. Chaque lecteur invente une signification, et deux lecteurs en inventent deux différentes.
- **Les niveaux d'abstraction mélangés.** Un seul diagramme qui montre une app mobile, une classe Java, un cluster Kubernetes et un processus métier. Utilisez des vues distinctes (section 4.2).
- **La documentation uniquement en diapositives.** Les présentations sont faites pour une réunion et un public, sont rarement versionnées et ne permettent pas la recherche. Conservez la source de vérité dans le dépôt et construisez les diapositives à partir d'elle.
- **Le quoi sans le pourquoi.** Une description complète de la structure sans aucune justification, si bien que les équipes futures ne peuvent pas distinguer les contraintes essentielles de celles qui sont accidentelles.
- **La documentation en écriture seule.** Produite pour cocher une case de gouvernance, jamais revue par un vrai lecteur, et donc jamais testée pour vérifier qu'elle remplit son objectif.

Questions fréquentes

Notre code est propre et bien testé. N'est-ce pas une documentation suffisante ?
Un code propre et des tests sont une excellente documentation de *ce que* fait le système à un niveau fin. Ils ne montrent pas la vue d'ensemble à travers les services et les dépôts, les objectifs de qualité, les contraintes extérieures au code, ni pourquoi la structure est ce qu'elle est. C'est exactement à cela que sert la documentation d'architecture.

Qui doit rédiger la documentation d'architecture ?
Les personnes qui prennent les décisions et vivent avec, ce qui, dans la plupart des équipes modernes, signifie les ingénieurs autant que les architectes. L'architecte est généralement responsable de la structure d'ensemble et de la cohérence, et les équipes sont responsables des parties qui décrivent leurs services. L'essentiel est que chaque document ait un responsable identifié.

Comment empêcher la documentation de devenir obsolète ?
Gardez-la réduite, gardez-la à côté du code, faites de sa mise à jour une partie de la définition de « terminé » pour les changements qui la concernent, générez ce qui peut l'être, et revoyez-la à un rythme fixe. La section 4.7 présente ces habitudes en détail.

Les portails de développeurs : rendre la documentation facile à trouver

Une documentation que personne ne trouve pourrait tout aussi bien ne pas exister. Un portail de développeurs donne aux ingénieurs un seul endroit où chercher. Backstage, le plus connu (section 10.7), s'articule autour d'un catalogue logiciel construit à partir de fichiers de métadonnées stockés avec le code, de sorte que la propriété est consignée et mise à jour par les revues habituelles. Sa fonctionnalité TechDocs prend le Markdown conservé à côté du code et le publie sous forme de site de documentation, accessible depuis la page du service dans le catalogue. Sur la plateforme de livraison de repas, la page de Commandes pourrait réunir l'équipe propriétaire, ses enregistrements de décision et ses runbooks. Un portail est un index, pas un remède : les pages périmées restent périmées, et en exploiter un a un coût. Une petite équipe peut n'avoir besoin que d'un dépôt bien tenu.

AsyncAPI et les registres de schémas comme documentation des contrats

Les flux d'événements se perdent facilement de vue, parce qu'aucun service ne les montre à lui seul. Un document AsyncAPI (section 6.5) les consigne en JSON ou YAML lisible par machine : les canaux, les applications qui y envoient ou y reçoivent, et les formats de messages. Il fonctionne avec des protocoles comme Kafka, MQTT et AMQP ; vérifiez la version actuelle. Un registre de schémas ajoute l'application des règles. Celui de Confluent, par exemple, vérifie chaque nouvelle version d'un schéma au regard d'une règle de compatibilité, ascendante par défaut, et refuse les versions incompatibles, et Apicurio Registry, en open source, peut aussi stocker des documents OpenAPI et AsyncAPI (section 7.7). Ainsi, la documentation d'`OrderPlaced` ne peut pas s'écarter sans bruit de ce qui est publié. Aucun des deux n'explique ce qu'un champ *signifie* ; pour cela, il faut encore des mots.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Citez les trois usages de la documentation d'architecture identifiés dans Views and Beyond.

La formation (faire découvrir le système), la communication entre parties prenantes, et le rôle de base pour l'analyse et la construction.

▼ « Des logiciels opérationnels plus qu'une documentation exhaustive » signifie-t-il que les équipes agiles ne doivent pas documenter leur architecture ?

Non. Le manifeste précise que les éléments de droite gardent de la valeur ; il fixe une priorité lorsque les deux entrent en concurrence et rejette la documentation produite pour elle-même, pas la documentation utile.

▼ Pourquoi la justification doit-elle être documentée à la main, alors que les diagrammes de classes ne devraient généralement pas l'être ?

La justification ne peut pas être retrouvée à partir du code et change rarement ; elle doit donc être écrite par une personne. La structure des classes peut être lue ou générée à partir du code et change souvent ; une copie dessinée à la main dérive donc vite.

▼ Contre quoi met en garde la règle « tenir la documentation à jour, mais pas trop » ?

À la fois contre une documentation périmée qui induit les lecteurs en erreur et contre l'agitation permanente due à la mise à jour des documents pour chaque idée non arrêtée ; mettez à jour à un rythme prévisible une fois les décisions prises.

▼ À vous : Une équipe de billetterie a un wiki de 80 pages, obsolète, que personne ne lit. Les développeurs demandent par quoi le remplacer et craignent de perdre du temps. Qu'écririez-vous à la main, que généreriez-vous, que supprimeriez-vous, et comment cela resterait-il à jour ?

Écrivez à la main ce que le code ne peut pas vous dire : le contexte, les objectifs de qualité, les principes et les décisions avec leurs raisons, sous forme d'ADR courts. Générez ce que le code indique déjà et qui change souvent, comme les références d'API. Supprimez les pages périmées plutôt que de les rafistoler. Conservez tout dans le dépôt, relu dans la même pull request que le changement de code, afin que tout reste à jour. Organisez le tout autour des questions des lecteurs, celles d'un nouveau venu ou d'un exploitant par exemple, et non autour de l'ancienne structure.

SOURCES

- Manifesto for Agile Software Development: https://agilemanifesto.org/

- Documenting Software Architectures: Views and Beyond, 2nd edition (SEI): https://www.sei.cmu.edu/library/documenting-software-architectures-views-and-beyond-second-edition/
- Just Barely Good Enough models and documents (Scott Ambler, Agile Modeling): https://agilemodeling.com/essays/barelygoodenough.htm
- Docs as Code (Write the Docs): https://www.writethedocs.org/guide/docs-as-code/
- Principles of technical documentation (Gernot Starke, INNOQ): https://www.innoq.com/en/blog/2022/01/principles-of-technical-documentation/
- Backstage: TechDocs: https://backstage.io/docs/features/techdocs/
- Backstage: Software Catalog: https://backstage.io/docs/features/software-catalog/
- AsyncAPI specification: https://www.asyncapi.com/docs/reference/specification/latest
- Confluent: Schema evolution and compatibility: https://docs.confluent.io/platform/current/schema-registry/fundamentals/schema-evolution.html
- Apicurio Registry: https://www.apicur.io/registry/

SECTION 4.2 SUR 4.8 · INTERMÉDIAIRE · 22 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Vues et points de vue : ISO/IEC/IEEE 42010, 4+1, Rozanski et Woods

AVANT DE LIRE

Cette section s'appuie sur :

- 1.1 Définir l'architecture
- 1.5 Parties prenantes et préoccupations

Termes déjà vus qu'elle utilise : Partie prenante, Préoccupation, Description d'architecture

L'IDÉE EN BREF

Une maison n'est jamais décrite par un seul dessin. L'acheteur regarde le plan des pièces, l'électricien le plan de câblage, le plombier le plan de la tuyauterie et l'agent du service d'urbanisme le dessin de la façade. Chaque dessin laisse volontairement de côté la plupart des choses pour pouvoir répondre clairement aux questions d'une personne. L'architecture logicielle fonctionne de la même façon. Au lieu d'un immense diagramme, les architectes dessinent plusieurs vues du même système, chacune pour un groupe de personnes particulier, et ne choisissent que les vues dont ces personnes ont réellement besoin.

CE QU'IL FAUT SAVOIR

- Une vue représente un système sous l'angle de préoccupations particulières ; un point de vue est l'ensemble réutilisable de conventions (préoccupations encadrées, parties prenantes, types de modèles) permettant de construire un type de vue.
- Dans l'ISO/IEC/IEEE 42010:2022, les parties prenantes ont des préoccupations, les points de vue encadrent les préoccupations et régissent les vues, les vues répondent aux préoccupations et se composent de composants de vue régis par des types de modèles, et les correspondances relient des éléments d'une vue à l'autre.
- Le modèle 4+1 de Kruchten (IEEE Software, 1995) utilise des vues logique, des processus, de développement et physique, reliées et validées par un petit ensemble de scénarios.
- Rozanski et Woods définissent sept points de vue (contexte, fonctionnel, information, concurrence, développement, déploiement, opérationnel) et des perspectives, comme la sécurité et les performances, qui s'appliquent à toutes les vues au lieu d'être dessinées comme des vues.
- Le SEI regroupe les vues en catégories module, composants et connecteurs, et allocation ; un ensemble documentaire complet en comprend au moins une de chaque.

- Choisissez l'ensemble minimal de vues en faisant correspondre parties prenantes et préoccupations à des vues, en combinant des vues quand c'est possible et en documentant les zones à risque plus en profondeur que les zones de routine.

Un système, de nombreuses vues

Aucun diagramme unique ne peut décrire un système réel. Un dessin qui montre comment le code de la plateforme de livraison de repas est découpé en modules ne dit rien des serveurs sur lesquels ils tournent ; un dessin des serveurs et des réseaux ne dit rien de ce qui se passe, étape par étape, quand un livreur accepte une course. Si vous essayez de tout mettre sur une seule page, vous obtenez l'affiche d'architecture bien connue : des dizaines de boîtes, des traits qui signifient six choses différentes, et personne capable d'y trouver la réponse à une question précise.

L'idée qui résout ce problème est ancienne et simple. Décrivez le système au moyen de plusieurs vues, chacune montrant un ensemble de structures pour un ensemble de préoccupations, de même qu'un bâtiment est décrit par un plan des pièces, une élévation, un plan électrique et un plan de plomberie. Chacune est une description partielle ; ensemble, elles décrivent le tout. Cette idée sous-tend la norme internationale, le modèle 4+1 de Kruchten, le catalogue de points de vue de Rozanski et Woods, l'approche Views and Beyond du SEI et, sous une forme légère, le modèle C4 de la section suivante. Cette section explique le vocabulaire qu'ils partagent et comment choisir les vues dont un système particulier a besoin.

Rappelez-vous, d'après la section 1.5, qu'une partie prenante est toute personne qui a un intérêt dans le système et qu'une préoccupation est toute question importante pour une partie prenante. Les vues existent pour répondre aux préoccupations. Une vue qui ne répond à la préoccupation de personne n'est que de la décoration.

Le modèle conceptuel de l'ISO/IEC/IEEE 42010:2022

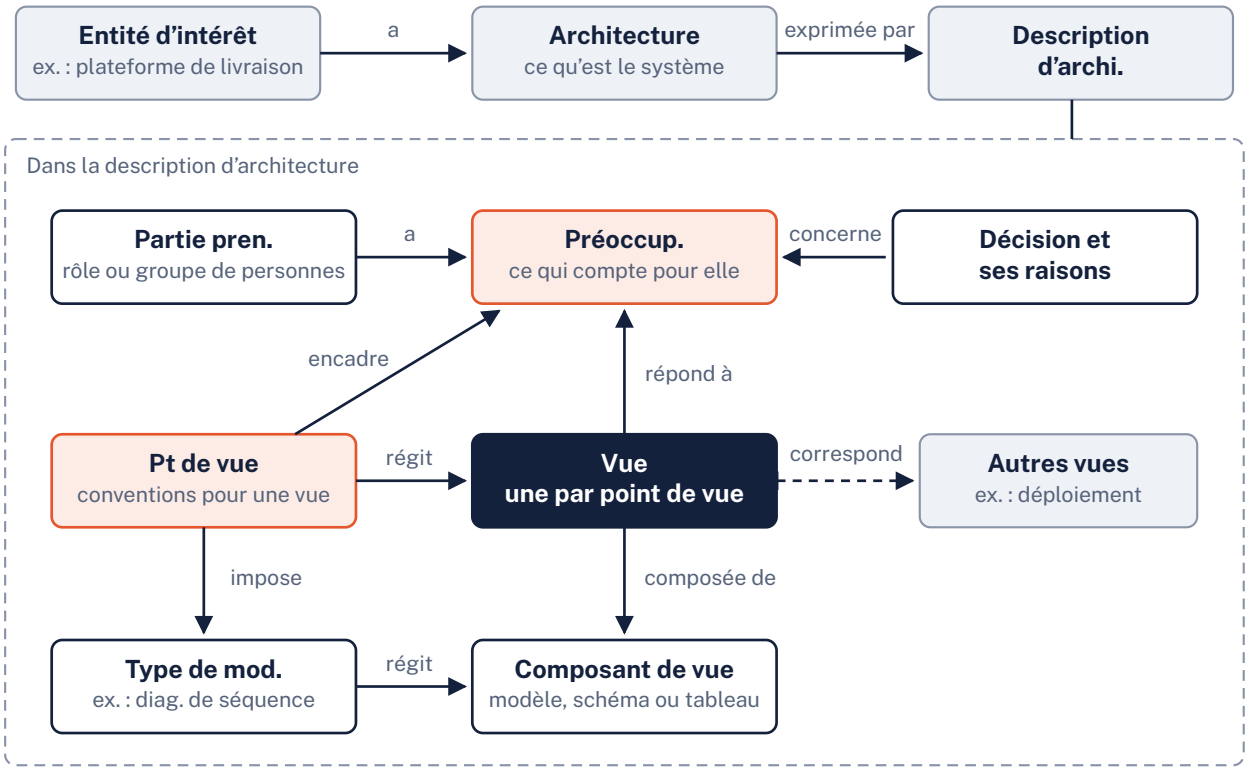
L'ISO/IEC/IEEE 42010 est la norme internationale de description d'architecture. Son édition actuelle a été publiée en 2022 et a remplacé l'édition de 2011. Elle ne vous dit pas comment concevoir, quelle notation utiliser ni quelles vues dessiner ; elle définit les concepts qu'une description d'architecture doit contenir et la façon dont ils se relient, pour que des descriptions produites par des méthodes différentes puissent être comparées et vérifiées. Son modèle conceptuel est le squelette commun de presque toutes les approches de documentation de ce chapitre.

Le modèle part de l'entité d'intérêt : la chose dont on décrit l'architecture, qui peut être un système logiciel, un système de systèmes, une gamme de produits ou une entreprise (l'édition de 2011 l'appelait le système d'intérêt). L'entité a une architecture, et une description

d’architecture l’exprime. Dans la description, les **parties prenantes** ont des **préoccupations**. Un point de vue est une spécification réutilisable de conventions pour construire et utiliser un type de vue : il nomme les préoccupations qu’il encadre, les parties prenantes typiques et les types de modèles (notations et conventions de modélisation) à utiliser. Une **vue** applique un point de vue à une entité particulière : elle répond à ces préoccupations pour ce système, et se compose de composants de vue, comme des diagrammes, des tableaux et des modèles, chacun régi par un type de modèle (l’édition de 2011 appelait les composants de vue des modèles d’architecture).

Deux autres éléments comptent en pratique. Les correspondances consignent les relations entre éléments de la description, le plus utilement entre vues : ce conteneur de la vue des conteneurs tourne sur ce cluster de la vue de déploiement ; cette exigence est traitée par cette décision. Et les **décisions d’architecture et leur justification** font partie de la description, reliées aux préoccupations auxquelles elles répondent. L’édition de 2022 définit aussi plus complètement deux concepts que l’édition de 2011 mentionnait déjà, les perspectives des parties prenantes (des façons de regrouper les préoccupations) et les aspects d’architecture (des caractéristiques de l’entité qui traversent les vues), renomme « architecture framework » en cadre de description d’architecture, et définit ce que doivent contenir les cadres de description d’architecture conformes, comme le modèle de documentation propre à une entreprise.

Le modèle conceptuel de la 42010, simplifié



Les préoccupations sont la charnière : les points de vue les encadrent, les vues y répondent et les décisions les concernent.

SI VOUS DÉBUTEZ

Si vous débutez : point de vue ou vue

Un point de vue est comme un type de carte : « une carte routière » se définit par ses conventions (routes en rouge, autoroutes plus épaisses, une échelle) et par ses utilisateurs (les conducteurs). Une vue est une carte particulière dessinée selon ces conventions : la carte routière de l'Écosse. Vous définissez le point de vue une fois, peut-être pour toute l'entreprise, et vous en dessinez une vue pour chaque système.

Un type de modèle est la technique de dessin que prescrit un point de vue, comme « diagramme de séquence UML » ou « diagramme de conteneurs C4 ». Cette distinction permet à une organisation de dire : la vue de déploiement de chaque système doit montrer les régions, les zones réseau et les services managés, sous forme de diagramme de déploiement C4 accompagné d'un tableau des environnements.

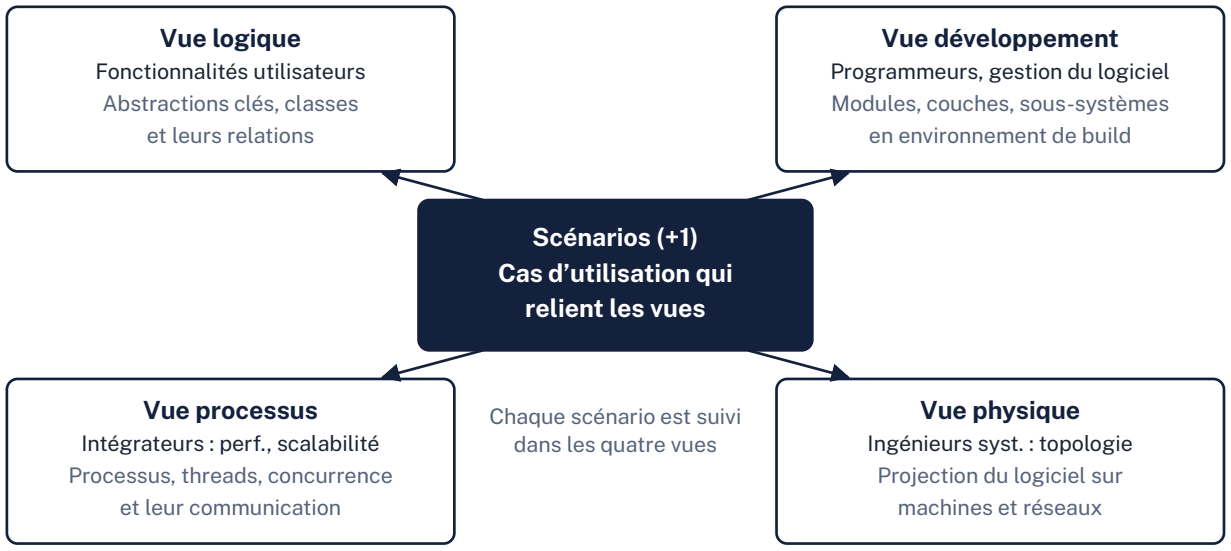
Le modèle de vues 4+1 de Kruchten

Philippe Kruchten, alors chez Rational Software, a publié *Architectural Blueprints: The "4+1" View Model of Software Architecture* dans *IEEE Software* en novembre 1995. Le modèle de vues 4+1 a été la première approche multivue largement adoptée et a été intégré au Rational Unified Process. L'argument de Kruchten était que les diagrammes uniques de l'époque cherchaient à saisir trop de choses à la fois, et que des parties prenantes différentes (utilisateurs finaux, développeurs, ingénieurs système, chefs de projet) ont besoin de vues différentes.

Voici les quatre vues. La **vue logique** décrit les fonctionnalités que le système offre aux utilisateurs finaux, généralement sous forme des abstractions clés et de leurs relations (en UML, des diagrammes de classes et d'états). La **vue des processus** traite des préoccupations d'exécution : processus et threads, concurrence, distribution, performances, scalabilité et tolérance aux pannes. La **vue de développement** (aussi appelée vue d'implémentation) décrit l'organisation du logiciel dans son environnement de développement : modules, sous-systèmes, couches et leurs dépendances, qui comptent pour les programmeurs et pour les managers qui planifient le travail. La **vue physique** (ou vue de déploiement) projette le logiciel sur le matériel et les réseaux. Le « +1 », ce sont les **scénarios** : un petit ensemble de cas d'utilisation importants, chacun suivi à travers les quatre vues, qui à la fois illustrent l'architecture et valident que les vues s'accordent entre elles.

Les scénarios sont la partie la plus souvent oubliée et sans doute la plus précieuse. Une vue isolée peut paraître nette tout en cachant une lacune ; suivre un scénario réel comme « le client passe une commande » à travers les vues logique, des processus, de développement et physique révèle les éléments manquants et les hypothèses incohérentes. La même idée réapparaît dans la vue d'exécution d'arc42, dans les diagrammes dynamiques C4 et, à la section 4.8, dans le conseil d'expliquer un système en suivant une requête.

Le modèle de vues 4+1



Quatre vues pour quatre groupes de parties prenantes, reliées par un petit ensemble de scénarios.

Rozanski et Woods : points de vue et perspectives

L'ouvrage de Nick Rozanski et Eóin Woods *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives* (2e édition, Addison-Wesley, 2012) propose un catalogue de sept points de vue destiné aux grands systèmes d'information, ainsi qu'une seconde idée, complémentaire : les perspectives.

Leurs sept points de vue sont : **contexte** (les relations du système avec les personnes, les systèmes et les entités externes qui l'entourent), **fonctionnel** (les éléments fonctionnels d'exécution, leurs responsabilités, leurs interfaces et leurs interactions principales), **information** (comment les données sont stockées, gérées, possédées et distribuées), **concurrence** (quelles parties s'exécutent en parallèle et comment elles sont coordonnées), **développement** (ce dont ont besoin les développeurs, les testeurs et ceux qui construisent le logiciel : structure des modules, traitements communs, normes et organisation des lignes de code), **déploiement** (l'environnement d'exécution et la façon dont le logiciel s'y projette) et **opérationnel** (comment le système est installé, migré, administré, supervisé et pris en charge en production).

Une perspective est leur nom pour une préoccupation de qualité qui traverse toutes les vues, comme la sécurité, les performances et la scalabilité, la disponibilité et la résilience, ou l'évolution (leur catalogue comprend aussi l'accessibilité, les ressources de développement, l'internationalisation, la localisation, la réglementation et l'utilisabilité). Une perspective n'est pas une vue de plus à dessiner. C'est un ensemble de questions, de tactiques et de vérifications appliqué à chaque vue : pour la sécurité, vous examinez la vue de contexte pour les acteurs

externes et les frontières de confiance, la vue d'information pour les données sensibles, la vue de déploiement pour les zones réseau, et ainsi de suite. Cela saisit ce que des vues isolées ne peuvent pas saisir : les qualités émergent du système entier, elles doivent donc être vérifiées partout (comparez avec les modèles de qualité du chapitre 2).

Le point de vue opérationnel mérite une attention particulière, car de nombreuses approches le sous-estiment. Qui installe le système, comment les données sont migrées depuis l'ancien, comment la configuration est gérée, ce que supervisent les opérateurs et comment ils sont alertés : ce sont des questions architecturales dont les réponses façonnent le système, et les repousser juste avant la mise en service est une cause classique de lancements ratés.

Les trois catégories du SEI : module, composants et connecteurs, allocation

Le chapitre 1 a présenté le regroupement des structures du SEI en trois types (voir la section 1.1). *Documenting Software Architectures: Views and Beyond* utilise le même regroupement pour organiser les vues, et c'est le moyen le plus utile de vérifier qu'un ensemble de vues est complet.

Les vues des modules montrent des unités d'implémentation (paquets, bibliothèques, espaces de noms) et des relations comme *fait partie de*, *dépend de* et *est un*. Les styles typiques sont les vues de décomposition, d'utilisation, de généralisation, en couches et de modèle de données. Les vues composants et connecteurs (C&C) montrent les éléments présents à l'exécution (services, clients, bases de données, files d'attente) et les connecteurs par lesquels ils interagissent (appels HTTP, messages, données partagées) ; les styles typiques sont client-serveur, publication-abonnement, tubes et filtres, données partagées et multiniveau. Les vues d'allocation projettent des éléments logiciels sur des choses non logicielles : sur le matériel et l'infrastructure cloud (déploiement), sur le système de fichiers et les artefacts de build (implémentation et installation) et sur les équipes (affectation du travail).

Les trois catégories correspondent naturellement aux autres approches. La vue de développement de Kruchten est une vue des modules, sa vue des processus une vue C&C et sa vue physique une vue d'allocation. Les points de vue fonctionnel et de concurrence de Rozanski et Woods relèvent surtout du C&C, leur point de vue de développement du module, et leurs points de vue de déploiement et opérationnel de l'allocation. Les diagrammes de conteneurs et de composants C4 de la section suivante sont des vues C&C avec une touche de module, et les diagrammes de déploiement C4 sont des vues d'allocation. Quand vous assemblez un ensemble documentaire, vérifiez qu'il contient au moins une vue de chaque catégorie, car chacune répond à des questions auxquelles les autres ne peuvent pas répondre.

Les approches comparées

Domaine de préoccupation	4+1 (Kruchten)	Rozanski et Woods	Catégorie SEI	Diagramme C4
Frontière du système, utilisateurs, systèmes externes	(implicite dans les scénarios)	Contexte	C&C (contexte)	Contexte système
Éléments d'exécution et interactions	Processus (et logique)	Fonctionnel, concurrence	Composants et connecteurs	Conteneur, composant, dynamique
Propriété et flux des données	(pas de vue séparée)	Information	Module (modèle de données) et C&C	(pas de vue séparée)
Organisation du code	Développement	Développement	Module	Composant, code
Projection sur l'infrastructure	Physique	Déploiement	Allocation	Déploiement
Exploitation du système en production	(non couvert)	Opérationnel	Allocation (en partie)	(non couvert)
Validation par des cas d'utilisation	Scénarios (+1)	(appliqué à chaque vue)	Documentation du comportement	Dynamique

Faire correspondre parties prenantes, préoccupations et vues

L'outil le plus pratique issu de la norme est un simple tableau : listez les parties prenantes et leurs principales préoccupations, puis nommez la ou les vues qui répondent à chaque préoccupation. Les lacunes (une préoccupation à laquelle aucune vue ne répond) montrent où la description est incomplète ; les vues dont personne n'a besoin sont candidates à la suppression. Le tableau ci-dessous est une première version pour la plateforme de livraison de repas.

Parties prenantes, préoccupations et vues pour la plateforme de livraison de repas

Partie prenante	Principales préoccupations	Vues qui y répondent
Responsables produit et sponsors	Périmètre, partenaires externes, ce qui est acheté ou construit, coût	Vue de contexte ; vue d'ensemble d'une page avec les objectifs de qualité et les principaux risques
Développeurs de chaque équipe	Où placer le code, interfaces entre services, conventions	Vue des conteneurs ; vue des composants pour les services complexes ; concepts transversaux ; décisions
Exploitation et SRE	Où les choses tournent, basculement, capacité, dépendances, reprise	Vue de déploiement par environnement ; vue opérationnelle (supervision, runbooks, sauvegarde et restauration)
Équipe sécurité	Frontières de confiance, authentification, données de carte, secrets	Vues de contexte et de déploiement annotées avec les zones de confiance ; concept de sécurité (une perspective appliquée à chaque vue)
Délégué à la protection des données (DPD)	Données à caractère personnel des clients et des livreurs : où elles circulent, combien de temps elles sont conservées	Vue d'information avec un diagramme de flux de données et un tableau de conservation
Équipe d'intégration du prestataire de paiement	Quels appels vous faites, webhooks, idempotence, versions	Spécification d'interface et diagramme de séquence du flux de paiement
Évaluateur de la sécurité des paiements par carte (auditeur)	Périmètre de l'environnement des données de carte, contrôles, historique des changements	Vue de déploiement montrant la frontière des données de carte ; journal des décisions ; historique des versions

Choisir l'ensemble minimal de vues

Une fois que l'on sait qu'il existe de nombreux points de vue, la tentation est de tous les produire. Résistez-y. Chaque vue doit être maintenue cohérente avec le système et avec les autres vues ; chacune ajoute donc un coût de maintenance. Le but est le plus petit ensemble de vues qui réponde aux préoccupations importantes des parties prenantes importantes, selon la règle de la section 4.1 : écrire pour le lecteur. Views and Beyond décrit une procédure semblable : construire un tableau parties prenantes-vues, combiner les vues lorsqu'un seul diagramme peut servir plusieurs besoins, puis prioriser.

Une procédure pour choisir les vues

1. **1. Lister les parties prenantes et les préoccupations**
   Utilisez l'analyse des parties prenantes du chapitre 1. Pour chaque préoccupation, notez son importance et ce qui est en jeu si elle est mal comprise.

2. **2. Associer les préoccupations à des vues candidates**
   Utilisez un catalogue (4+1, Rozanski et Woods, ou les catégories du SEI) comme liste de contrôle pour ne pas oublier des domaines entiers, généralement l'information et l'exploitation.

3. **3. Combiner quand c'est possible**
   Un diagramme de conteneurs annoté avec les technologies et les protocoles sert souvent à la fois les développeurs et l'exploitation ; un diagramme de déploiement avec des zones de confiance sert l'exploitation et la sécurité. Combiner réduit le nombre d'artefacts à maintenir.

4. **4. Vérifier la couverture des trois catégories**
   Assurez-vous que l'ensemble comprend au moins une vue des modules, une vue composants et connecteurs et une vue d'allocation, plus quelques scénarios qui les parcourent.

5. **5. Décider de la profondeur de chaque vue**
   Toutes les vues n'ont pas besoin du même niveau de détail. Documentez en profondeur les parties risquées ou complexes et brièvement les parties de routine.

6. **6. Consigner les correspondances**
   Indiquez comment les éléments d'une vue se rapportent à ceux d'une autre (d'un conteneur à un nœud de déploiement, d'un module à un conteneur) pour que les lecteurs puissent passer d'une vue à l'autre. C'est dans les incohérences entre vues que se cachent de nombreuses erreurs de conception.

EXEMPLE COMMENTÉ

Exemple commenté : l'ensemble minimal de vues pour la plateforme de livraison de repas

L'application de la procédure à la plateforme décrite à la section 4.1 (cinq équipes, paiements par carte via un prestataire externe, une évaluation de la sécurité des paiements à venir) donne l'ensemble suivant.

Vue de contexte	Un diagramme de contexte système C4 : clients, personnel des restaurants, livreurs, agents du support, prestataire de paiement, service de cartographie, services de notifications push. Sert à tout le monde ; c'est la première chose que lit un nouveau venu.
Vue C&C	Un diagramme de conteneurs C4 pour l'ensemble de la plateforme, plus des diagrammes de composants uniquement pour les services de commandes et d'affectation des livreurs, qui sont les plus complexes et changent le plus.
Vue d'information	Un tableau des principales entités de données, du service propriétaire de chacune, des endroits où circulent les données à caractère personnel et des durées de conservation ; un diagramme de flux de données pour les données des clients. Ajoutée parce que les préoccupations du délégué à la protection des données n'étaient autrement pas prises en compte.
Vue d'allocation	Un diagramme de déploiement C4 pour la production, où sont marquées la frontière des données de carte et les zones réseau, et un court tableau des différences dans l'environnement de staging.
Vue des modules	Pas dessinée à la main. La structure interne des modules de chaque service est générée à partir du code et vérifiée par des tests d'architecture.
Scénarios	Quatre diagrammes de séquence : passer commande, refus de la commande par le restaurant, affectation d'un livreur, remboursement. Chacun est vérifié par rapport aux vues de conteneurs et de déploiement.
Perspectives	La sécurité et la disponibilité sont appliquées comme listes de contrôle à chaque vue lors de la revue, au lieu d'être dessinées comme des vues séparées.

Sept artefacts, pour la plupart constitués d'un seul diagramme, répondent aux préoccupations des sept groupes de parties prenantes du tableau ci-dessus.

La vue d'information n'a été ajoutée que parce que le tableau parties prenantes-vues a révélé une lacune. Sans le tableau, elle aurait été oubliée, ce qui est courant.

La vue des modules est générée, et non dessinée, parce qu'elle peut être retrouvée à partir du code et change souvent (section 4.1).

POUR ALLER PLUS LOIN

Pour aller plus loin : bien documenter une vue

Views and Beyond recommande une structure standard pour chaque vue, qui fait aussi une bonne liste de contrôle de revue : une **présentation principale** (généralement le diagramme, qui montre les éléments et les relations), un **catalogue des éléments** (ce qu'est chaque élément, ses responsabilités, ses interfaces et ses propriétés), un **diagramme de contexte** qui montre ce qui se trouve hors de la partie représentée, un **guide de variabilité** (ce qui peut varier, comme des composants optionnels ou la configuration), et la **justification** de la conception présentée.

Il insiste aussi sur la documentation qui se situe *au-delà* des vues, d'où le titre du livre : une feuille de route expliquant comment la documentation est organisée, une vue d'ensemble du système, la correspondance entre vues, la justification des décisions qui traversent les vues, et un répertoire ou un glossaire. La plupart des anti-patrons de la section 4.1 viennent de l'omission de ces parties.

ATTENTION

Erreurs courantes avec les vues

- Mélanger les catégories sur un même diagramme, par exemple dessiner des paquets de code, des services en exécution et des serveurs comme des boîtes sur la même surface, avec des traits qui signifient des choses différentes.
- Dessiner des vues que personne n'a demandées parce qu'un cadre les énumère, tout en manquant la seule vue dont l'équipe d'exploitation a besoin.
- Laisser des vues se contredire : la vue des conteneurs montre six services, mais la vue de déploiement en montre cinq. Les correspondances et les revues détectent ce problème.
- Traiter les attributs de qualité comme une vue de plus au lieu d'une perspective à vérifier dans chaque vue.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Quelle est la différence entre un point de vue et une vue ?**

Un point de vue est une spécification réutilisable de conventions pour un type de vue (préoccupations encadrées, parties prenantes, types de modèles). Une vue applique un point de vue à un système précis et répond à ces préoccupations pour ce système.

▼ **Quel rôle jouent les scénarios dans le modèle 4+1 ?**

Ce sont un petit ensemble de cas d'utilisation importants, suivis à travers les quatre vues, qui illustrent l'architecture et valident que les vues sont cohérentes et complètes.

▼ **En quoi une perspective selon Rozanski et Woods diffère-t-elle d'un point de vue ?**

Un point de vue produit une vue. Une perspective est une préoccupation de qualité transversale, comme la sécurité, appliquée sous forme de questions et de tactiques à chaque vue au lieu d'être dessinée comme une vue à part entière.

▼ **À quelle catégorie du SEI appartient un diagramme de déploiement, et pourquoi ?**

À l'allocation, parce qu'il projette des éléments logiciels sur des choses non logicielles, ici le matériel et l'infrastructure cloud.

▼ **À vous : Pour une plateforme de billetterie, trois lecteurs comptent : les nouveaux développeurs, les opérateurs le jour de la mise en vente et un évaluateur de la sécurité des paiements. Quel ensemble minimal de vues produiriez-vous, et en quoi la préoccupation de chaque lecteur le justifie-t-elle ?**

Une vue de contexte pour tout le monde. Une vue conteneurs partagée par les développeurs et les exploitants, de sorte qu'un seul diagramme serve deux préoccupations. Une vue de déploiement ou d'exploitation pour les exploitants, montrant les zones, la mise à l'échelle et le traitement des pannes. Une perspective de sécurité appliquée à ces vues, montrant les frontières de confiance et les flux de données de carte pour l'évaluateur. Ajoutez une vue d'exécution du parcours d'achat, car c'est la zone à risque. Passez les vues détaillées des parties courantes. Moins il y a de vues, mieux elles restent à jour.

SOURCES

- ISO/IEC/IEEE 42010:2022 Software, systems and enterprise: Architecture description (ISO): https://www.iso.org/standard/74393.html
- ISO/IEC/IEEE 42010 summary (arc42 quality model): https://quality.arc42.org/standards/iso-42010
- Philippe Kruchten, Architectural Blueprints: The 4+1 View Model of Software Architecture (arXiv copy of the 1995 IEEE Software paper): https://arxiv.org/abs/2006.04975
- Viewpoints catalogue, Rozanski and Woods: https://www.viewpoints-and-perspectives.info/home/viewpoints/
- Documenting Software Architectures: Views and Beyond, 2nd edition (SEI): https://www.sei.cmu.edu/library/documenting-software-architectures-views-and-beyond-second-edition/

SECTION 4.3 SUR 4.8 · INTERMÉDIAIRE · 24 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Le modèle C4 : contexte, conteneurs, composants et code

AVANT DE LIRE

Cette section s'appuie sur :

- 0.4 Comment les ordinateurs communiquent
- 0.7 Serveurs et cloud
- 4.2 Vues et points de vue

Termes déjà vus qu'elle utilise : API (interface de programmation d'application), HTTP, Point de vue

L'IDÉE EN BREF

Pensez à une carte en ligne. En vue éloignée, vous voyez un pays entier et les autoroutes entre les villes. Zoomez, et vous voyez les quartiers d'une ville, puis les rues d'un quartier, puis un seul bâtiment. Chaque niveau de zoom est utile à des personnes différentes, et vous savez toujours quel niveau vous regardez. Le modèle C4 fait de même pour le logiciel : une première image montre le système entier et qui l'utilise, la suivante montre les applications et les bases de données qu'il contient, la suivante montre les parties d'une application, et la dernière, rarement nécessaire, montre le code lui-même.

CE QU'IL FAUT SAVOIR

- Le modèle C4, créé par Simon Brown, décrit le logiciel au moyen d'une hiérarchie de quatre diagrammes (contexte système, conteneurs, composants et code) reposant sur les abstractions personne, système logiciel, conteneur et composant.
- Un conteneur C4 est une unité exécutable ou déployable séparément qui exécute du code ou stocke des données, comme une application web, une application mobile, un service, un schéma de base de données ou un topic de messages ; ce n'est pas la même chose qu'un conteneur Docker.
- Les diagrammes de contexte système et de conteneurs sont recommandés pour toutes les équipes ; les diagrammes de composants seulement là où ils apportent de la valeur, et les diagrammes de code rarement, de préférence générés à partir du code.
- Les diagrammes complémentaires sont le paysage système (tous les systèmes d'une organisation), le diagramme dynamique (la collaboration à l'exécution pour un scénario) et le diagramme de déploiement (les conteneurs placés sur l'infrastructure, un par environnement).

- Chaque diagramme a besoin d'un titre indiquant son type et sa portée, et d'une légende ; chaque élément, d'un nom, d'un type, d'une description et (pour les conteneurs et les composants) d'une technologie ; chaque relation, d'un sens unique, d'un libellé précis et, entre conteneurs, d'un protocole.
- Les outils de modélisation comme Structurizr génèrent de nombreux diagrammes cohérents à partir d'un seul modèle, tandis que les outils de dessin produisent des images indépendantes qui divergent peu à peu.

Des cartes de votre code

Le modèle C4 est une approche de la représentation de l'architecture logicielle en diagrammes, créée par Simon Brown. Il est issu des formations en architecture logicielle qu'il donnait depuis 2006 environ : selon le site du C4, ses origines remontent à une période située entre 2006 et 2009, les types de diagrammes ont été nommés début 2010 et le nom « C4 » a été employé pour la première fois début 2011. Brown avait remarqué que les équipes à qui l'on demandait de dessiner leur architecture produisaient des diagrammes difficiles à comprendre : des formes et des couleurs inexpliquées, des boîtes qui pouvaient représenter n'importe quoi, d'une classe à un centre de données, des lignes sans libellé et plusieurs niveaux de détail mélangés sur une même page. UML, qui aurait pu résoudre le problème, était en grande partie sorti de l'usage quotidien.

Sa réponse n'a pas été une nouvelle notation, mais un petit ensemble d'*abstractions* et une hiérarchie de diagrammes construite sur elles, qu'il compare aux niveaux de zoom d'une carte en ligne. Vous commencez par le pays entier, zoomez sur une ville, puis sur un quartier, puis sur une rue. Chaque niveau montre une quantité de détail différente dans un but différent, et vous savez toujours quel niveau vous regardez. C4 est délibérément indépendant de toute notation et de tout outil : vous pouvez le dessiner sur un tableau blanc, dans un outil de dessin ou avec un outil de modélisation, à condition de suivre quelques règles.

C4 est devenu l'une des approches les plus répandues pour les diagrammes d'architecture, parce qu'il est facile à apprendre, qu'il s'accorde naturellement avec la réflexion à plusieurs vues de la section 4.2 et qu'il produit des diagrammes que les développeurs comme les lecteurs non techniques peuvent suivre. Dans les termes de 42010, chaque type de diagramme C4 est un point de vue doté de son propre type de modèle ; C4 se contente de garder cet ensemble restreint.

Les abstractions : personnes, systèmes logiciels, conteneurs, composants

C4 repose sur quatre abstractions, plus le code. Une **personne** est un utilisateur humain du système : un client, un livreur, un agent du support. Un système logiciel est le niveau d'abstraction le plus élevé, quelque chose qui apporte de la valeur à ses utilisateurs et qui est généralement détenu et construit par une seule équipe ou organisation ; la plateforme de livraison de repas en est un, tout comme le prestataire de paiement et le service de cartographie dont elle dépend.

Un conteneur est quelque chose qui doit être en cours d'exécution pour que le système fonctionne : selon les mots de Brown, il s'agit essentiellement d'une frontière d'exécution autour d'un code qui est exécuté ou de données qui sont stockées. Par exemple : une application web côté serveur, une application monopage qui s'exécute dans un navigateur, une application mobile, une application de bureau, une fonction serverless, un schéma de base de données, un stockage de fichiers ou un topic de messages. Chacun est déployable ou exécutable séparément, et les conteneurs communiquent entre eux au-delà des frontières de processus, généralement par un réseau.

Un composant est un regroupement de fonctionnalités liées derrière une interface bien définie, qui vit *à l'intérieur* d'un conteneur et s'exécute dans le même processus que les autres composants de ce conteneur. Dans un service Java, ce peut être un ensemble de classes derrière une interface ; dans un service Node.js, un module. Les composants ne sont pas déployables séparément ; le conteneur est l'unité de déploiement. Enfin, le **code** désigne les classes, les fonctions et les tables qui implémentent un composant.

ATTENTION

Un conteneur n’est pas un conteneur Docker

Le malentendu le plus courant à propos de C4 consiste à lire « conteneur » comme « conteneur Docker ». Le terme C4 est antérieur à la popularité de Docker et désigne toute unité exécutable ou déployable séparément : une application mobile sur un téléphone, une application JavaScript dans un navigateur et un schéma de base de données sont tous des conteneurs, même si aucun d’eux ne tourne dans Docker. À l’inverse, un seul conteneur C4 peut être déployé sous forme de vingt conteneurs Docker derrière un équilibreur de charge ; le nombre de copies en cours d’exécution, et leur emplacement, relèvent d’un diagramme de déploiement, pas d’un diagramme de conteneurs.

Si le mot crée de la confusion dans votre organisation, la plupart des équipes le conservent et l’expliquent dans la légende ou le glossaire. Le site du C4 reconnaît ce conflit, note que les deux notions coïncident parfois utilement, et indique que les équipes peuvent changer la terminologie si nécessaire ; si vous le faites, changez-la partout.

Les quatre diagrammes principaux

Niveau	Portée	Montre	Public principal	Recommandé ?
1. Contexte système	Un système logiciel	Le système sous forme d'une seule boîte, les personnes qui l'utilisent et les autres systèmes logiciels avec lesquels il interagit	Tout le monde, technique ou non, au sein de l'équipe comme à l'extérieur	Oui, pour toutes les équipes
2. Conteneurs	Un système logiciel	Les conteneurs (applications, services, stockages de données, topics) qu'il contient, leurs responsabilités, leurs technologies et leur façon de communiquer	Architectes, développeurs, équipes d'exploitation et de support	Oui, pour toutes les équipes
3. Composants	Un conteneur	Les composants qu'il contient, leurs responsabilités et technologies, et leurs relations	Architectes et développeurs	Seulement là où il apporte de la valeur ; le générer s'il doit être conservé longtemps
4. Code	Un composant	Classes, interfaces, fonctions ou tables, généralement sous forme de diagramme de classes UML ou de diagramme entité-association	Développeurs	Rarement ; le générer à la demande à partir du code pour les parties les plus complexes

SI VOUS DÉBUTEZ

Pas à pas : système, conteneur ou composant ?

Le point qui fait le plus souvent trébucher ici consiste à décider à quelle abstraction C4 correspond une partie d'un système. Les noms se ressemblent : parcourez donc la plateforme de livraison de repas de l'extérieur vers l'intérieur, en vous demandant pour chaque partie si elle peut être déployée et exécutée de façon autonome.

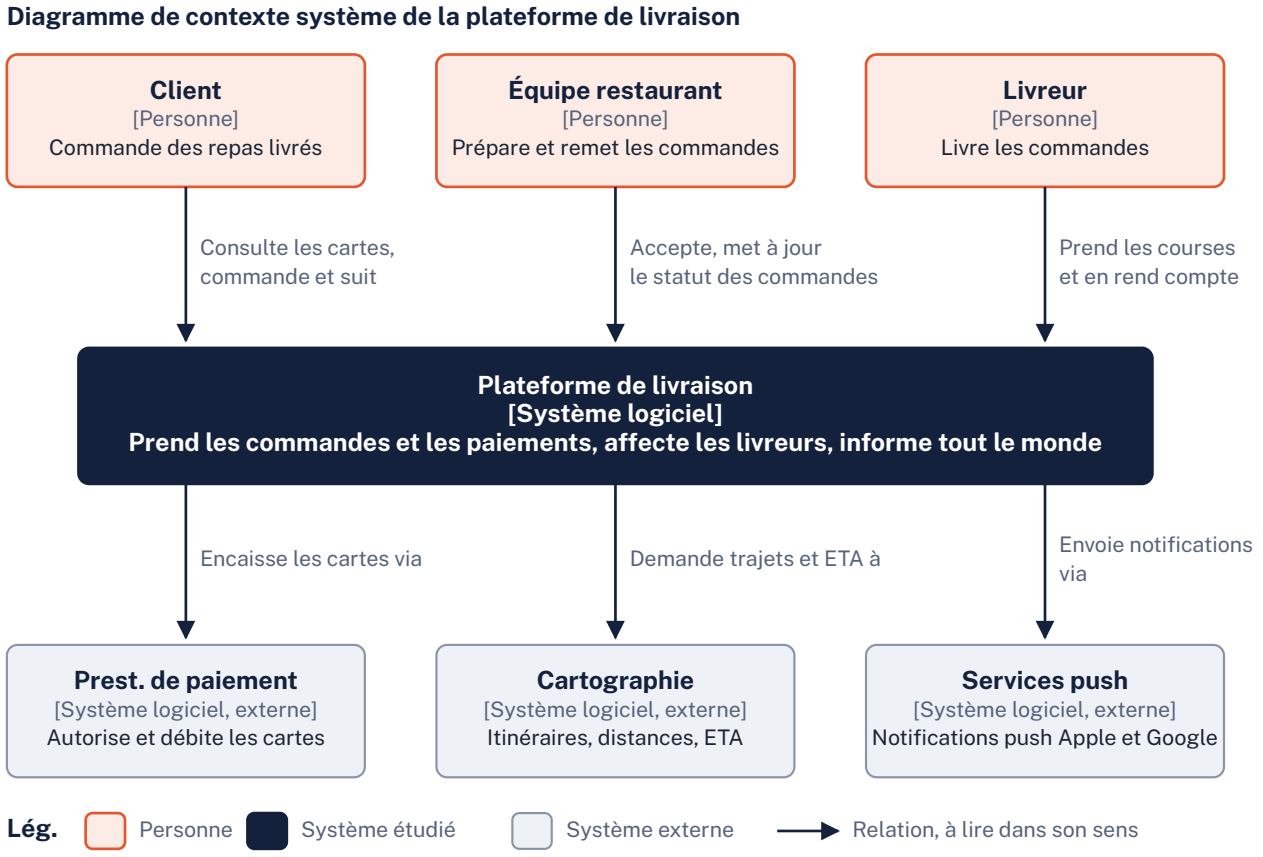
- Commencez par le haut. La plateforme de livraison de repas dans son ensemble est un système logiciel : elle apporte de la valeur aux clients, au personnel des restaurants et aux livreurs. Le prestataire de paiement est un autre système logiciel, construit et détenu par une autre organisation.
- Regardez maintenant à l'intérieur de la plateforme. Le service des commandes s'exécute comme un programme à part entière et communique avec les autres parties par le réseau : c'est donc un conteneur. Il en va de même pour l'app client sur un téléphone et pour l'app restaurant qui s'exécute dans le navigateur d'une tablette.
- Un conteneur n'a pas besoin d'exécuter votre propre code. La base des commandes stocke des données, et la plateforme ne peut pas fonctionner si elle n'est pas en marche : c'est donc aussi un conteneur. Les diagrammes de la section traitent de la même façon le topic des événements de commande, qui contient des messages.
- Zoomez sur le service des commandes. Son flux de commande, qui applique les règles des commandes, et son client de paiement, qui appelle le prestataire de paiement, sont des composants. Ils s'exécutent dans le même processus que le reste du service et sont déployés avec lui ; le client de paiement ne peut pas être déployé seul.
- **La leçon :** une partie qui peut être déployée et exécutée seule est un conteneur ; une partie située à l'intérieur d'un conteneur, qui partage son processus, est un composant ; l'ensemble qui apporte de la valeur à ses utilisateurs, généralement détenu par une seule équipe ou organisation, est un système logiciel. Tranchez cette question avant de dessiner, et chaque boîte se retrouvera au bon niveau de zoom.

Niveau 1 : le diagramme de contexte système

Le diagramme de contexte système est le point de départ. Il place le système que vous décrivez au centre, sous forme d'une seule boîte, et l'entoure des personnes qui l'utilisent et des systèmes logiciels dont il dépend ou qui dépendent de lui. Il masque délibérément tout détail interne, y compris les technologies, les protocoles et les services internes. Il répond aux questions que posent en premier tout nouvel arrivant, tout chef de produit et toute personne chargée de la revue de sécurité : qu'est-ce que ce système, qui l'utilise et à quoi est-il connecté ?

Parce qu'il masque l'intérieur, le diagramme de contexte est aussi le bon diagramme pour les discussions avec les parties prenantes non techniques et pour délimiter le périmètre : où s'arrête notre responsabilité ? Dans l'exemple de livraison de repas ci-dessous, il montre immédiatement que le paiement par carte, le calcul d'itinéraires et l'envoi des notifications push sont achetés à l'extérieur, ce qui est architecturalement significatif (chacun est une dépendance externe, avec sa propre disponibilité, son propre coût et son propre contrat) et invisible dans la plupart des vues au niveau du code.

Plateforme de livraison de repas : contexte système



Niveau 1 : la plateforme sous forme d'une seule boîte, avec ses trois types d'utilisateurs et trois systèmes externes ; chaque élément a un nom, un type et une description, et chaque flèche porte un libellé.

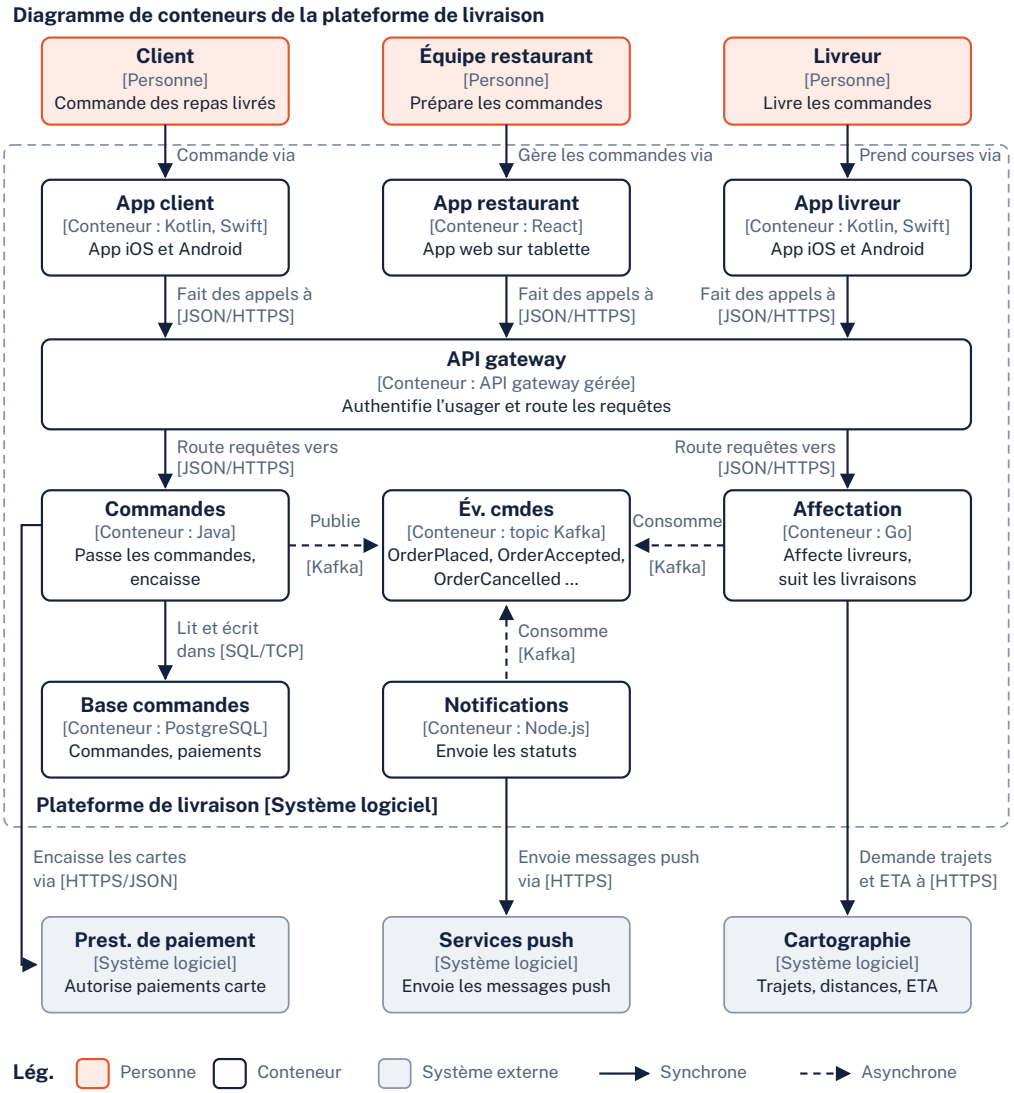
Niveau 2 : le diagramme de conteneurs

Le diagramme de conteneurs zoome sur la boîte du système et montre les conteneurs qu'elle contient : les applications et les stockages de données qui composent le système, la responsabilité de chacun, la technologie que chacun utilise et la façon dont ils communiquent. Pour la plupart des équipes, c'est le diagramme le plus précieux dont elles disposent. Il montre les grandes décisions structurelles (combien d'unités déployables, lesquelles communiquent

de façon synchrone et lesquelles par messagerie, lesquelles possèdent des données) et il est suffisamment concret pour les développeurs et les équipes d'exploitation, tout en tenant sur une seule page.

Remarquez deux choix dans le diagramme ci-dessous. Le topic de messages des événements de commande est dessiné comme un conteneur à part entière, car c'est un stockage dans lequel les producteurs écrivent et que les consommateurs lisent ; les recommandations C4 proposent ce style explicite ou, à défaut, d'omettre le topic et d'écrire « via le topic des événements de commande » sur les flèches. Le dessiner montre clairement le couplage qui passe par le topic. Par ailleurs, les relations asynchrones utilisent une ligne en pointillés, que la légende explique. Le diagramme de conteneurs ne dit rien du nombre de copies de chaque service en cours d'exécution, de leur région ni de l'équilibreur de charge derrière lequel elles se trouvent ; cela relève d'un diagramme de déploiement.

Plateforme de livraison de repas : conteneurs



Niveau 2 : les applications, les services et les stockages de données à l'intérieur de la frontière de la plateforme, avec les technologies sur chaque élément et les protocoles sur chaque flèche.

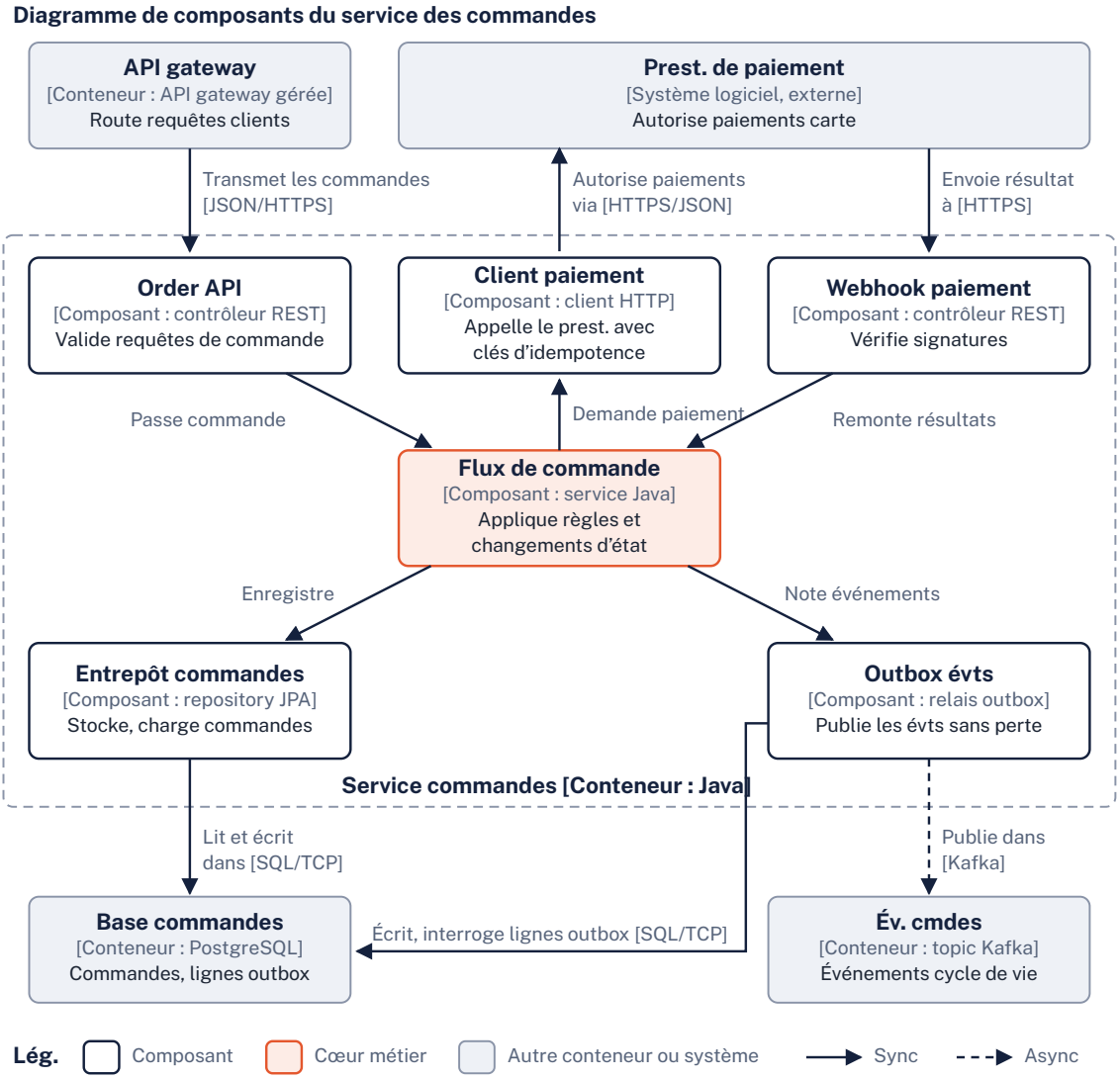
Niveau 3 : le diagramme de composants

Le diagramme de composants zoome sur un conteneur et montre les composants qu'il contient. Il s'adresse aux architectes et aux développeurs qui travaillent sur ce conteneur, et c'est là que la structure interne d'un service devient visible : quelle partie traite les requêtes entrantes, où vivent les règles métier, quelle partie communique avec la base de données, comment les événements sortent du service.

Les recommandations C4 consistent à ne dessiner des diagrammes de composants que là où ils apportent de la valeur, généralement pour les conteneurs complexes, risqués ou modifiés par de nombreuses personnes, et à les générer automatiquement s'ils doivent être conservés longtemps, car la structure des composants change aussi souvent que le code. Pour la

plateforme de livraison de repas, le service des commandes remplit ces conditions : il manipule de l'argent, communique avec un prestataire externe et doit publier des événements de façon fiable. Le diagramme ci-dessous montre une structure typique, y compris un composant outbox qui écrit les événements dans la base de données au sein de la même transaction que la commande, puis les relaie vers Kafka (le patron outbox est traité au chapitre 6).

Service des commandes : composants



Niveau 3 : l'intérieur d'un conteneur, avec les conteneurs et systèmes voisins représentés hors de la frontière en pointillés, afin que chaque relation ait ses deux extrémités.

Niveau 4 : le code, et les diagrammes complémentaires

Le niveau du code zoome sur un composant et montre ses classes, ses interfaces ou ses tables, généralement sous forme de diagramme de classes UML ou de diagramme entité-association. Les recommandations C4 le considèrent comme facultatif et comme ne valant généralement pas la peine d'être dessiné à la main : la plupart des environnements de développement

peuvent le générer à la demande, et une copie dessinée à la main est périmée dès que quelqu'un refactorise. Ne le dessinez que pour les parties les plus importantes ou les plus complexes, comme un modèle de domaine au cœur du métier (la section 4.4 montre un tel diagramme de classes).

Trois **diagrammes complémentaires** comblent les lacunes laissées par la hiérarchie. Un **diagramme de paysage système** est un diagramme de contexte sans système unique au centre : il montre tous les systèmes logiciels et toutes les personnes d'une organisation ou d'un domaine, ce qui est utile aux architectes d'entreprise et de plateforme. Un diagramme dynamique montre comment les éléments collaborent à l'exécution pour un scénario, sous forme d'interactions numérotées sur un diagramme de conteneurs ou de composants (ou sous forme de diagramme de séquence UML) ; c'est l'équivalent C4 des scénarios du modèle 4+1. Un diagramme de déploiement montre comment les conteneurs sont placés sur l'infrastructure dans un environnement : des nœuds de déploiement comme des régions cloud, des clusters Kubernetes, des machines virtuelles, des services de base de données managés et des appareils, avec le nombre d'instances. C4 recommande un diagramme de déploiement par environnement important, généralement la production et éventuellement le staging.

Des règles de notation qui rendent les diagrammes lisibles

C4 n'impose ni formes ni couleurs, mais il impose ce que chaque diagramme doit dire. Ces règles, condensées à partir des recommandations de notation C4, sont ce qui rend les diagrammes ci-dessus lisibles sans réunion pour les expliquer.

Liste de contrôle : un diagramme C4 qui s'explique de lui-même

- [ ] **Un titre indiquant le type et la portée**
  Par exemple « Diagramme de conteneurs de la plateforme de livraison de repas », pour que les lecteurs sachent tout de suite à quel niveau de zoom ils se trouvent.

- [ ] **Une légende**
  Expliquez chaque forme, couleur, style de bordure, style de ligne et pointe de flèche utilisés, même si la notation semble évidente. Les lecteurs d'autres équipes ne partagent pas vos conventions.

- [ ] **Chaque élément a un nom, un type et une description**
  Le type entre crochets (Personne, Système logiciel, Conteneur, Composant) lève toute ambiguïté ; une courte phrase de responsabilité dit à quoi sert l'élément.

- [ ] **Une technologie sur chaque conteneur et chaque composant**
  « Conteneur : Java », « Conteneur : PostgreSQL ». Les choix technologiques font partie de l'architecture, et les lecteurs en ont besoin.

- [ ] **Chaque relation va dans un seul sens et porte un libellé**
  Utilisez une flèche par relation, avec un libellé qui dit ce qu'elle fait dans le sens de la flèche (« Publie les événements de commande dans »), jamais simplement « utilise » ni une flèche à double pointe.

- [ ] **Des protocoles sur les relations entre conteneurs**
  JSON/HTTPS, SQL/TCP, Kafka. Le caractère synchrone ou asynchrone est l'une des informations les plus importantes du diagramme.

- [ ] **Des acronymes expliqués**
  Écrivez-les en toutes lettres ou définissez-les dans la légende ou le glossaire.

- [ ] **Un seul niveau d'abstraction par diagramme**
  Pas de classes sur un diagramme de conteneurs, pas de serveurs sur un diagramme de composants.

- [ ] **La couleur en complément, pas comme seul signal**
  Les diagrammes sont imprimés en noir et blanc et lus par des personnes daltoniennes ; utilisez aussi le style de ligne et les libellés, et gardez le même sens aux couleurs d'un diagramme à l'autre.

EXEMPLE COMMENTÉ

Exemple commenté : améliorer un diagramme d'équipe typique

Une équipe de l'entreprise de livraison de repas partage cette description du diagramme dessiné sur son tableau blanc : cinq boîtes libellées « App », « Backend », « Commandes », « BD » et « Stripe » ; des lignes entre elles sans pointe de flèche ; la boîte Commandes est verte et les autres sont bleues ; il n'y a ni titre ni légende. Appliquons les règles C4 pas à pas :

Titre et niveau	Décidez ce qu'est le diagramme. Il mélange une application mobile, un service et une base de données : c'est donc un diagramme de conteneurs. Donnez-lui le titre « Diagramme de conteneurs de la plateforme de livraison de repas ».
Nom, type, description	« App » devient « App client [Conteneur : Kotlin, Swift] App iOS et Android pour commander des repas ». « Backend » s'avère être l'API gateway (passerelle d'API) ; « BD » devient « Base des commandes [Conteneur : PostgreSQL] ».
Systèmes externes	« Stripe » est un nom de produit qui désigne un système externe. Représentez-le sous la forme « Prestataire de paiement [Système logiciel] » en dehors de la frontière du système, en nommant le fournisseur dans la description si cela compte pour les lecteurs.
Relations	Remplacez chaque ligne simple par une seule flèche avec un libellé : « Appelle [JSON/HTTPS] », « Lit et écrit dans [SQL/TCP] », « Autorise les paiements par carte via [HTTPS] ». L'équipe découvre qu'une même ligne signifiait deux choses différentes (un appel dans un sens et un webhook dans l'autre) : elle devient donc deux flèches.
Éléments manquants	Le fait de libeller les flèches révèle que les événements de commande parviennent au service d'affectation par un topic Kafka qui n'était pas dessiné. Ajoutez le topic comme conteneur, avec des flèches asynchrones en pointillés.
Légende et couleur	La boîte verte s'avère signifier « détenu par notre équipe ». Ajoutez-le dans une légende, ou remplacez la couleur par une frontière en pointillés portant le nom de l'équipe.
Résultat	Le diagramme a gagné un titre, une légende, trois éléments et deux fois plus de flèches, mais il demande moins d'explications, car chaque question qu'un lecteur se poserait trouve sa réponse sur la page.

L'essentiel de la valeur est venu du libellé des flèches : les lignes sans libellé cachent les informations architecturales les plus importantes (le sens, le protocole, le caractère synchrone ou asynchrone).

Le diagramme amélioré a révélé une dépendance (le topic) que l'équipe elle-même avait oubliée, ce qui est un effet secondaire typique de l'application des règles.

ATTENTION

Erreurs courantes avec C4

- Dessiner chaque microservice comme un système logiciel distinct alors qu'une seule équipe les détient tous. Selon les recommandations C4, les services détenus par une même équipe sont des conteneurs à l'intérieur du système de cette équipe ; les services détenus par des équipes distinctes peuvent chacun être modélisés comme un système logiciel.
- Représenter un bus de messages ou un service mesh partagé comme un seul grand conteneur par lequel tout passe, ce qui masque qui dépend réellement de qui. Modélisez chaque topic ou chaque file, ou écrivez-les sur les flèches.
- Dessiner à la main des diagrammes de composants pour chaque conteneur. Ils deviennent vite obsolètes ; dessinez-les de façon sélective ou générez-les.
- Mettre des détails de déploiement (réplicas, équilibreurs de charge, zones de disponibilité) sur le diagramme de conteneurs plutôt que sur un diagramme de déploiement.
- Des flèches libellées seulement « utilise » ou « données », ou des flèches pointant dans les deux sens.
- Omettre les personnes du diagramme de contexte, ce qui supprime la raison d'être du système.

Outils : dessin ou modélisation

Les diagrammes C4 peuvent être produits avec n'importe quel outil de dessin, et de nombreuses équipes commencent par un tableau blanc ou une application générique de création de diagrammes. Le site du C4 distingue deux familles d'outils. Les **outils de diagrammes** dessinent des boîtes et des lignes ; ils sont souples et familiers, mais les diagrammes sont des images indépendantes : renommer un élément oblige à modifier chaque diagramme qui le montre, et rien ne vérifie que les diagrammes concordent. Les **outils de modélisation** contiennent un modèle unique des personnes, des systèmes, des conteneurs, des composants et des relations, et génèrent chaque diagramme comme une vue de ce modèle ; le renommage se fait une seule fois, les diagrammes restent cohérents et le modèle peut être interrogé et validé.

Structurizr, créé par Simon Brown, est l'outil de modélisation de référence pour C4. Son langage textuel, le Structurizr DSL, décrit le modèle et les vues dans un fichier qui vit dans le système de gestion de versions, à côté du code. Parmi les autres options figurent C4-PlantUML (des formes C4 pour PlantUML), la prise en charge des diagrammes C4 par Mermaid et divers outils de modélisation commerciaux ; la section 4.7, Diagrammes en tant que code, en montre des exemples. Le choix compte moins que l'habitude : gardez la source des diagrammes là où l'équipe travaille, et mettez-la à jour dans la même pull request que le changement qu'elle décrit.

Questions fréquentes

C4 remplace-t-il UML ?
Pas exactement. C4 définit des abstractions et des types de diagrammes, pas une notation, et il ne dit rien de nombreux aspects qu'UML peut exprimer précisément, comme les machines à états ou les séquences de messages détaillées. Beaucoup d'équipes utilisent C4 pour la structure et empruntent à UML les diagrammes de séquence, d'états-transitions et de déploiement là où la précision est nécessaire (section 4.4).

Comment les fonctions serverless s'intègrent-elles dans C4 ?
Chaque fonction, ou chaque groupe de fonctions déployées ensemble, est un conteneur : une unité déployable séparément qui exécute du code. Les stockages de données et les files partagés qu'elles utilisent sont aussi des conteneurs.

Notre système est un monolithe déployé d'un seul bloc. Le diagramme de conteneurs est-il inutile ?
Non. Un monolithe a généralement quand même un client web ou mobile, l'application serveur, une base de données et peut-être un stockage de fichiers et une tâche planifiée, chacun étant un conteneur. Le diagramme de composants du monolithe devient alors la vue la plus intéressante de ses modules internes.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Qu'est-ce qu'un conteneur dans le modèle C4 ? Donnez deux exemples qui ne font pas intervenir Docker.**

Une unité exécutable ou déployable séparément qui exécute du code ou stocke des données. Exemples : une application mobile sur un téléphone, une application monopage dans un navigateur, un schéma de base de données, une fonction serverless.

▼ **Quels sont les deux diagrammes C4 que les recommandations C4 conseillent à toutes les équipes ?**

Le diagramme de contexte système et le diagramme de conteneurs.

▼ **Où faut-il indiquer le nombre de réplicas, les régions et les équilibreurs de charge ?**

Sur un diagramme de déploiement, un par environnement, et non sur le diagramme de conteneurs.

▼ **Pourquoi C4 exige-t-il des flèches à sens unique et avec un libellé ?**

Les lignes sans libellé ou à double sens cachent la nature de la relation (appel, flux de données, événement), son sens et le protocole qu'elle utilise, qui font partie des informations architecturales les plus importantes du diagramme.

▼ **À vous : Un diagramme montre les boîtes « Web », « API », « Kafka », « Postgres » et « Paiement », reliées par des lignes sans flèches, sans titre ni légende. Une boîte est rouge. Appliquez les règles de C4 : qu'est-ce qui ne va pas, et que changeriez-vous ?**

Il n'a pas de titre indiquant le type et la portée, ni de légende : personne ne sait donc ce que signifie le rouge. Les boîtes ont besoin d'un nom, d'un type, d'une description et, pour les conteneurs, d'une technologie. Le paiement est un système logiciel externe et doit avoir un aspect différent. Les lignes ont besoin d'un sens unique, d'une étiquette précise et d'un protocole, comme « publie OrderPlaced [Kafka] ». C'est un diagramme de conteneurs : appelez-le ainsi. Ne gardez le rouge que si la légende l'explique, sinon supprimez-le.

SOURCES

- The C4 model for visualising software architecture (Simon Brown): https://c4model.com/
- C4 model: Container (not Docker!): https://c4model.com/abstractions/container
- C4 model: Notation: https://c4model.com/diagrams/notation
- C4 model: System context diagram: https://c4model.com/diagrams/system-context
- Structurizr DSL documentation: https://docs.structurizr.com/dsl

Les diagrammes UML que les architectes utilisent encore

AVANT DE LIRE

Cette section s'appuie sur :

- 0.6 API et intégration
- 4.2 Vues et points de vue
- 4.3 Le modèle C4

Termes déjà vus qu'elle utilise : Interface, Appel synchrone, Vue d'allocation, Inversion des dépendances

L'IDÉE EN BREF

Partout dans le monde, les musiciens savent lire la même partition, parce que les notes et les symboles ont le même sens partout. UML est un ensemble comparable de symboles partagés pour dessiner du logiciel : telle flèche signifie « attend une réponse », telle autre « envoie et continue », un losange plein signifie « est composé de ». Vous n'avez pas besoin de tout connaître. Les architectes utilisent surtout une poignée de ses schémas : l'un montre qui parle à qui et dans quel ordre, un autre les étapes par lesquelles passe quelque chose, un autre l'endroit où tout s'exécute, plus quelques autres, chacun n'étant utilisé que lorsque sa précision aide vraiment.

CE QU'IL FAUT SAVOIR

- UML est un langage de modélisation généraliste normalisé par l'OMG ; la version actuelle est UML 2.5.1 (décembre 2017), qui définit quatorze types de diagrammes répartis en deux familles, structure et comportement.
- Fowler distingue UML comme croquis (communication sélective et informelle), comme plan détaillé (modèles complets et précis) et comme langage de programmation (modèles exécutables) ; les architectes font surtout des croquis.
- Les diagrammes de composants montrent les interfaces fournies (boule) et requises (douille) ; les diagrammes de déploiement montrent les nœuds, les artefacts et les chemins de communication.
- Les diagrammes de séquence montrent des lignes de vie et des activations ; une pointe de flèche pleine indique un appel synchrone, une pointe ouverte sur une ligne continue un message asynchrone, et une ligne en pointillés une réponse ; des fragments comme alt, opt et loop ajoutent le flux de contrôle.

- Les diagrammes d'états-transitions décrivent des cycles de vie avec des états et des transitions libellées événement [garde] / action ; les diagrammes d'activité décrivent des flux de travail avec des décisions, des bifurcations, des jonctions et des couloirs.
- Les diagrammes de classes sont à réserver aux petits modèles de domaine, avec associations, multiplicités et composition ; les diagrammes de classes complets doivent être générés, pas dessinés à la main.
- La précision vaut son prix aux interfaces entre organisations, dans les travaux réglementés et critiques pour la sûreté, pour les cycles de vie partagés par plusieurs équipes et pour le développement distribué.

Ce qu'est UML, et où il en est aujourd'hui

Le langage de modélisation unifié (UML) est un langage graphique généraliste qui sert à décrire la structure et le comportement des logiciels. Il est né de la fusion, chez Rational Software au milieu des années 1990, des méthodes de Grady Booch, James Rumbaugh et Ivar Jacobson, et a été adopté comme norme en 1997 par l'Object Management Group (OMG), un consortium de normalisation du secteur. UML 2.0 (2005) a été une révision majeure. La version actuelle est UML 2.5.1, publiée par l'OMG en décembre 2017 ; la version antérieure, UML 2.4.1, a aussi été publiée par l'ISO sous la référence ISO/IEC 19505 (2012). La spécification définit quatorze types de diagrammes, répartis entre les diagrammes de structure (comme les diagrammes de classes, de composants et de déploiement) et les diagrammes de comportement (comme les diagrammes d'activité, d'états-transitions et de séquence).

La réputation d'UML a beaucoup changé. À la fin des années 1990 et au début des années 2000, il était présenté comme la base du développement dirigé par les modèles, dans lequel des modèles détaillés seraient transformés en code. Cette ambition a largement échoué en dehors de domaines spécialisés, et beaucoup d'équipes associent UML à des processus lourds et à des diagrammes que personne ne lisait. Lorsque Marian Petre a interrogé 50 développeurs professionnels pour une étude présentée à ICSE 2013, 35 ont déclaré ne pas utiliser UML du tout, et la plupart des autres l'utilisaient de façon sélective et informelle ; la plupart des équipes dessinent des boîtes et des lignes informelles plutôt que de l'UML strict. Pourtant, une poignée de types de diagrammes UML restent les meilleurs outils disponibles pour certaines questions, en particulier les questions de comportement dans le temps, et leur notation est largement comprise. Un architecte n'a pas besoin des quatorze types de diagrammes ; il lui en faut peut-être six, utilisés à dessein.

Cette section présente ces six types, chacun avec un petit exemple tiré de la plateforme de livraison de repas utilisée tout au long de ce chapitre. Elle suppose acquis le vocabulaire C4 de la section 4.3 ; les deux se combinent bien, avec C4 pour la vue d'ensemble structurelle et UML là où la précision sur les interfaces, les interactions ou les états est rentable.

Croquis, plan détaillé ou langage de programmation

UML Distilled de Martin Fowler (3e édition, Addison-Wesley, 2003) et ses essais de la même année distinguent trois façons d'utiliser UML, une classification à laquelle Steve Mellor est parvenu de son côté. Comme **croquis**, UML est utilisé de façon informelle et sélective pour communiquer des idées : sur un tableau blanc avant d'écrire du code (croquis prospectif) ou pour expliquer une partie d'un système existant (croquis rétrospectif). Seuls les éléments importants apparaissent, la notation est suivie de façon approximative, et le diagramme est souvent jeté. Fowler le résume lui-même ainsi : «l'exhaustivité est l'ennemie de la compréhensibilité».

Comme **plan détaillé**, UML sert à produire des modèles complets et précis que quelqu'un d'autre implémente, ou qui sont générés à partir du code existant pour le documenter intégralement ; le but est l'exhaustivité, et des outils sont généralement nécessaires. Comme **langage de programmation**, le modèle UML est suffisamment détaillé pour être compilé en code exécutable, ce qui est l'objectif de l'architecture dirigée par les modèles (MDA). Ce mode survit dans quelques niches de l'embarqué et des systèmes critiques pour la sûreté, souvent au moyen de profils et d'outils spécialisés, mais il ne s'est pas généralisé.

Pour les architectes, le mode croquis est la règle et le mode plan détaillé l'exception. La question utile pour chaque diagramme est : qui le lira, et quelle décision ou quelle compréhension soutient-il ? Un diagramme de séquence esquissé pour s'entendre sur la gestion des échecs de paiement est précieux, même s'il ignore la moitié de la notation. Un modèle de classes complet d'un service de 200 classes, maintenu à la main, ne l'est généralement pas.

Diagrammes de composants : les parties et leurs interfaces

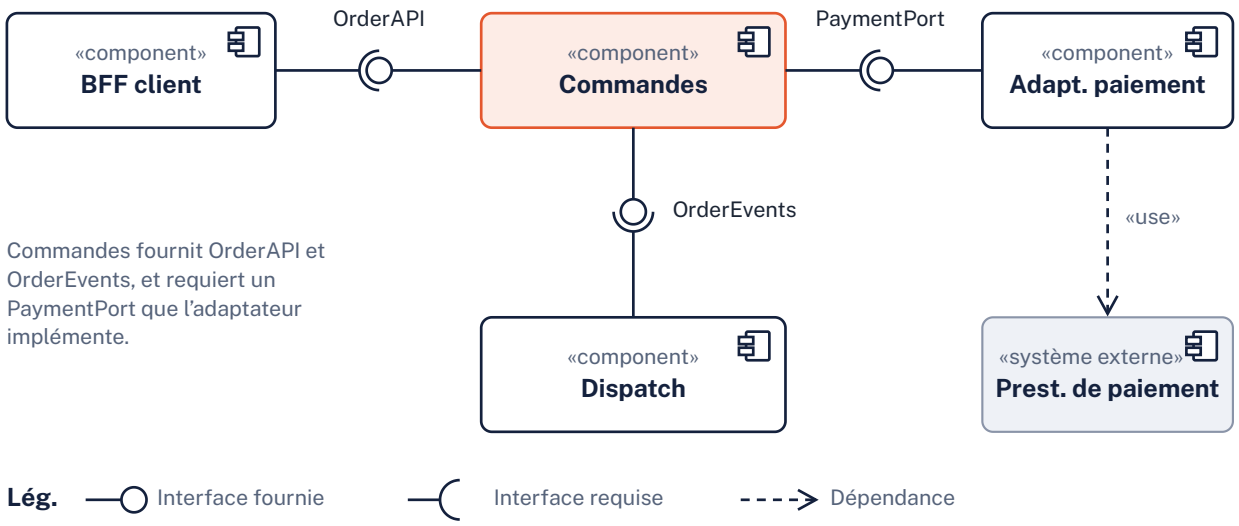
Un diagramme de composants UML montre les parties modulaires d'un système et, surtout, leurs **interfaces**. Sa notation caractéristique est la « boule et douille » : un petit cercle au bout d'une tige (une interface fournie, que le composant offre) et un demi-cercle (une interface requise, qu'il a besoin qu'un autre lui fournisse). Quand une boule s'emboîte dans une douille, les deux composants sont reliés par cette interface. Les dépendances sont des flèches en pointillés, éventuellement libellées avec un stéréotype comme «use».

C'est exactement ce qui intéresse les architectes quand ils veulent montrer le sens des dépendances et la possibilité de remplacer une partie. Dans l'exemple, le composant Commandes ne dépend pas directement du prestataire de paiement ; il requiert une interface `PaymentPort`, qu'implémente un adaptateur de paiement distinct. C'est l'inversion des dépendances et la logique ports et adaptateurs du chapitre 3 rendues visibles : changer de

prestataire de paiement signifie écrire un nouvel adaptateur, et non modifier Commandes. Un diagramme de composants C4 peut montrer les mêmes parties, mais il ne distingue pas les interfaces fournies des interfaces requises.

Diagramme de composants

Diagramme de composants : Commandes et interfaces



Les boules sont les interfaces qu'un composant fournit ; les douilles, les interfaces qu'il requiert. Commandes dépend d'un port, pas du prestataire. (BFF : backend for frontend, un composant côté serveur dédié à une seule app cliente.)

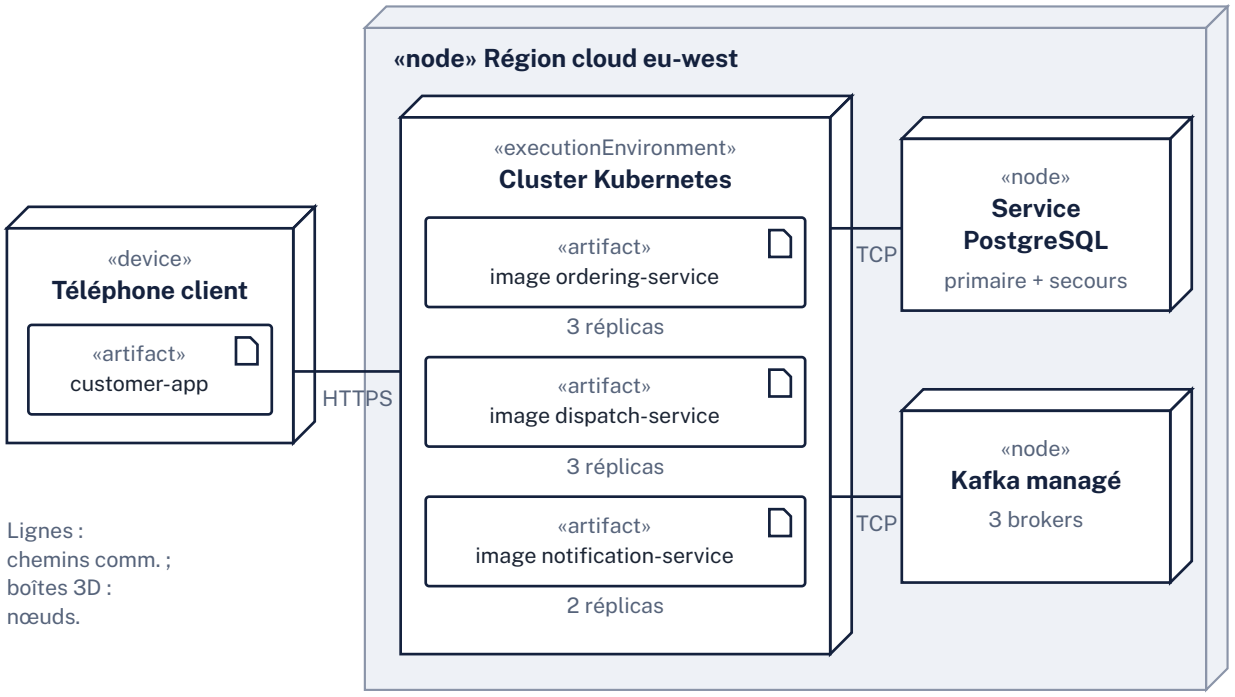
Diagrammes de déploiement : nœuds et artefacts

Un diagramme de déploiement UML montre l'infrastructure physique ou virtuelle et ce qui s'exécute dessus. Ses éléments sont des **nœuds**, dessinés sous forme de boîtes en trois dimensions, qui sont des ressources de calcul : un «device» comme un téléphone ou un serveur, ou un «executionEnvironment» comme un système d'exploitation, un environnement d'exécution de conteneurs ou un cluster Kubernetes ; les nœuds peuvent être imbriqués. Les **artefacts**, dessinés sous forme de rectangles avec une icône de document et le stéréotype «artifact», sont les fichiers physiques déployés : un JAR, une image de conteneur, un paquet d'application mobile, un fichier de configuration. Les lignes entre les nœuds sont des **chemins de communication**, libellés avec des protocoles.

Les diagrammes de déploiement sont des vues d'allocation au sens du SEI (section 4.2). Ils répondent aux questions de l'équipe d'exploitation : où s'exécute chaque partie, en combien de copies, dans quelle région, quels chemins réseau existent. Le diagramme de déploiement de C4

couvre le même terrain avec une notation moins formelle ; l'un ou l'autre convient, pourvu qu'il y en ait un par environnement important et qu'il montre les réplicas, les services managés et les frontières réseau.

Diagramme de déploiement



Les nœuds sont l'endroit où les choses s'exécutent ; les artefacts sont ce qui est déployé ; les lignes sont des chemins de communication.

Diagrammes de séquence : qui appelle qui, et dans quel ordre

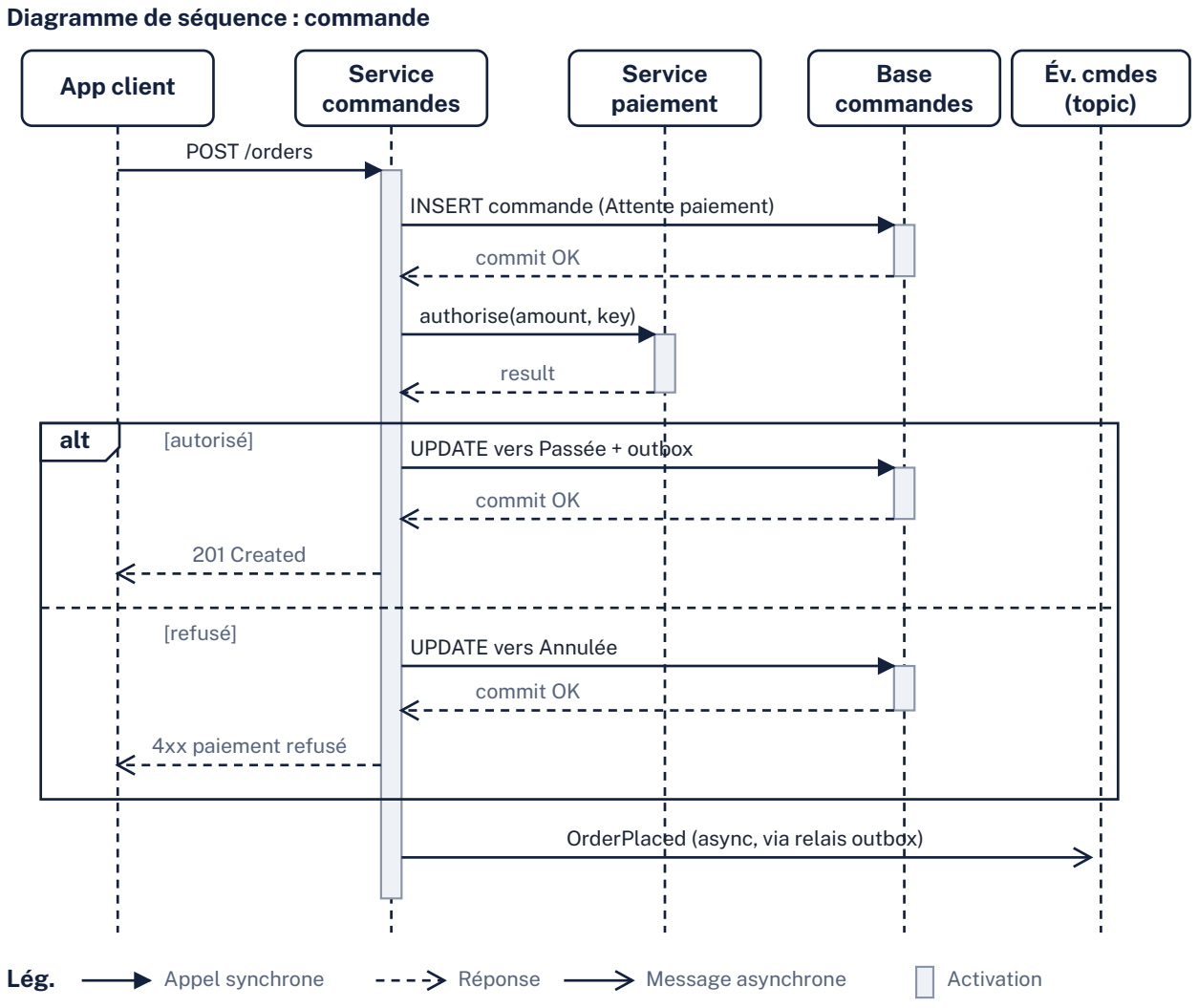
Si les architectes ne gardent qu'un seul diagramme UML, c'est généralement le diagramme de séquence. Il montre comment des participants interagissent dans le temps pour un scénario. Les participants apparaissent en haut, chacun avec une ligne de vie verticale en pointillés ; le temps s'écoule vers le bas. Un fin rectangle sur une ligne de vie, l'**activation** (spécification d'exécution), montre quand ce participant est occupé à traiter un message.

Les pointes de flèche ont un sens. Un **message synchrone**, pour lequel l'émetteur attend une réponse, a un trait plein et une pointe pleine. Une **réponse** a un trait en pointillés et une pointe ouverte. Un **message asynchrone**, pour lequel l'émetteur poursuit son travail sans attendre, a un trait plein et une pointe ouverte. Les représenter correctement compte sur le plan architectural : que le service des commandes attende le prestataire de paiement ou qu'il publie et passe à la suite détermine sa latence, ses modes de défaillance et son couplage (voir la section 0.6). L'ordre des étapes compte aussi : le diagramme ci-dessous enregistre la

commande avant d'appeler le prestataire ; une panne entre les deux ne peut donc jamais laisser un client débité sans trace de la commande, et la clé d'idempotence rend sûre une nouvelle tentative d'autorisation.

Les **fragments combinés** ajoutent le flux de contrôle : une boîte libellée `alt`, avec des sections gardées séparées par une ligne en pointillés, montre des alternatives (if/else), `opt` montre une partie facultative, `loop` une répétition et `par` des sections parallèles. Utilisez-les avec parcimonie ; un diagramme de séquence avec des boucles et des alternatives imbriquées, c'est du code mal dessiné. Dessinez un diagramme par scénario important, y compris les scénarios d'échec, car c'est là que se joue l'essentiel de la réflexion architecturale.

Diagramme de séquence



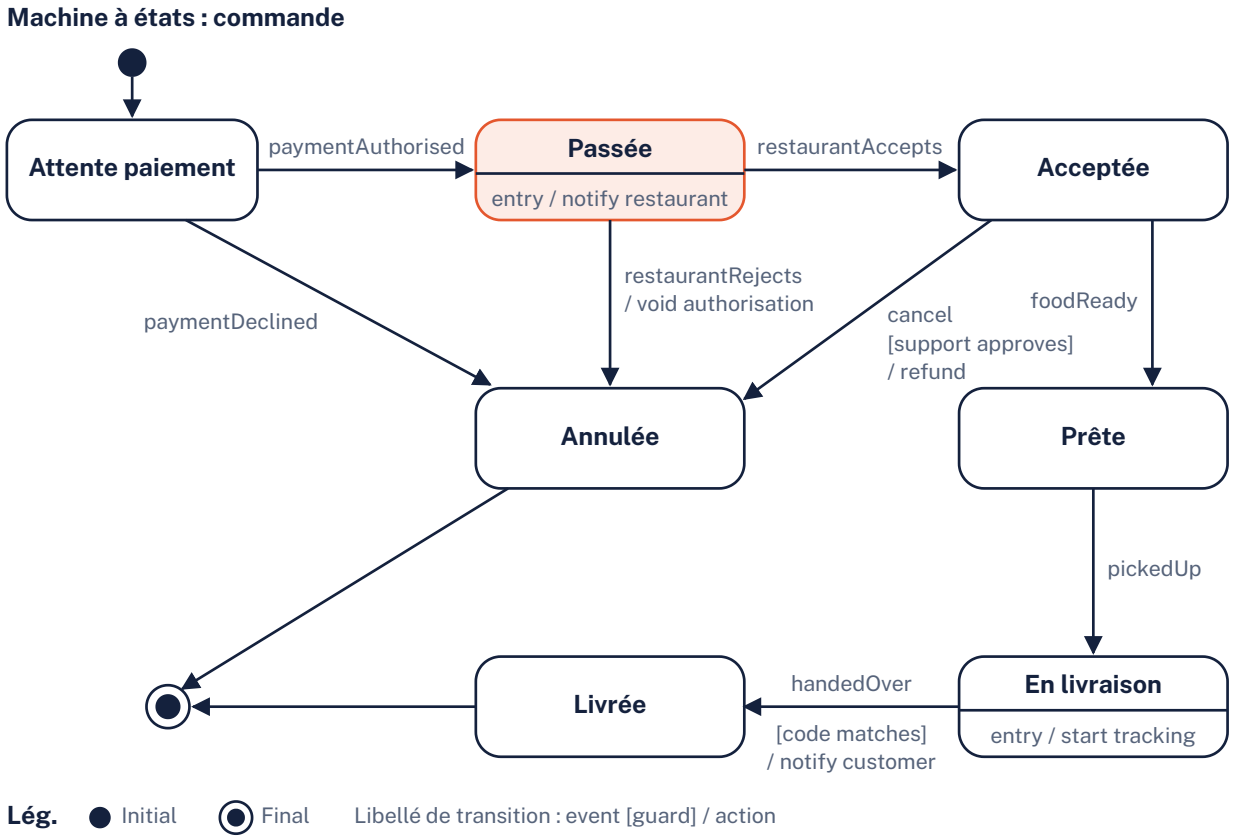
Passer une commande en toute sécurité : enregistrer d'abord la commande, autoriser la carte avec une clé d'idempotence, puis mettre à jour la commande. Un fragment alt montre les deux issues et un événement asynchrone termine le flux.

Diagrammes d’états-transitions : cycles de vie et règles

De nombreux objets métier ont un cycle de vie : une commande, un paiement, un abonnement, un sinistre d’assurance. Un diagramme d’états-transitions le décrit avec précision. Les **états** sont des rectangles aux coins arrondis ; un cercle plein marque le pseudo-état **initial** et un cercle pointé l’état **final**. Les **transitions** sont des flèches libellées `event [guard] / action` : l’événement (event) qui déclenche la transition, une condition de garde (guard) facultative qui doit être vraie, et une action facultative exécutée au passage. Les états peuvent avoir des comportements internes, comme `entry / notify restaurant`.

Les machines à états relèvent de l’architecture plus souvent qu’on ne le pense. Le cycle de vie de commande ci-dessous est partagé par plusieurs services : le service des commandes en est propriétaire, les services d’affectation et de notifications réagissent à ses transitions, et le personnel du support peut en déclencher certaines. S’accorder sur les états et les transitions autorisées, dans un seul diagramme, évite toute une catégorie de bugs dans lesquels un service pense qu’une commande est annulée alors qu’un autre lui affecte un livreur. Le diagramme fait aussi apparaître des questions que le texte cache : une commande peut-elle être annulée une fois en cours de livraison ? (Ici, non : il n’existe aucune transition de cet état vers Annulée.) Un remboursement est-il une action sur une transition ou un cycle de vie distinct ? L’argent suit lui aussi les états : la carte est seulement autorisée quand la commande est passée, et le montant est capturé quand le restaurant accepte ; un refus annule donc l’autorisation, et seule une annulation ultérieure nécessite un remboursement.

Diagramme d’états-transitions



Le cycle de vie de la commande : chaque transition autorisée, avec ses événements, ses gardes et ses actions, sur une seule page.

SI VOUS DÉBUTEZ

Pas à pas : lire un libellé de transition

L'idée la plus difficile de cette section est le petit langage écrit sur chaque flèche d'un diagramme d'états-transitions : event [guard] / action, soit événement [garde] / action. Suivez une commande à travers le cycle de vie présenté dans la section, et lisez chaque libellé tour à tour.

- Une nouvelle commande commence dans l'état Attente paiement, celui que désigne le cercle plein. Elle y reste jusqu'à ce qu'arrive un événement nommé par l'une de ses flèches sortantes.
- L'événement paymentAuthorised arrive, et sa flèche mène à Passée. Ce libellé ne comporte qu'un événement : ni condition ni action. À l'arrivée, le comportement d'entrée de Passée s'exécute, notify restaurant, si bien que le restaurant est informé de la commande.
- L'événement restaurantAccepts fait passer la commande à Acceptée. Une demande d'annulation arrive alors. La flèche qui part d'Acceptée indique cancel [support approves] / refund : cancel est l'événement, la garde entre crochets est la condition qui doit être vraie (le support approuve), et refund, après la barre oblique, est l'action exécutée au passage (rembourser).
- Si le support n'a pas approuvé, la garde est fausse : la transition ne se déclenche pas, et la commande reste simplement à l'état Acceptée.
- Si le support a approuvé, la transition se déclenche : le remboursement est effectué au passage, et la commande se retrouve à l'état Annulée.
- Si la commande avait déjà été En livraison, aucune flèche libellée cancel ne partirait de cet état : la demande n'aurait donc pas pu l'annuler. Une flèche absente est aussi une règle.
- **La leçon :** lisez chaque flèche comme « quand cet événement se produit, et si cette condition est vraie, fais ceci et passe là ». S'accorder sur ces libellés dans un seul diagramme aide plusieurs services à respecter les mêmes règles pour une commande.

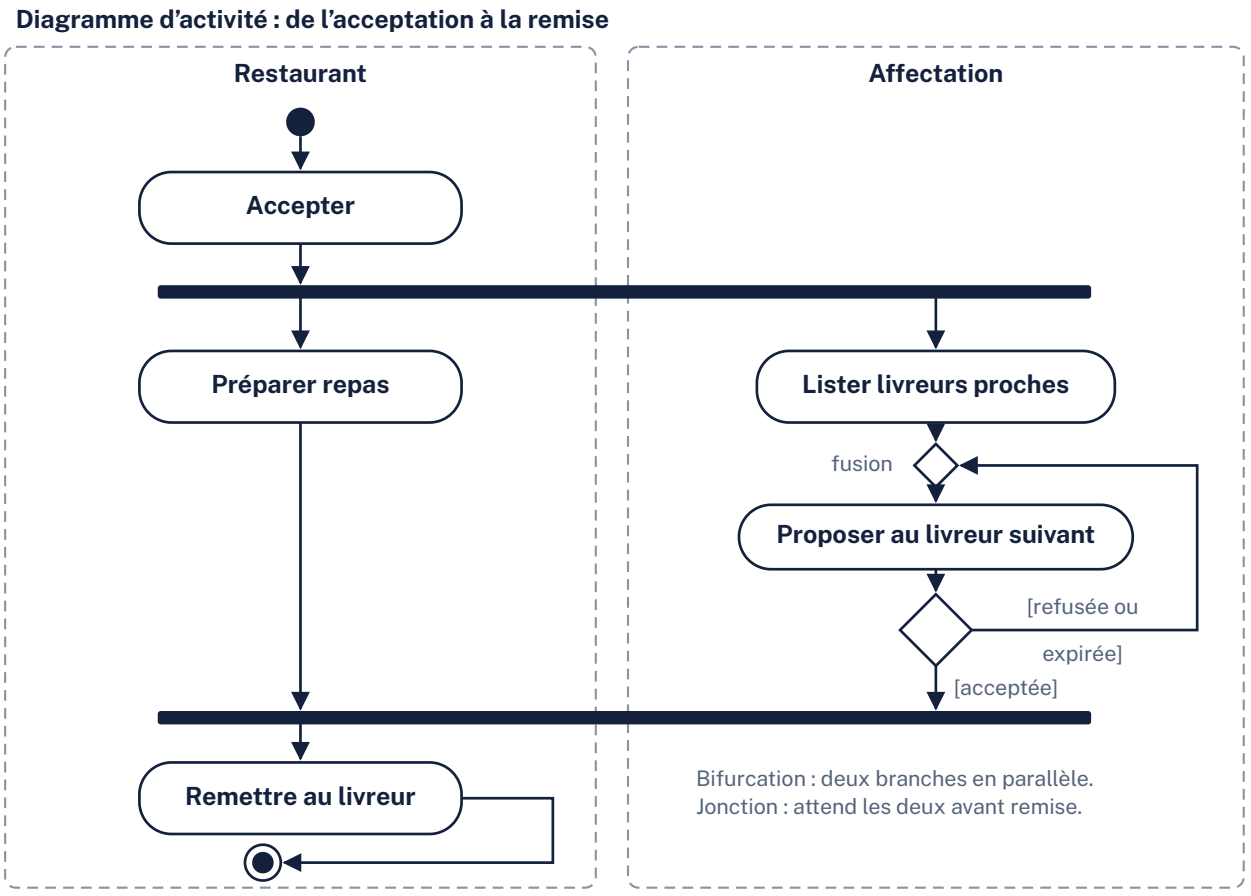
Diagrammes d'activité : flux de travail et parallélisme

Un diagramme d'activité est un organigramme à la sémantique précise. Les rectangles aux coins arrondis sont des **actions** ; les losanges sont des **décisions** (avec des gardes sur les flux sortants) et des fusions ; les barres épaisses sont des **bifurcations**, qui divisent un flux en flux parallèles, et des **jonctions**, qui attendent tous les flux entrants avant de continuer. Les

partitions, souvent appelées couloirs (swimlanes), montrent qui ou quoi exécute chaque action. Une boucle doit repasser par un losange de fusion : deux flux dessinés directement vers une même action signifieraient que l'action attend les deux, comme une jonction.

Les architectes utilisent les diagrammes d'activité pour les processus métier et les flux de travail qui couvrent plusieurs parties d'un système, en particulier lorsque le parallélisme et les responsabilités comptent. L'exemple montre que la préparation du repas et l'affectation d'un livreur se déroulent en parallèle, que l'affectation boucle jusqu'à ce qu'un livreur accepte, et que la remise attend les deux. Cela soulève immédiatement une question d'architecture : que se passe-t-il si aucun livreur n'accepte dans les vingt minutes alors que le repas est prêt ? Le diagramme rend la lacune visible ; la réponse pourrait être une transition de délai d'expiration (timeout), une étape de compensation ou une modification du cycle de vie de la commande. Les flux de travail de longue durée de ce type sont souvent implémentés sous forme de sagas ou de workflows orchestrés, traités au chapitre 6.

Diagramme d'activité



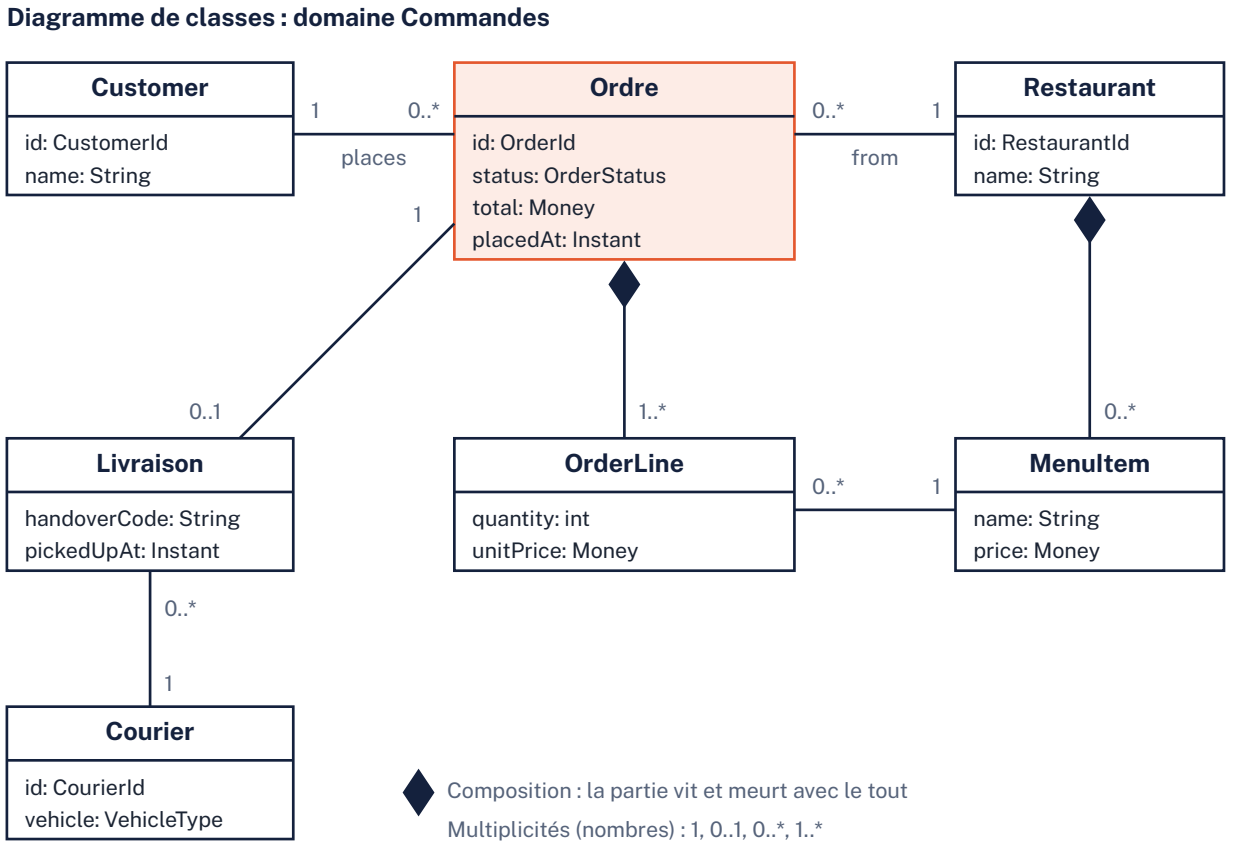
La préparation du repas et l'affectation d'un livreur se déroulent en parallèle entre une bifurcation et une jonction ; la décision boucle jusqu'à ce qu'un livreur accepte.

Diagrammes de classes, avec parcimonie : le modèle de domaine

Le diagramme de classes est le diagramme UML le plus connu, et celui que les architectes devraient utiliser avec le plus de parcimonie. Dessiner à la main chaque classe d'un service, c'est le mode plan détaillé sous sa forme la moins utile : le diagramme est périmé en quelques semaines, et le code dit la même chose plus précisément. Là où les diagrammes de classes méritent leur place, c'est pour le modèle de domaine : un petit diagramme des concepts métier clés, de leurs attributs et de leurs relations, dont les développeurs, les chefs de produit et les experts du domaine peuvent discuter ensemble. Dans les termes de la conception pilotée par le domaine (chapitre 5), il capture le langage omniprésent d'un contexte délimité.

L'essentiel tient en peu de choses. Une boîte de classe a un nom, des attributs et éventuellement des opérations. Une ligne simple est une **association**, éventuellement nommée, avec des multiplicités à chaque extrémité (`1` , `0..1` , `0..*` , `1..*`). Un losange plein marque une **composition** : les parties appartiennent à exactement un tout et ne lui survivent pas, comme les lignes de commande appartiennent à une commande. Un triangle creux marque une généralisation (héritage) et une ligne en pointillés à pointe ouverte une dépendance. Les multiplicités portent de vraies décisions de conception : `Order 1 to 0..1 Delivery` dit qu'une commande a au plus une livraison, ce qui exclut de répartir une grosse commande entre deux livreurs. Si le métier pouvait vouloir cela, c'est sur le diagramme que la question apparaît.

Diagramme de classes du modèle de domaine



Un modèle de domaine assez petit pour en discuter avec le métier : concepts clés, attributs, associations et multiplicités.

Les six diagrammes en un coup d'œil

Diagramme	Répond à	Notation clé	Usage typique par les architectes
Composant	Quelles sont les parties, et par quelles interfaces dépendent-elles les unes des autres ?	Boîtes de composants, interfaces fournies (boule) et requises (douille), dépendances	Montrer le sens des dépendances et les parties remplaçables ; ports et adaptateurs
Déploiement	Où s'exécute chaque partie, en combien de copies, connectée comment ?	Nœuds (boîtes 3D), artefacts, chemins de communication	Infrastructure de production et de staging, zones réseau
Séquence	Dans quel ordre les participants interagissent-ils pour ce scénario ?	Lignes de vie, activations, messages synchrones et asynchrones, réponses, fragments	Scénarios d'exécution clés, en particulier les défaillances, les nouvelles tentatives et les intégrations
États-transitions	Dans quels états cet objet peut-il se trouver, et qu'est-ce qui le fait passer de l'un à l'autre ?	États, états initial et final, transitions avec event [guard] / action	Cycles de vie partagés entre services : commandes, paiements, sinistres
Activité	Quel est le flux de travail, qui réalise chaque étape, qu'est-ce qui s'exécute en parallèle ?	Actions, décisions, bifurcations et jonctions, partitions	Processus métier, flux de travail de longue durée, sagas
Classes	Quels sont les concepts clés, et comment sont-ils liés ?	Classes, associations, multiplicités, composition, généralisation	Petits modèles de domaine ; diagrammes générés à partir du code

Quand UML est excessif, et quand la précision compte

UML est excessif quand le public ne le connaît pas et que la question n'en a pas besoin : un diagramme de contexte C4 communique mieux avec des dirigeants qu'un diagramme de cas d'utilisation UML, et un croquis au tableau blanc est l'outil qui convient pour une discussion de conception qui sera terminée en une heure. Il est excessif quand on maintient à la main des diagrammes qui pourraient être générés, et quand on impose une notation stricte dans un

croquis dont le but est d'explorer. L'observation de Fowler, selon laquelle la précision ajoutée dans UML 2 sert surtout les utilisateurs des modes plan détaillé et langage de programmation et alourdit la tâche de ceux qui font des croquis, reste valable.

La précision paie quand un malentendu coûte cher. Les **interfaces entre entreprises** en sont le cas le plus clair : quand l'entreprise de livraison de repas s'intègre à un prestataire de paiement ou au système de commande d'une grande chaîne de restaurants, chaque partie développe à partir de la description de l'autre et ne découvre les incohérences qu'en test ou en production. Un diagramme de séquence précis du flux de paiement, avec les délais d'expiration (timeouts), les nouvelles tentatives, les clés d'idempotence et les webhooks, joint à la spécification d'interface, est une assurance peu coûteuse. Les **secteurs réglementés et critiques pour la sûreté** (dispositifs médicaux, ferroviaire, aéronautique, automobile, certains systèmes financiers) exigent souvent une documentation de conception que les auditeurs peuvent suivre des exigences jusqu'à l'implémentation ; l'UML formel, ou son cousin d'ingénierie système SysML, y est courant. Les **règles de concurrence et de cycle de vie** partagées par plusieurs équipes, comme les états de commande ci-dessus, profitent d'une notation qui oblige à énoncer chaque transition. Enfin, le **développement externalisé ou distribué**, où les personnes qui implémentent une conception ne peuvent pas facilement interroger son auteur, a besoin de descriptions qui laissent moins de place à l'interprétation.

ATTENTION

Erreurs courantes avec UML

- Utiliser les mauvaises pointes de flèche dans les diagrammes de séquence, si bien qu'un événement asynchrone ressemble à un appel bloquant, ou l'inverse. Les lecteurs en tirent des conclusions architecturales.
- Dessiner à la main des diagrammes de classes du système entier. Générez-les si nécessaire, et ne dessinez que le modèle de domaine.
- Omettre la légende parce que « c'est de l'UML standard ». Beaucoup de lecteurs ne connaissent pas la notation ; une légende de deux lignes ne coûte rien.
- Un seul diagramme de séquence géant avec toutes les branches. Dessinez un diagramme par scénario, et faites des scénarios d'échec des diagrammes distincts.
- Exiger l'UML complet dans un croquis, ce qui ralentit la discussion et décourage complètement les gens de faire des diagrammes.

Questions fréquentes

Notre équipe doit-elle vraiment apprendre UML ?
Apprenez l'essentiel de la notation des six diagrammes de cette section ; cela prend quelques heures et donne à l'équipe un vocabulaire commun pour les interfaces, les interactions et les cycles de vie. Il n'est guère utile d'apprendre les autres types de diagrammes ou la spécification complète, sauf si vous travaillez dans un domaine qui les exige.

Quel est le lien entre UML et C4 ?
Ils sont complémentaires. C4 fournit un petit ensemble d'abstractions et une hiérarchie de diagrammes structurels ; UML fournit des notations précises, en particulier pour le comportement. Beaucoup d'équipes utilisent C4 pour les vues de contexte, de conteneurs et de composants, et UML pour les diagrammes de séquence, d'états-transitions et parfois de déploiement.

Puis-je dessiner de l'UML avec du texte plutôt qu'avec un outil de dessin ?
Oui. PlantUML et Mermaid génèrent tous deux des diagrammes de séquence, d'états-transitions, de classes et d'activité à partir de courts fichiers texte qui peuvent être placés en gestion de versions. La section 4.7 en montre des exemples.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Dans un diagramme de séquence, comment distinguer un appel synchrone d'un message asynchrone ?**

Un appel synchrone a une ligne continue avec une pointe de flèche pleine (l'émetteur attend) ; un message asynchrone a une ligne continue avec une pointe ouverte (l'émetteur continue). Les réponses sont des lignes en pointillés avec des pointes ouvertes.

▼ **Que signifie le libellé `handedOver [code matches] / notify customer` sur une transition ?**

Lorsque l'événement handedOver se produit et que la garde « code matches » (le code correspond) est vraie, la transition se déclenche et l'action « notify customer » (notifier le client) est exécutée.

▼ **Pourquoi les architectes devraient-ils rarement dessiner à la main les diagrammes de classes de services entiers ?**

Ils font double emploi avec ce que le code dit plus précisément et deviennent vite obsolètes ; les petits modèles de domaine méritent d'être dessinés, et les diagrammes complets peuvent être générés au besoin.

▼ **Donnez deux situations dans lesquelles un UML précis vaut l'effort.**

Par exemple, les interfaces entre entreprises, le développement réglementé ou critique pour la sûreté, les cycles de vie partagés par plusieurs équipes, ou le développement externalisé dans lequel les personnes qui implémentent ne peuvent pas facilement interroger les concepteurs.

▼ **À vous : Trois équipes de livraison de repas ne cessent d'être en désaccord sur le moment où une commande peut être annulée. Quel diagramme UML dessineriez-vous ? Donnez un exemple de transition dans la notation appropriée et dites pourquoi une plus grande précision vaut ici son prix.**

Un diagramme d'états, car la discussion porte sur le cycle de vie d'un objet. Exemple : de Confirmed à Cancelled sur cancelRequested [la cuisine n'a pas commencé] / refundPayment, selon la forme événement [garde] / action. Un diagramme de séquence montrerait un seul chemin, mais pas tous les états autorisés. La précision en vaut la peine, car plusieurs équipes partagent ce cycle de vie et les erreurs coûtent de l'argent réel. Une esquisse soignée d'une douzaine d'états met fin aux discussions sans exiger un modèle UML complet du système.

SOURCES

- UML specification versions (Object Management Group): https://www.omg.org/spec/UML/
- About the Unified Modeling Language 2.5.1 (OMG): https://www.omg.org/spec/UML/2.5.1/About-UML/
- UmlMode (Martin Fowler): https://martinfowler.com/bliki/UmlMode.html
- UmlAsSketch (Martin Fowler): https://martinfowler.com/bliki/UmlAsSketch.html

SECTION 4.5 SUR 4.8 · INTERMÉDIAIRE · 25 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Enregistrements de décision d'architecture (ADR)

AVANT DE LIRE

Cette section s'appuie sur :

- 0.8 Comment les logiciels sont construits
- 1.2 Architecture ou conception
- 1.6 Décisions et compromis

Termes déjà vus qu'elle utilise : Architecturalement significatif, Attribut de qualité, Compromis, Pull request

L'IDÉE EN BREF

Imaginez que vous emménagez dans une maison et que vous trouvez un mur qui semble ne servir à rien. Faut-il l'abattre ? Si le constructeur avait laissé un mot disant « ce mur soutient la salle de bains de l'étage », vous le sauriez. Un enregistrement de décision d'architecture est ce mot pour le logiciel : une page courte et datée qui explique un choix important, les alternatives envisagées et ce que ce choix coûte. Les mots ne sont jamais réécrits ; si les propriétaires changent d'avis, ils ajoutent un nouveau mot qui dit ce qui a changé et pourquoi.

CE QU'IL FAUT SAVOIR

- Un enregistrement de décision d'architecture (ADR) consigne une décision significative avec son contexte, les options envisagées, la décision et ses conséquences ; Michael Nygard a popularisé cette forme légère dans un billet de blog de 2011.
- Le format de Nygard comporte un titre, le contexte, la décision, le statut et les conséquences ; le contexte est rédigé de façon neutre, la décision à la voix active (« Nous allons… ») et les conséquences incluent les coûts autant que les bénéfices.
- MADR (version 4.0.0, 2024) ajoute les facteurs de décision, les options envisagées avec leurs avantages et leurs inconvénients, et la confirmation ; un Y-statement condense une décision en une seule phrase : dans le contexte de, face à, nous avons opté pour, et écarté, afin d'obtenir, en acceptant que.
- Les enregistrements passent de proposé à accepté ou rejeté, puis plus tard à obsolète ou remplacé ; le contenu d'un enregistrement accepté n'est jamais réécrit, car un nouvel enregistrement qui le remplace préserve l'historique.

- Les ADR vivent dans le dépôt de code sous forme de fichiers Markdown numérotés (par exemple docs/decisions/0012-title.md), sont revus dans des pull requests et sont répertoriés dans un journal des décisions.
- Rédigez un ADR pour les décisions coûteuses à inverser, qui touchent des qualités clés, contraignent d'autres équipes ou ont été contestées ; évitez les enregistrements rédigés après coup, sans véritables alternatives ou ne présentant que des conséquences positives.

Ce qu'est un ADR et le problème qu'il résout

Un enregistrement de décision d'architecture (ADR) est un court document texte qui consigne une décision architecturalement significative : la situation qui l'a imposée, les options mises en balance, ce qui a été choisi et ce qui découle de ce choix. Un ensemble de ces enregistrements, conservés dans l'ordre, forme le journal des décisions d'un système. L'idée de consigner les décisions est ancienne (la recherche sur la justification des choix de conception remonte à plusieurs décennies, et Jeff Tyree et Art Akerman ont publié un modèle détaillé de décision dans *IEEE Software* en 2005), mais la forme légère qu'utilisent aujourd'hui la plupart des équipes vient du billet de blog de Michael Nygard, *Documenting Architecture Decisions*, publié le 15 novembre 2011.

Le point de départ de Nygard était un problème que connaît toute équipe qui dure. Un nouveau développeur découvre dans le code une décision qui paraît étrange : une file supplémentaire, une table dupliquée, une bibliothèque que personne ne choisirait aujourd'hui. Sans savoir pourquoi elle a été prise, le nouveau venu n'a que deux mauvaises options. Il peut l'accepter aveuglément, en préservant quelque chose dont les raisons ne s'appliquent peut-être plus, ou la modifier aveuglément, en cassant quelque chose dont les raisons valent toujours. Le remède de Nygard était de consigner les raisons au moment où la décision est prise, dans un petit fichier conservé avec le code, pour que la motivation voyage avec le système. Les gros documents d'architecture y parviennent rarement, car personne ne les met à jour ; un court enregistrement d'une seule décision, écrit une fois et jamais réécrit, reste vrai, parce qu'il prétend seulement décrire ce qui était connu et décidé à une date donnée.

Cela s'appuie directement sur l'anatomie d'une décision présentée au chapitre 1 : décisions et compromis : contexte, options, critères, décision et ses raisons, conséquences et statut. Un ADR n'est rien d'autre que cette anatomie mise par écrit dans un format prévisible. Il répond aussi à la question que la section Architecture ou conception laisse ouverte en pratique : si l'architecture est l'ensemble des décisions significatives, le journal des ADR est ce qu'une équipe possède de plus proche d'une liste de son architecture.

Le format de Nygard

Nygard a proposé cinq parties. La plupart des équipes les utilisent encore, en plaçant souvent le statut en tête pour que les lecteurs voient d'emblée si l'enregistrement est en vigueur. Ses conseils sur la manière de rédiger chaque partie comptent autant que les intitulés.

Les cinq parties d'un ADR à la manière de Nygard

Composant	Ce qu'elle contient	Comment bien la rédiger
Titre	Un court groupe nominal qui nomme la décision, avec un numéro d'ordre, comme «ADR 12 : événements de commande via un courtier de messages».	Nommez la décision, pas le problème : un lecteur qui parcourt le journal doit apprendre ce qui a été décidé à partir du seul titre.
Contexte	Les forces en présence : le problème, les exigences de qualité et les contraintes pertinentes, les faits techniques et organisationnels, et la tension entre eux.	Nygard demande un langage neutre : décrivez les faits et les forces, pas la réponse préférée. Donnez des chiffres quand ils existent (charge de pointe, objectifs, dates d'incidents).
Décision	La réponse à ces forces.	Des phrases complètes à la voix active, commençant par «Nous allons…», pour qu'il n'y ait aucun doute sur ce qui a été décidé et par qui.
Statut	Proposé, accepté, puis éventuellement obsolète ou remplacé, avec une référence à l'enregistrement qui le remplace.	Tenez-le à jour ; c'est le seul champ qui change après l'acceptation.
Conséquences	Le contexte qui résulte de l'application de la décision.	Listez toutes les conséquences, pas seulement les positives : ce qui devient plus facile, ce qui devient plus difficile, les nouveaux risques, le nouveau travail et les options désormais fermées.

Nygard suggérait aussi qu'un enregistrement tienne sur une ou deux pages, que les enregistrements soient numérotés séquentiellement sans que les numéros soient jamais réutilisés, et qu'ils vivent dans le dépôt du projet en texte brut. Chacune de ces petites règles a sa raison d'être : les enregistrements courts sont lus, des numéros stables peuvent être cités dans les commentaires du code et les messages de commit, et le texte brut dans le dépôt peut être revu et versionné comme du code.

Autres modèles : MADR, Y-statements et formes plus riches

Le format de Nygard ne dit rien d'explicite sur les options rejetées, et beaucoup d'équipes ont constaté que les alternatives sont précisément ce qu'un lecteur ultérieur veut le plus connaître. Plusieurs modèles les ajoutent.

MADR (Markdown Architectural Decision Records, maintenu sur adr.github.io) est le modèle structuré le plus utilisé. Sa version actuelle au moment de la rédaction est la 4.0.0, publiée en septembre 2024. Le modèle complet comporte un titre, puis *Contexte et énoncé du problème*, des *Facteurs de décision* facultatifs, les *Options envisagées*, le *Résultat de la décision* avec des sous-sections facultatives *Conséquences* et *Confirmation*, des *Avantages et inconvénients des options* facultatifs et des *Informations complémentaires* facultatives. Les métadonnées comme le statut, la date, les décideurs, et les personnes consultées et informées peuvent figurer dans un en-tête YAML. Deux idées de MADR méritent d'être reprises même si vous utilisez un autre format. Les facteurs de décision explicitent les critères avant que les options soient jugées, ce qui est la discipline de la matrice pondérée du chapitre 1 sous une forme compacte. La *Confirmation* indique comment l'équipe vérifiera que la décision est réellement mise en œuvre et qu'elle tient toujours, par exemple au moyen d'une revue, d'un test ou d'une vérification automatisée de l'architecture.

Le Y-statement (déclaration en Y), proposé par Olaf Zimmermann et ses collègues et présenté pour la première fois à la conférence SATURN en 2012, condense une décision en une seule phrase structurée : *Dans le contexte de* (un cas d'utilisation ou un composant), *face à* (une préoccupation ou une exigence de qualité), *nous avons opté pour* (l'option) *et écarté* (les alternatives), *afin d'obtenir* (les bénéfices), *en acceptant que* (les inconvénients). Le nom est un jeu de mots sur «why» (« pourquoi » en anglais). Un Y-statement est idéal pour le résumé d'un journal des décisions, une diapositive, un commentaire de code ou la description d'une pull request ; pour une porte à sens unique contestée, il est généralement trop condensé, et les équipes le développent en un enregistrement complet.

Des modèles plus lourds existent pour les contextes réglementés ou de très grande taille. Le modèle de Tyree et Akerman de 2005 ajoute des champs comme les hypothèses, les contraintes, les décisions liées, les exigences liées et les implications. Ils sont utiles quand les décisions doivent être rattachées aux exigences pour un auditeur ; pour la plupart des équipes produit, ils ajoutent un effort que personne ne lit.

Choisir un modèle

Modèle	Taille	Consigne les options rejetées ?	Idéal pour
Nygard (2011)	D'une demi-page à deux pages	Seulement si vous les placez dans le contexte ou les conséquences	Les équipes qui débutent ; la plus petite habitude qui fonctionne.
MADR 4.0.0 (2024)	D'une à trois pages	Oui : les options envisagées, avec, en option, les avantages et inconvénients de chacune	La plupart des équipes produit ; les décisions contestées où les alternatives doivent être visibles.
Y-statement (2012)	Une phrase	Oui, dans la clause *et écarté*	Résumés de journal des décisions, petites décisions, descriptions de pull requests, diapositives.
Tyree et Akerman (2005)	Plusieurs pages	Oui, avec en plus les hypothèses, les contraintes et la traçabilité	Environnements réglementés qui exigent une traçabilité vers les exigences.

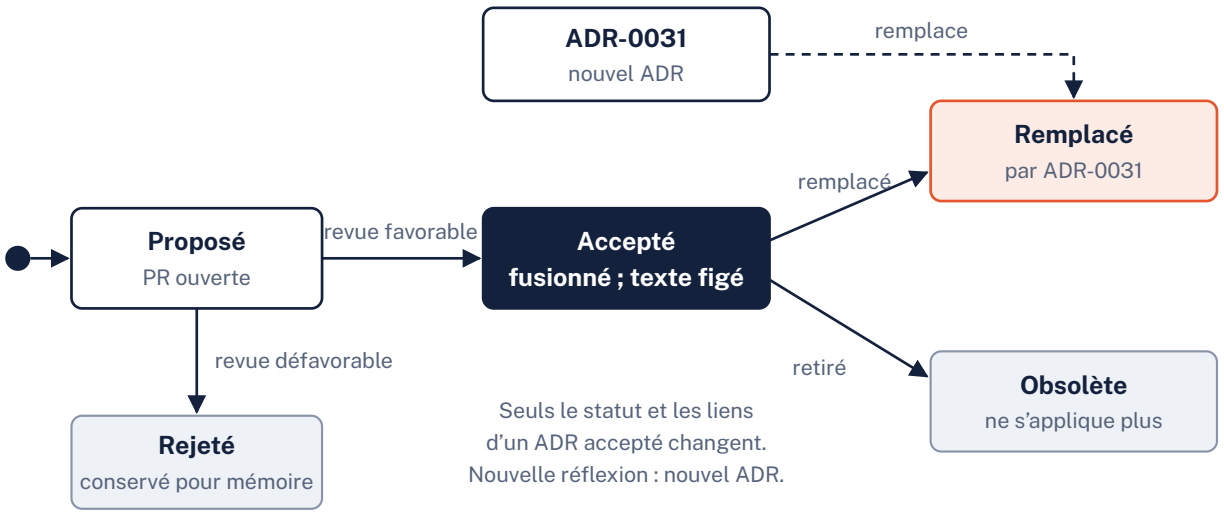
Statut, cycle de vie, et pourquoi on ne modifie pas les enregistrements acceptés

Un ADR passe par un petit nombre d'états. Il commence comme **proposé**, généralement sous la forme d'une pull request qui ajoute le fichier, pour que la discussion ait lieu dans les commentaires de revue. Quand les personnes responsables sont d'accord, il est fusionné comme **accepté**. Une proposition refusée peut être conservée comme **rejetée**, ce qui est précieux : la prochaine personne qui suggère la même idée trouve le raisonnement antérieur. Plus tard, un enregistrement accepté peut devenir **obsolète** (la décision ne s'applique plus, peut-être parce que le composant a été retiré) ou être **remplacé** par un enregistrement plus récent, avec des liens dans les deux sens. MADR liste le même ensemble : proposé, rejeté, accepté, obsolète et remplacé.

La règle centrale est que le contenu d'un enregistrement accepté n'est pas réécrit. Si l'équipe change d'avis, elle rédige un nouvel ADR qui remplace l'ancien, et ne modifie que la ligne de statut de l'ancien enregistrement pour renvoyer vers le nouveau. Les raisons sont pratiques. D'abord, l'enregistrement témoigne de ce qui était connu et décidé à un moment donné ; le modifier après coup transforme l'histoire en fiction et détruit la confiance qui rend le journal utile. Ensuite, l'ancien enregistrement explique pourquoi le système avait cette forme pendant une période, ce qui compte quand on débogue d'anciennes données ou qu'on lit du code ancien. Enfin, un nouvel enregistrement oblige l'équipe à dire ce qui a changé dans le contexte, ce qui est souvent l'enseignement le plus important. Les petites corrections (une coquille, un lien

cassé) sont sans danger ; changer la décision, les raisons ou les conséquences ne l'est pas. La description de cette pratique par Martin Fowler dit la même chose : une fois accepté, un enregistrement n'est pas rouvert ; on rédige plutôt un enregistrement qui le remplace.

Le cycle de vie d'un ADR



Les enregistrements sont proposés et revus, puis acceptés ou rejetés ; les enregistrements acceptés deviennent ensuite obsolètes ou sont remplacés, jamais réécrits.

SI VOUS DÉBUTEZ

Pas à pas : changer d'avis sans réécrire l'histoire

L'idée qui demande le plus d'attention ici est de comprendre pourquoi un enregistrement accepté n'est jamais réécrit, même quand l'équipe change d'avis. Le chapitre 1 a énoncé la règle ; suivez un changement réel dans le journal des décisions de la plateforme de livraison de repas pour voir ce qu'elle signifie en pratique.

- En 2025, l'équipe accepte l'ADR-0009 : le service de notification envoie les notifications push via un SDK tiers. Son contexte, sa décision et ses conséquences décrivent ce que l'équipe sait et choisit à ce moment-là.
- En 2026, après que l'ADR-0012 a introduit un courtier de messages, l'équipe décide de remplacer le SDK par sa propre diffusion push, qui fonctionne sur le courtier.
- Le raccourci tentant consiste à ouvrir l'ADR-0009 et à le réécrire pour décrire la nouvelle conception. Mais alors le journal affirme que l'équipe a choisi sa propre diffusion push dès 2025, et quiconque lit du code ancien ou d'anciennes données de notification ne trouve aucune trace de l'utilisation du SDK, ni de ses raisons.
- À la place, l'équipe rédige un nouvel enregistrement, l'ADR-0015, et le propose dans une pull request comme n'importe quel autre. Son contexte dit ce qui a changé depuis l'ADR-0009, ce qui est souvent la partie la plus utile de l'enregistrement.
- Quand l'ADR-0015 est accepté, la seule modification apportée à l'ADR-0009 est sa ligne de statut, qui indique désormais « Remplacé par l'ADR-0015 ». L'ADR-0015 renvoie vers lui, et le journal des décisions liste les deux.
- Un nouveau venu qui lit le journal voit maintenant toute l'histoire : ce qui a été décidé en 2025, ce qui l'a remplacé en 2026, et pourquoi.
- **La leçon :** un enregistrement accepté décrit ce qui était connu et décidé à sa date, et c'est pourquoi il reste vrai. Toute nouvelle réflexion va dans un nouvel enregistrement ; en dehors de petites corrections comme une coquille, l'ancien ne change que sa ligne de statut.

Où conserver les ADR

Conservez les ADR dans le dépôt du code qu'ils régissent, sous forme de fichiers en texte brut (généralement en Markdown), une décision par fichier, avec un numéro d'ordre et un court identifiant (slug) dans le nom du fichier. Deux conventions de répertoires sont courantes : `doc/adr`, la valeur par défaut de l'outil en ligne de commande `adr-tools` de Nat Pryce, et `docs/decisions`, recommandée par MADR avec des noms comme `0012-publish-order-events-through-broker.md`. Thoughtworks a placé les ADR légers dans l'anneau *Adopt* de son

Technology Radar (pour la dernière fois en 2018) et a expressément recommandé la gestion de versions plutôt qu'un wiki, pour que les enregistrements restent en phase avec le code qu'ils décrivent.

Conserver les enregistrements à côté du code présente des avantages concrets. Ils sont revus dans les mêmes pull requests que le code qui les met en œuvre, ils sont versionnés et branchés avec lui, un développeur peut les trouver avec les mêmes outils de recherche, et le code peut y renvoyer : un commentaire comme «voir ADR-0012» au-dessus d'un relais d'outbox indique au lecteur suivant où se trouvent les raisons. Les décisions qui couvrent de nombreux dépôts, comme le choix d'un fournisseur d'identité à l'échelle de l'organisation, ont besoin d'un emplacement à elles ; un dépôt d'architecture dédié avec les mêmes conventions fonctionne bien, avec des liens depuis chaque base de code concernée.

```
docs/
  decisions/
    README.md                                  ← the decision log (index)
    0001-record-architecture-decisions.md
    0004-use-hosted-card-fields.md
    0007-use-managed-postgresql-for-orders.md
    0012-publish-order-events-through-broker.md
    template.md
src/
  ordering/
    outbox_relay.py      # comment: see docs/decisions/0012-...
```

Une organisation type : des fichiers d'ADR numérotés à côté du code, un index, un modèle et un commentaire de code qui renvoie à l'enregistrement

Quelles décisions méritent un ADR

Rédiger un ADR pour chaque choix enfouit les plus importants ; n'en rédiger aucun fait perdre les raisons. Le critère est l'importance, telle que définie au chapitre 1 : une décision mérite un enregistrement quand elle est coûteuse à inverser, touche un attribut de qualité clé, a une large portée, contraint d'autres équipes ou a été contestée. En pratique, cela concerne des choix comme le stockage de données principal, le style de communication entre services, un fournisseur d'identité, une politique de versionnage d'API, une plateforme d'hébergement, une décision de développer ou d'acheter, et les conventions dont la violation coûte cher (la représentation des montants, le stockage des fuseaux horaires). Cela inclut aussi les décisions de *ne pas* faire quelque chose que l'on ne cesse de suggérer, comme ne pas introduire de cache pour l'instant.

Les choix locaux, peu coûteux à inverser et invisibles hors d'un seul module ne nécessitent généralement pas d'enregistrement : le choix d'une bibliothèque d'assertions de test, le nom d'une classe, l'organisation interne d'un composant. Un signal utile est la question que poserait

un nouveau venu : si quelqu'un risque de demander «mais pourquoi diable ont-ils fait ça ?» dans l'année, rédigez l'enregistrement maintenant, pendant que les raisons sont fraîches.

GUIDE DE DÉCISION

Guide de décision : cette décision a-t-elle besoin d'un ADR ?

Posez ces questions dans l'ordre quand un choix technique surgit lors d'une discussion de conception ou d'une revue de code.

Q1 **Revenir sur cette décision dans un an demanderait-il plus de quelques jours de travail, une migration de données ou des modifications par d'autres équipes ?**

- **Oui** Rédigez un ADR. C'est une porte à sens unique, ou presque.
- **Non** Passez à la question suivante.

Q2 **Touche-t-elle sensiblement un attribut de qualité qui compte pour les parties prenantes (par exemple la disponibilité, la latence, la sécurité ou le coût) ou une contrainte comme la réglementation ?**

- **Oui** Rédigez un ADR, et reliez-le au scénario de qualité qu'il sert.
- **Non** Passez à la question suivante.

Q3 **A-t-elle été contestée, ou l'équipe a-t-elle rejeté une option que l'on risque de proposer à nouveau ?**

- **Oui** Rédigez un ADR court (un Y-statement peut suffire) pour que le raisonnement ne soit pas rediscuté depuis le début.
- **Non** Passez à la question suivante.

Q4 **Un nouveau venu regardera-t-il probablement le résultat en demandant pourquoi on a procédé ainsi ?**

- **Oui** Rédigez un ADR court, ou au moins un commentaire de code et un message de commit qui expliquent la raison.
- **Non** Pas besoin d'ADR. Un message de commit clair et le code lui-même suffisent.

Un ADR complet commenté

La plateforme de livraison de repas utilisée tout au long de ce chapitre comprend des applications pour les clients, les restaurants et les livreurs, un service des commandes, des paiements via un prestataire de paiement externe, l'affectation des livreurs et des notifications. L'enregistrement ci-dessous suit la structure de MADR, légèrement raccourcie. Remarquez comment le contexte donne des chiffres et des dates plutôt que des adjectifs, comment les facteurs sont énoncés avant les options, comment les options rejetées sont examinées

équitablement, et comment les conséquences incluent des coûts réels. La décision elle-même (une messagerie asynchrone pour les événements de commande) est étudiée en détail au chapitre 6 : architecture pilotée par les événements ; ici, l'important est la manière dont elle est consignée.

```
# ADR-0012: Publish order events through a managed message broker

- Status: Accepted
- Date: 2026-09-14
- Deciders: ordering team lead, platform lead, principal architect
- Consulted: restaurant, courier and notifications teams; SRE
- Related: ADR-0007 (managed PostgreSQL for the order service)

## Context and problem statement

When a customer's payment succeeds, the ordering service calls the restaurant,
courier-dispatch and notification services synchronously over HTTP before it
confirms the order. On Friday evenings (peak about 40 orders/s) p99 order
confirmation time reaches 4.2 s against our target of 1.5 s (scenario QS-3).
On Friday 21 August a notification-service outage blocked all new orders for
23 minutes. Two new consumers (loyalty points, analytics) want order data.

## Decision drivers

- Order confirmation must not depend on notification or dispatch availability.
- p99 confirmation under 1.5 s at three times today's peak.
- No order event may be lost, even if a service crashes mid-request.
- New consumers should not require changes to the ordering service.
- The platform team has three people; operational load must stay small.

## Considered options

1. Keep synchronous calls; add timeouts, retries and circuit breakers.
2. Publish events to a managed message broker using a transactional outbox.
3. Let consumers poll a shared "order_events" table in the order database.

## Decision outcome

Chosen option: 2. We will publish OrderPlaced, OrderAccepted, OrderPickedUp,
OrderDelivered and OrderCancelled events to our cloud provider's managed
broker. The ordering service writes each event to an outbox table in the
same database transaction as the order; a relay publishes it. Payment
authorisation stays synchronous, because the customer must see the result.

### Consequences

- Good: confirmation no longer waits for downstream services.
- Good: loyalty and analytics can subscribe without ordering-team work.
- Bad: services become eventually consistent; the restaurant tablet may
  show a new order a second or two after the customer's confirmation.
- Bad: a broker to pay for and monitor; consumers must be idempotent
  because delivery is at-least-once.
- Neutral: event schemas become public contracts and need versioning.

### Confirmation
```

```
Load test at 120 orders/s in staging before go-live; a build check fails
if the ordering service gains an HTTP client for notification or dispatch.

## Pros and cons of the other options

- Option 1: least new infrastructure, but a slow dependency still delays
  every order, and each new consumer means another call to add.
- Option 3: no new infrastructure, but couples every consumer to our schema
  and adds polling load to the order database (see ADR-0007).
```

Source Markdown de l'ADR-0012 dans docs/decisions/0012-publish-order-events-through-broker.md

EXEMPLE COMMENTÉ

Exemple commenté : la même décision sous forme de Y-statement

Pour le journal des décisions, la description d'une pull request ou une diapositive, l'équipe rédige aussi la décision sous la forme d'un seul Y-statement.

Y-statement (déclaration en Y)	Dans le contexte de **la confirmation des commandes sur la plateforme de livraison de repas**, face à **des services en aval lents ou indisponibles qui bloquent les commandes aux heures de pointe**, nous avons opté pour **la publication des événements de commande via un courtier de messages managé avec une outbox transactionnelle** et écarté **les appels synchrones avec disjoncteurs, ainsi que l'interrogation d'une table partagée par les consommateurs**, afin d'obtenir **une confirmation des commandes qui ne dépend pas des services de notification, d'affectation ou des restaurants, et de nouveaux consommateurs qui n'exigent aucune modification du service des commandes**, en acceptant que **les services deviennent cohérents à terme, que le courtier doive être exploité et payé, et que chaque consommateur doive gérer les événements en double**.
Ce qu'il conserve	Le contexte, la préoccupation déterminante, le choix, les options rejetées, les bénéfices et les coûts acceptés : chaque élément de l'enregistrement complet, en un coup d'œil.
Ce qu'il perd	Les chiffres (p99 de 4,2 s, panne de 23 minutes), les décideurs, le plan de confirmation et le raisonnement contre chaque option rejetée. Pour une décision aussi significative, l'enregistrement complet reste nécessaire.

Un bon test pour tout ADR consiste à vérifier qu'on peut le condenser en un Y-statement sensé. Si vous ne pouvez pas remplir les clauses *et écarté* ou *en acceptant que*, l'enregistrement n'a probablement envisagé aucune véritable alternative ou a caché ses coûts.

Revoir les ADR

Comme un ADR est un fichier du dépôt, le mécanisme de revue naturel est la pull request. La personne qui propose l'ouvre avec le statut *proposé* ; les personnes concernées commentent ; l'enregistrement est révisé ; quand les personnes responsables sont d'accord, le statut passe à *accepté* et la pull request est fusionnée. La discussion reste ainsi attachée à l'enregistrement et visible pour quiconque lira plus tard l'historique.

Qui doit faire la revue dépend de la manière dont les décisions sont gouvernées. Dans un modèle centralisé, un comité d'architecture approuve les enregistrements significatifs. Dans le processus de consultation (advice process) décrit par Andrew Harmel-Law (martinfowler.com, 2021), n'importe qui peut prendre une décision architecturale, à condition de demander d'abord l'avis de toutes les personnes réellement concernées et de celles qui ont l'expertise pertinente, et de consigner ces avis ; la personne qui décide n'est pas tenue de suivre les avis, mais doit les entendre. Les ADR sont la colonne vertébrale de ce processus : les avis figurent dans l'enregistrement, et un forum consultatif ouvert et régulier examine les enregistrements proposés. Les modèles de gouvernance sont traités au chapitre 11 : la gouvernance dans les organisations agiles.

La liste de contrôle *ecADR* d'Olaf Zimmermann (2020) donne aux personnes qui font la revue une définition compacte de « terminé » pour une décision. **E**vidence (éléments probants) : il y a des raisons de croire que l'option fonctionnera, d'après un spike, une preuve de concept ou une expérience fiable. **C**riteria (critères) : au moins deux options ont été comparées au regard des préoccupations des parties prenantes. **A**greement (accord) : la décision a été mise à l'épreuve par des pairs et l'équipe l'accepte. **D**ocumentation : elle est consignée et partagée avec les personnes concernées. **R**ealisation and review (réalisation et réexamen) : la mise en œuvre est planifiée, et une date ou un déclencheur pour la réexaminer est fixé. Une personne qui vérifie ces cinq points, ainsi que l'honnêteté des conséquences, repère la plupart des enregistrements faibles.

Anti-patrons

Les ADR échouent de manière reconnaissable. Zimmermann en a catalogué plusieurs en 2023 sous des noms faciles à retenir, et les plus nuisibles méritent d'être connus.

ATTENTION

Comment les ADR tournent mal

- **Rédigé après coup pour justifier.** La décision a été prise dans un couloir et l'enregistrement est reconstitué à rebours pour paraître rationnel. Ses options sont inventées et son contexte est sélectif. Rédigez l'enregistrement pendant la décision, comme une proposition, et non comme un communiqué de presse après coup.
- **Aucune option envisagée** (le *Sprint* de Zimmermann), ou une *Dummy Alternative* (alternative bidon) : une option de paille qui ne pourrait jamais fonctionner, ajoutée pour que l'option favorite paraisse examinée. S'il n'existe aucune alternative crédible, dites-le et expliquez pourquoi.
- **Aucune conséquence réelle** (le *Free Lunch Coupon*, ou bon pour un repas gratuit) ou seulement des bénéfices (le *Fairy Tale*, ou conte de fées). Toute décision significative a des coûts ; un enregistrement qui les cache sera démasqué, et les lecteurs cesseront de faire confiance au journal.
- **Le Mega-ADR.** Des diagrammes, des spécifications de composants et du code entassés dans un enregistrement de plusieurs pages. Mettez plutôt un lien vers la documentation d'architecture ; l'ADR consigne la décision et ses raisons.
- **Modifier des enregistrements acceptés** pour les faire correspondre à la conception actuelle, ce qui détruit l'historique que le journal a justement pour rôle de conserver.
- **Des ADR pour tout,** y compris pour des portes à double sens triviales, jusqu'à ce que personne ne les lise ; ou des ADR que personne ne trouve, éparpillés entre des wikis, des fils de discussion et des présentations.

Le journal des décisions comme index

Chaque enregistrement est utile ; le journal dans son ensemble l'est davantage. Un court fichier d'index (souvent le `README.md` du dossier des décisions) liste chaque enregistrement avec son numéro, son titre, son statut et sa date. Un nouveau venu le lit en cinq minutes et obtient un historique condensé de l'architecture : ce qui a été décidé, ce qui a été rejeté, ce qui a été remplacé et quand. Des outils comme `adr-tools` peuvent générer l'index et créer des enregistrements de remplacement qui mettent automatiquement à jour le statut de l'ancien, et des outils de documentation, dont le DSL de Structurizr, peuvent importer un dossier d'ADR dans la documentation publiée.

Traitez le journal comme une partie de la description d'architecture. Le modèle arc42 comporte une section consacrée aux décisions d'architecture qui peut simplement y renvoyer, et un pipeline docs-as-code peut le publier avec le reste de la documentation.

```
# Decision log

| ADR  | Title                                                | Status                | Date       |
|------|------------------------------------------------------|-----------------------|------------|
| 0001 | Record architecture decisions                        | Accepted              | 2025-03-02 |
| 0004 | Use the payment provider's hosted card fields        | Accepted              | 2025-04-18 |
| 0007 | Use managed PostgreSQL for the order service         | Accepted              | 2025-06-11 |
| 0009 | Send push notifications through a third-party SDK    | Superseded by 0015    | 2025-09-30 |
| 0012 | Publish order events through a managed broker        | Accepted              | 2026-09-14 |
| 0013 | Store courier locations in a time-series database    | Rejected              | 2026-09-20 |
| 0015 | Replace the SDK with our own push delivery on broker | Accepted              | 2026-10-08 |
```

docs/decisions/README.md : le journal des décisions de la plateforme de livraison de repas

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Pourquoi la section Contexte d'un ADR doit-elle éviter de plaider pour l'option choisie ?**

Parce que son rôle est de consigner les forces et les faits tels qu'ils étaient, pour qu'un lecteur ultérieur puisse juger s'ils tiennent toujours. Si le contexte est rédigé comme un plaidoyer, le lecteur ne peut pas séparer la situation de la conclusion.

▼ **Votre équipe change d'avis sur une décision acceptée. Que faites-vous de l'ancien ADR ?**

Vous laissez son contenu inchangé, vous passez son statut à remplacé avec un lien vers le nouvel enregistrement, et vous rédigez un nouvel ADR qui explique ce qui a changé dans le contexte et ce qui est désormais décidé.

▼ **Qu'apporte la section Confirmation de MADR qui manque au format de Nygard ?**

L'énoncé de la manière dont l'équipe vérifiera que la décision est mise en œuvre et qu'elle tient toujours, par exemple au moyen d'un test de charge, d'une revue ou d'une vérification automatisée des dépendances.

▼ **Quel est le test rapide pour savoir si un ADR a réellement envisagé des alternatives et des coûts ?**

Essayez de le reformuler en Y-statement. Si les clauses « et écarté » ou « en acceptant que » ne peuvent pas être remplies honnêtement, l'enregistrement manque de véritables options ou de véritables conséquences.

▼ **À vous : Critiquez cet ADR : « Nous avons choisi Kafka parce que c'est le meilleur courtier. Conséquences : rapide et scalable. » Six mois plus tard, quelqu'un le modifie pour indiquer RabbitMQ. Qu'est-ce qui ne va pas, et qu'aurait-il fallu faire à chaque fois ?**

Il ne donne ni contexte, ni variantes, ni critères, et « meilleur » n'est pas une raison. Les conséquences ne citent que des bénéfices, sans coûts tels que l'exploitation du cluster. Il devrait énoncer le problème avec des chiffres, comparer équitablement au moins deux options, puis dire « Nous allons… » et nommer ce à quoi on renonce. Le modifier plus tard détruit l'historique : rédigez un nouvel ADR qui le remplace, et marquez l'ancien comme remplacé.

SOURCES

- Documenting Architecture Decisions, Michael Nygard (2011): https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions
- MADR: Markdown Architectural Decision Records: https://adr.github.io/madr/
- Architecture Decision Record, Martin Fowler (bliki): https://martinfowler.com/bliki/ArchitectureDecisionRecord.html
- Scaling the Practice of Architecture, Conversationally, Andrew Harmel-Law: https://martinfowler.com/articles/scaling-architecture-conversationally.html
- A Definition of Done for Architectural Decision Making, Olaf Zimmermann: https://ozimmer.ch/practices/2020/05/22/ADDefinitionOfDone.html
- Jeff Tyree and Art Akerman: Architecture Decisions: Demystifying Architecture (IEEE Software, 2005): https://doi.org/10.1109/MS.2005.27

SECTION 4.6 SUR 4.8 · INTERMÉDIAIRE · 22 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

arc42 et autres modèles de documentation

AVANT DE LIRE

Cette section s'appuie sur :

- 4.2 Vues et points de vue
- 4.3 Le modèle C4
- 4.5 Enregistrements de décision

Termes déjà vus qu'elle utilise : Description d'architecture, Point de vue, Correspondance, Scénario d'attribut de qualité, Enregistrement de décision d'architecture (ADR)

L'IDÉE EN BREF

Un modèle est comme la structure standard du manuel d'une voiture : chaque manuel a une section sur les pneus, une sur les voyants d'alerte, une sur l'entretien, si bien que vous savez où chercher dans n'importe quelle voiture. arc42 fait de même pour l'architecture logicielle. Il propose douze sections standard, comme les objectifs, le contexte, la structure et les décisions, pour que les rédacteurs sachent quoi couvrir et que les lecteurs sachent où trouver l'information. Vous n'êtes pas obligé de remplir toutes les sections ; un petit système peut n'en nécessiter que quatre ou cinq, courtes.

CE QU'IL FAUT SAVOIR

- arc42, créé par Gernot Starke et Peter Hruschka et gratuit sous licence Creative Commons, est un modèle en douze sections ; il prescrit où placer l'information, pas comment concevoir ni quelle notation utiliser.
- Les sections sont : introduction et objectifs, contraintes, contexte et périmètre, stratégie de solution, vue des blocs de construction, vue d'exécution, vue de déploiement, concepts transversaux, décisions d'architecture, exigences de qualité, risques et dette technique, et glossaire.
- Chaque section est facultative ; une description minimale couvre trois à cinq scénarios de qualité, le contexte, une brève stratégie de solution, le niveau 1 de la vue des blocs de construction et les concepts transversaux clés.
- Utilisez arc42 avec légèreté : renvoyez aux exigences, aux ADR et au code d'infrastructure plutôt que de les copier, et indiquez délibérément quand une section n'est pas nécessaire.

> - Le modèle Views and Beyond du SEI est plus complet, avec une feuille de route, des dossiers par vue et une mise en correspondance des vues ; l'IEEE 1016-2009 spécifie les descriptions de conception logicielle et a le statut « inactive-reserved » depuis 2020.
> - L'ISO/IEC/IEEE 42010 prescrit le contenu d'une description d'architecture, pas un modèle ; une description arc42 est conforme dès lors que les parties prenantes, les préoccupations, les points de vue, les vues, les correspondances et les justifications sont rendus explicites.

Ce qu'est arc42

arc42 est un modèle gratuit pour décrire et communiquer l'architecture logicielle, créé par Gernot Starke et Peter Hruschka et utilisé en pratique depuis 2005. Il est publié sous licence Creative Commons Attribution-ShareAlike, dans plus d'une douzaine de langues et dans des formats comme AsciiDoc, Markdown et Word, avec une version «plain» (vierge) et une version qui intègre des textes d'aide. Il est particulièrement répandu dans les pays germanophones et dans les formations iSAQB, mais il est utilisé dans le monde entier.

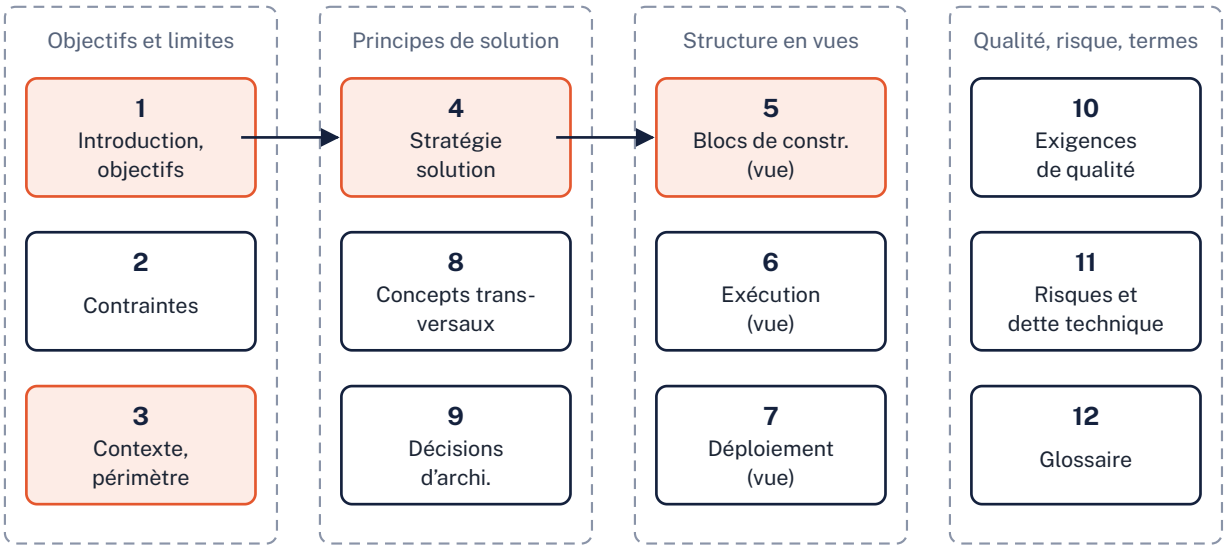
arc42 n'est délibérément ni une méthode ni une notation. Il ne vous dit pas comment concevoir, et il n'impose ni UML, ni C4, ni aucun autre style de diagramme. C'est un ensemble de douze tiroirs étiquetés : pour chaque type d'information architecturale, il indique où la ranger, pour que chaque lecteur sache où chercher et que chaque rédacteur sache ce qui manque. Ses auteurs affirment clairement que chaque section est facultative ; un petit système peut remplir quatre sections en quelques pages, un grand peut remplir les douze sur de nombreuses pages.

Le modèle s'appuie sur des idées vues plus tôt dans ce chapitre. Ses trois vues (blocs de construction, exécution et déploiement) correspondent étroitement aux catégories modules, composants et connecteurs, et allocation du SEI, ainsi qu'aux vues de développement, de processus et physique de Kruchten, présentées dans vues et points de vue. Sa section consacrée aux décisions est l'endroit où vont les enregistrements de décision d'architecture, et sa section consacrée à la qualité est l'endroit où vont les scénarios d'attributs de qualité du chapitre 2.

Les douze sections

Le diagramme regroupe les sections selon la question à laquelle elles répondent. Les sections 1 à 3 fixent les objectifs et les limites : à quoi sert le système, ce qui limite la solution et où le système s'arrête. La section 4 énonce les idées fondamentales de la solution, et les sections 8 et 9 contiennent les principes et les décisions qui s'appliquent à l'ensemble de celle-ci. Les sections 5 à 7 montrent la structure sous trois angles. Les sections 10 à 12 rendent explicites la qualité, les risques et le vocabulaire.

La structure d’arc42



En grisé : le cœur même d’une description minimale (objectifs, contexte, stratégie et blocs de construction de niveau 1) ; la FAQ d’arc42 y ajoute 3 à 5 scénarios de qualité et les concepts clés (8).

Les douze sections d’arc42 regroupées par finalité ; les sections mises en évidence forment le cœur de toute description, même minimale.

Ce que contient chaque section, et dans quelle mesure

Section	Ce qu'elle contient	Petit système (une équipe)	Grand système (nombreuses équipes)
1. Introduction et objectifs	Un bref aperçu des exigences, les trois à cinq principaux objectifs de qualité par ordre de priorité, et un tableau des parties prenantes avec les attentes de chaque groupe.	Une demi-page : finalité, trois objectifs de qualité, cinq ou six parties prenantes.	Deux ou trois pages, qui renvoient aux documents d'exigences plutôt que de les copier.
2. Contraintes	Tout ce qui limite la liberté de conception : réglementations, technologies imposées, règles organisationnelles, échéances, budget, conventions.	Une liste à puces.	Un tableau regroupant les contraintes techniques, organisationnelles et juridiques, chacune avec sa source.
3. Contexte et périmètre	Le système vu comme une boîte noire : les utilisateurs et les systèmes voisins, ce qui circule entre eux (contexte métier), et les canaux et protocoles utilisés (contexte technique).	Un diagramme de contexte (souvent un diagramme de contexte système C4) et un court tableau des interfaces externes.	Des diagrammes de contexte métier et technique, plus un tableau des interfaces avec leurs responsables et leurs contrats.
4. Stratégie de solution	Les quelques décisions et approches fondamentales qui façonnent tout le reste : style de décomposition, technologies clés, manière d'atteindre les principaux objectifs de qualité.	Une poignée de puces, chacune rattachée à un objectif de qualité.	Une ou deux pages ; un tableau qui associe chaque objectif de qualité à l'approche qui le sert.
5. Vue des blocs de construction	La décomposition statique en blocs de construction (sous-systèmes, services, composants, modules), affinée niveau par niveau, chacun avec ses responsabilités et ses interfaces. C'est généralement la section la plus volumineuse.	Le niveau 1 seulement : un diagramme (un diagramme de conteneurs C4 convient) et une phrase par bloc.	Le niveau 1 pour l'ensemble du système, les niveaux 2 et 3 seulement pour les parties complexes ou risquées.
6. Vue d'exécution	La manière dont les blocs de construction coopèrent dans les scénarios importants : cas d'utilisation clés, démarrage, gestion des erreurs, failover (basculement).	Un ou deux diagrammes de séquence pour les flux les plus importants.	Les scénarios critiques et les scénarios surprenants ; jamais tous les cas d'utilisation.

Section	Ce qu'elle contient	Petit système (une équipe)	Grand système (nombreuses équipes)
7. Vue de déploiement	L'infrastructure technique : environnements, nœuds, réseaux, et quels blocs de construction s'exécutent où.	Un diagramme ou une courte liste des services hébergés.	Des diagrammes par environnement et par région, reliés à l'infrastructure en tant que code qui les définit.
8. Concepts transversaux	Concepts qui s'appliquent à de nombreux blocs de construction : modèle du domaine, sécurité, gestion des erreurs, journalisation et observabilité, persistance, internationalisation, tests.	Un court paragraphe pour chacun des deux ou trois concepts qui comptent.	Une page ou plus par concept, souvent sous la responsabilité d'une équipe plateforme ou d'une équipe facilitatrice.
9. Décisions d'architecture	Les décisions importantes avec leurs raisons, sauf si elles sont consignées ailleurs.	Un lien vers le dossier des ADR.	Un lien vers le journal des décisions, avec éventuellement les cinq ou six décisions que tout nouveau venu doit connaître.
10. Exigences de qualité	Un arbre de qualité (l'arbre d'utilité du chapitre 2) et des scénarios de qualité concrets qui affinent les objectifs de qualité de la section 1.	Trois à cinq scénarios.	Un arbre d'utilité avec des scénarios priorisés, reliés à des fonctions d'aptitude.
11. Risques et dette technique	Les risques techniques et la dette connus, idéalement priorisés, avec des plans d'atténuation.	Une courte liste, revue chaque trimestre.	Un lien vers le registre de la dette, plus les principaux risques avec leurs responsables.
12. Glossaire	Les termes importants du domaine et les termes techniques, avec des définitions sur lesquelles tout le monde s'accorde.	Une dizaine de termes.	Un glossaire partagé, parfois par contexte délimité.

Utiliser arc42 avec légèreté

La façon la plus courante d'échouer avec arc42 est de le traiter comme un formulaire à remplir : douze intitulés sont créés, chacun est rempli avec quelque chose, et le résultat est long, générique et jamais lu. Les conseils du modèle lui-même vont dans l'autre sens. La FAQ d'arc42 indique qu'il n'existe pas de règle générale pour le minimum, car il dépend de la criticité, de la taille, de la complexité et du risque, mais suggère que même une description

minimale contienne trois à cinq exigences de qualité centrales sous forme de scénarios, le contexte et les interfaces externes, une brève stratégie de solution, la vue des blocs de construction de niveau 1 et les concepts transversaux les plus importants.

Quatre habitudes permettent de garder arc42 léger. **Supprimez les sections vides ou signalez-les délibérément** («Déploiement : plateforme managée unique ; voir le dépôt d'infrastructure»), pour que le lecteur sache que le vide est intentionnel. **Faites des liens plutôt que des copies** : les exigences vivent dans le backlog, les décisions dans le dossier des ADR, le déploiement dans l'infrastructure en tant que code ; arc42 renvoie vers eux. **Préférez des diagrammes accompagnés de courts tableaux d'éléments** à des pages de prose. **Écrivez pour les questions du prochain lecteur**, pas pour l'exhaustivité : si personne ne posera de question sur le comportement à l'exécution des écrans d'administration, ne le décrivez pas.

arc42 se combine bien avec le modèle C4. Une association courante consiste à placer un diagramme de contexte système C4 dans la section 3, un diagramme de conteneurs comme niveau 1 de la vue des blocs de construction dans la section 5, avec des diagrammes de composants comme niveau 2, et un diagramme de déploiement C4 dans la section 7. Conservée sous forme de diagrammes en tant que code dans le même dépôt que le texte arc42, toute la description peut être versionnée et revue comme du code.

SI VOUS DÉBUTEZ

Pas à pas : un renvoi, pas une copie

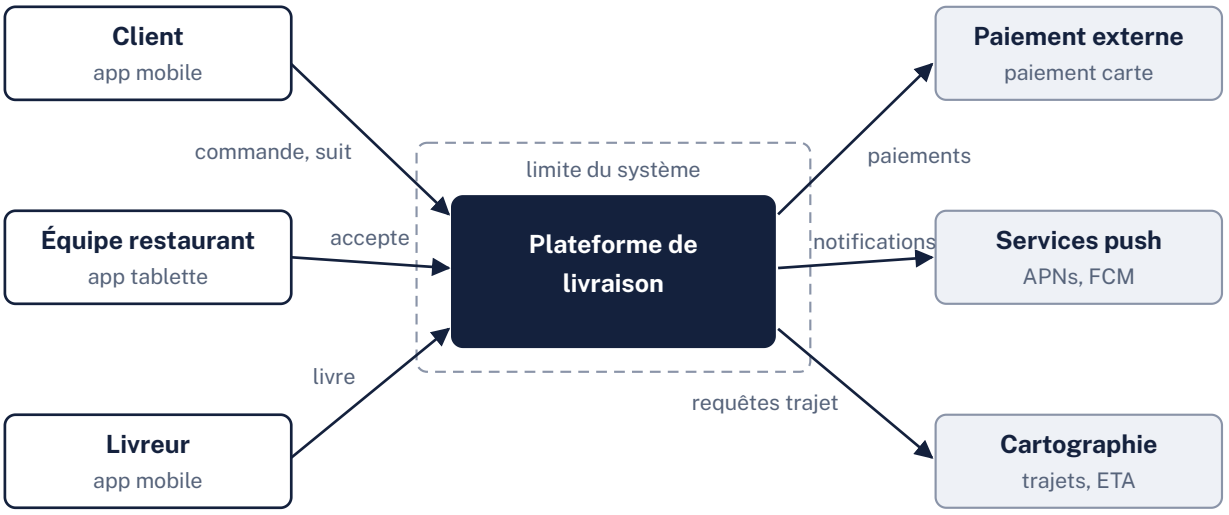
Le point qui fait le plus souvent trébucher ici est qu'arc42 est un ensemble de tiroirs étiquetés, pas un formulaire à remplir. De nombreuses sections devraient contenir un court renvoi vers l'endroit où se trouve déjà l'information réelle, pas une copie de celle-ci.

- L'équipe de livraison de repas tient déjà deux diagrammes C4 (contexte système et conteneurs), un journal des décisions dans docs/decisions et ses scénarios de qualité dans docs/quality.md. Elle les organise maintenant avec arc42.
- Les diagrammes vont dans leurs tiroirs : le diagramme de contexte dans la section 3, et le diagramme de conteneurs comme niveau 1 de la vue des blocs de construction dans la section 5.
- Pour la section 9, décisions d'architecture, quelqu'un suggère de coller le texte intégral de chaque enregistrement de décision, pour que tout se trouve dans un seul document.
- Puis l'ADR-0015 remplace l'ADR-0009, comme dans la section précédente. La pull request qui ajoute l'ADR-0015 modifie aussi le journal des décisions, mais personne ne pense à la copie collée, qui présente toujours l'ADR-0009 comme accepté.
- Un lecteur trouve maintenant deux réponses : la description arc42 dit que les notifications push passent par un SDK tiers, tandis que le journal des décisions dit que l'équipe fait désormais fonctionner sa propre diffusion push. Il ne peut pas savoir laquelle est vraie, et commence à se méfier de toute la description.
- La section 9 contient donc un lien vers docs/decisions et les numéros de quelques enregistrements incontournables. Toutes les autres sections fonctionnent de la même façon : les scénarios de qualité restent dans docs/quality.md, le déploiement dans le dépôt d'infrastructure, et une section qui n'a rien à ajouter est supprimée ou dit pourquoi en une ligne.
- **La leçon :** arc42 indique aux lecteurs où chercher ; il n'a pas à tout contenir lui-même. Donnez à chaque fait un seul emplacement et laissez le modèle y renvoyer : la description reste courte et vraie.

Exemple commenté : arc42 allégé pour la plateforme de livraison de repas

La page ci-dessous est une description arc42 allégée complète de la plateforme de livraison de repas utilisée tout au long de ce chapitre. Elle tient sur une page imprimée plus le diagramme de contexte, et pourtant elle répond aux questions qu'un nouvel ingénieur, un responsable des opérations ou un auditeur poserait en premier, et elle indique où se trouvent les détails.

Section 3 : contexte métier



Le contexte de la plateforme : trois groupes d'utilisateurs à gauche, trois services externes à droite, chaque flèche portant la mention de ce qui est échangé ; les canaux et les protocoles relèvent du contexte technique.

EXEMPLE COMMENTÉ

Exemple commenté : une description arc42 allégée sur une page

Plateforme de livraison de repas, description d'architecture, version revue le 2026-10-01 par l'architecte principal. Les sections qui n'apporteraient rien sont signalées comme telles au lieu d'être laissées vides.

1. Objectifs	Permettre aux clients de commander auprès de restaurants locaux et de suivre la livraison en temps réel. Objectifs de qualité par ordre de priorité : (1) aucune commande n'est perdue ni débitée deux fois ; (2) confirmation de commande avec un p99 inférieur à 1,5 s à 120 commandes/s ; (3) un nouveau consommateur des données de commande peut être ajouté sans modifier le service des commandes. Parties prenantes : clients, restaurants, livreurs, opérations, finance, délégué à la protection des données, prestataire de paiement.
2. Contraintes	Les données de carte ne touchent jamais nos systèmes (champs de carte hébergés par le prestataire de paiement, pour limiter le périmètre PCI DSS). Les données personnelles du Royaume-Uni et de l'UE restent dans des régions de l'UE. Cinq squads ; code côté serveur en Java, Go ou TypeScript uniquement.
3. Contexte	Voir le diagramme de contexte. Interfaces externes : API REST et webhooks du prestataire de paiement ; APNs et FCM pour le push ; fournisseur de cartographie pour les trajets et les ETA. Les responsables et les contrats des interfaces sont listés dans `docs/interfaces.md`.
4. Stratégie de solution	Services distincts pour les commandes, l'affectation et les notifications (objectif 3) ; événements de commande asynchrones via un courtier managé avec une outbox transactionnelle (objectifs 1 et 2, ADR-0012) ; PostgreSQL managé pour les commandes (objectif 1, ADR-0007) ; clés d'idempotence sur chaque appel de paiement (objectif 1).
5. Blocs de construction	Le niveau 1 est le diagramme de conteneurs C4 dans `docs/diagrams/containers.dsl` : trois applications clientes, l'API gateway, les services de commandes, d'affectation et de notification, la base de données des commandes et le topic des événements de commande sur le courtier managé. Le niveau 2 n'existe que pour les commandes, le service le plus risqué.
6. Exécution	Deux scénarios : *passer et payer une commande* (diagramme de séquence) et *délai d'expiration du prestataire de paiement* (montre la nouvelle tentative avec clé d'idempotence et l'état en attente).
7. Déploiement	Un seul fournisseur cloud, deux régions de l'UE, Kubernetes managé et bases de données managées ; défini dans le dépôt d'infrastructure, qui fait foi.

8. Concepts	Modèle du domaine (états du cycle de vie d'une commande) ; idempotence et nouvelles tentatives ; observabilité (identifiants de trace transportés dans chaque événement) ; traitement et conservation des données personnelles.
9. Décisions	Voir le journal des décisions dans `docs/decisions/`. Enregistrements incontournables : ADR-0004, ADR-0007, ADR-0012.
10. Qualité	Scénarios QS-1 à QS-5 dans `docs/quality.md`, chacun avec une mesure de la réponse ; QS-3 (latence de confirmation) et QS-1 (aucune commande perdue) ont des fonctions d'aptitude automatisées.
11. Risques et dette	Un seul fournisseur de courtier (dépendance au fournisseur, ou lock-in, acceptée dans l'ADR-0012) ; l'affectation lit encore directement la base de données des commandes (dette, suppression prévue au T1) ; limites de débit du fournisseur de cartographie aux heures de pointe (risque, responsable : squad Affectation).
12. Glossaire	Commande, panier, acceptation, retrait, remise, affectation, ETA, outbox, clé d'idempotence : définis dans `docs/glossary.md`.

Chaque section est soit remplie brièvement, soit renvoie à une source de vérité unique ; rien n'est copié, donc rien ne peut se désynchroniser.

Les objectifs de qualité de la section 1 orientent la stratégie de la section 4, qui cite les décisions de la section 9 : un lecteur peut faire remonter chaque choix majeur à un objectif.

Autres modèles

Le modèle *Views and Beyond* du SEI. Le Software Engineering Institute publie un modèle de documentation qui suit l'ouvrage *Documenting Software Architectures: Views and Beyond* de Paul Clements et de ses collègues. Son niveau supérieur comporte six parties : une feuille de route et une vue d'ensemble de la documentation, le contexte de l'architecture, les vues, la mise en correspondance des vues, les documents de référence, et un glossaire avec les acronymes. Chaque vue est documentée de la même manière, avec une présentation principale, un catalogue des éléments, un diagramme de contexte, un guide de variabilité et la justification (décrits dans l'encadré « Pour aller plus loin » de la section 4.2). Il est plus complet qu'arc42 et convient aux systèmes de grande taille, à longue durée de vie ou spécifiés par contrat, où chaque vue doit être complète et faire l'objet de renvois croisés ; sa mise en correspondance des vues est un point fort qu'arc42 laisse implicite.

IEEE 1016. La norme IEEE Std 1016-2009, *Software Design Descriptions*, spécifie le contenu et l'organisation d'une description de la conception logicielle (SDD), le document de conception détaillée traditionnellement exigé dans la défense, l'aérospatiale et d'autres projets régis par

des contrats. Elle a emprunté les concepts de partie prenante, de préoccupation, de point de vue et de vue à l'IEEE 1471 (le prédécesseur de la 42010) et liste des points de vue de conception comme le contexte, la composition, le point de vue logique, les dépendances, l'information, les interfaces, l'interaction, la dynamique des états, les algorithmes et les ressources. L'IEEE l'a placée au statut *inactive-reserved* (inactive, réservée) en 2020, mais elle est toujours citée dans des contrats et dans l'enseignement, et d'anciens programmes livrent encore des SDD sous sa forme. Elle est plus proche de la conception détaillée que de l'architecture.

Les documents de conception et les RFC, utilisés par de nombreuses organisations d'ingénierie, sont un autre type de modèle : une proposition pour un changement unique plutôt que la description d'un système entier. Ils sont traités deux sections plus loin, dans communiquer l'architecture.

Conformité à l'ISO/IEC/IEEE 42010

L'ISO/IEC/IEEE 42010:2022 fixe des exigences pour les descriptions d'architecture et pour les cadres de description d'architecture (ADF) : des conventions établies, comme arc42 ou l'approche Views and Beyond, pour produire des descriptions dans une communauté donnée. Elle ne prescrit ni modèle, ni notation, ni processus. Une description est *conforme* lorsqu'elle contient les informations que la norme exige, quel que soit le modèle qui les organise.

Dans les grandes lignes, cela signifie identifier le système décrit et son environnement, identifier les parties prenantes et leurs préoccupations, indiquer quels points de vue sont utilisés et quelles préoccupations chacun encadre, fournir des vues qui suivent ces points de vue, consigner les correspondances (comment les éléments de différentes vues sont liés) et consigner la justification des décisions clés. arc42 ne revendique pas la conformité d'emblée, et les descriptions arc42 brutes manquent souvent d'un lien explicite entre chaque préoccupation et la vue qui la traite. Les lacunes sont minimes et faciles à combler, comme le montre le tableau. La plupart des équipes n'ont pas besoin d'une conformité formelle ; celles des secteurs réglementés ou liées par un contrat en ont parfois besoin, et il est rassurant qu'un modèle léger puisse satisfaire la norme avec quelques ajouts.

Rendre une description arc42 conforme à la 42010

Ce que demande la 42010	Où arc42 le traite	Ce qu'il faut ajouter
Les parties prenantes et leurs préoccupations	Tableau des parties prenantes de la section 1.3 ; objectifs de qualité en 1.2 et en 10	Lister explicitement les préoccupations de chaque partie prenante, pas seulement ses attentes.
Des points de vue, chacun encadrant des préoccupations nommées	Implicites dans les sections 3, 5, 6 et 7	Un court tableau qui nomme chaque point de vue, les préoccupations qu'il encadre et sa notation (par exemple les diagrammes de conteneurs C4).
Des vues qui suivent les points de vue	Sections 3, 5, 6 et 7	Rien, si chaque vue indique quel point de vue elle suit.
Des correspondances entre les vues	En partie dans la section 7 (ce qui s'exécute où)	Des tableaux de correspondance, par exemple bloc de construction vers nœud de déploiement, bloc de construction vers équipe responsable.
La justification des décisions clés	Sections 4 et 9 (ADR)	Relier chaque décision aux préoccupations qu'elle traite.

ATTENTION

Erreurs courantes avec les modèles

- Remplir chaque section parce qu'elle existe. Une section vide avec une ligne qui explique pourquoi vaut mieux qu'une page de remplissage.
- Copier les exigences, les décisions et les détails d'infrastructure dans le modèle, où ils se désynchronisent des originaux. Faites plutôt des liens.
- Une vue des blocs de construction avec des dizaines de cases au niveau 1. Le niveau 1 doit tenir sur un écran ; le détail a sa place dans les niveaux inférieurs, et seulement là où il est rentable.
- Rédiger la description une fois au départ et plus jamais ensuite. Placez-la sous gestion de versions, revoyez-la à chaque changement significatif et affichez une date de «dernière revue».
- Mélanger des modèles sans carte. Si vous utilisez arc42 pour la structure et C4 pour les diagrammes, dites-le dans la section 1, pour que les lecteurs connaissent les conventions.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Quelle section d'arc42 devient généralement la plus volumineuse, et comment l'empêcher de grossir sans contrôle ?

La section 5, la vue des blocs de construction. Limitez le niveau 1 à un diagramme qui tient sur un écran et n'affinez aux niveaux 2 et 3 que pour les blocs de construction complexes ou risqués.

▼ Où les objectifs de qualité apparaissent-ils dans arc42, et en quoi les sections 1 et 10 diffèrent-elles ?

La section 1 liste les trois à cinq principaux objectifs de qualité par ordre de priorité, pour l'orientation ; la section 10 les affine, ainsi que toute autre qualité, en un arbre de qualité et en scénarios concrets et testables.

▼ Votre équipe conserve déjà des ADR dans le dépôt. Que doit contenir la section 9 d'arc42 ?

Un lien vers le journal des décisions, avec éventuellement une courte liste des décisions que tout nouveau venu doit lire, plutôt qu'une copie des enregistrements.

▼ Utiliser arc42 rend-il une description d'architecture conforme à l'ISO/IEC/IEEE 42010 ?

Pas automatiquement. La conformité dépend du contenu : les parties prenantes, les préoccupations, les points de vue, les vues, les correspondances et les justifications doivent être explicites, ce qui exige généralement quelques petits ajouts à arc42.

▼ À vous : Une équipe de trois personnes gère un petit service de cartes de restaurants. Son responsable lui demande de remplir les douze sections d'arc42. Que conseilleriez-vous à la place, et comment traiter les sections dont elle n'a pas besoin ?

Utilisez un arc42 allégé. Rédigez trois à cinq scénarios de qualité, le contexte, une brève stratégie de solution, les blocs de construction de niveau 1 et les principaux concepts transversaux, idéalement sur environ une page. Faites des liens vers les ADR et le code d'infrastructure au lieu de les copier. Chaque section est facultative : indiquez donc délibérément qu'une section n'est pas nécessaire, plutôt que de la laisser vide ou de la remplir pour la forme. arc42 dit où vont les informations ; il n'a jamais été conçu pour exiger du volume.

SOURCES

- arc42 overview: https://arc42.org/overview
- arc42 documentation: section 1, Introduction and Goals: https://docs.arc42.org/section-1/

- arc42 FAQ: What is the minimal amount of an arc42 documentation?: https://faq.arc42.org/questions/B-4/
- SEI Software Architecture Documentation Template (Views and Beyond): https://wiki.sei.cmu.edu/confluence/display/SAD/Software+Architecture+Documentation+Template
- IEEE 1016-2009: Software Design Descriptions: https://standards.ieee.org/ieee/1016/4502/

SECTION 4.7 SUR 4.8 · INTERMÉDIAIRE · 22 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Diagrammes en tant que code et documentation à jour

AVANT DE LIRE

Cette section s'appuie sur :

- 0.8 Comment les logiciels sont construits
- 4.1 Pourquoi documenter
- 4.3 Le modèle C4

Termes déjà vus qu'elle utilise : Gestion de versions, Pull request, Intégration continue (CI), Diagramme de séquence, Documentation vivante

L'IDÉE EN BREF

Imaginez un livre de recettes de famille dans lequel chaque modification est notée au crayon par la personne qui cuisine, sans aucune trace de qui a changé quoi. Au bout de quelques années, plus personne ne s'y fie. Imaginez maintenant que les recettes sont dactylographiées, que chaque modification est proposée, vérifiée par un second cuisinier et datée, et que le livre imprimé est réimprimé automatiquement après chaque modification approuvée. Les diagrammes en tant que code appliquent cette discipline aux diagrammes d'architecture : ils sont écrits sous forme de courts fichiers texte, conservés avec le logiciel, vérifiés quand le logiciel change et redessinés automatiquement.

CE QU'IL FAUT SAVOIR

- Les diagrammes en tant que code consistent à écrire les diagrammes dans un langage textuel et à laisser un outil les dessiner, afin qu'ils puissent être versionnés, comparés (diff), relus dans des pull requests et rendus par le pipeline de build.
- PlantUML (2009) excelle en UML, en particulier pour les diagrammes de séquence ; Mermaid (2014) s'affiche directement dans le Markdown sur GitHub et GitLab ; D2 (2022) offre des mises en page soignées ; Structurizr DSL définit un seul modèle C4 à partir duquel de nombreuses vues sont générées.
- Les outils de dessin ne connaissent que des formes, si bien qu'un même élément peut diverger d'un diagramme à l'autre ; les outils de modélisation conservent les éléments et leurs relations dans un modèle, de sorte que toutes les vues restent cohérentes et que le modèle peut être interrogé et vérifié.

- Un pipeline de documentation en tant que code rend les diagrammes, vérifie les liens, construit le site de documentation à chaque pull request et publie depuis main, si bien que les lecteurs voient une documentation qui correspond au code.
- La documentation reste à jour grâce à des habitudes, pas à des outils : des responsables nommés, une relecture en même temps que les modifications du code, des dates de dernière revue, des liens entre la documentation, le code et les décisions, et la suppression des pages périmées (les ADR sont remplacés, pas supprimés).

Ce que signifient les diagrammes en tant que code

Les diagrammes en tant que code (diagrams as code) consistent à écrire un diagramme sous forme d'un court fichier texte, dans un petit langage, et à laisser un outil le mettre en page et le dessiner. Au lieu de faire glisser des boîtes sur un canevas, vous écrivez `ordering → db: SQL`, et l'outil produit l'image. Le fichier texte vit dans le dépôt de code et il est traité comme du code : versionné, relu, vérifié par des builds automatisés et publié par le pipeline.

C'est une composante de la documentation en tant que code (docs as code), pratique défendue par la communauté Write the Docs, qui consiste à produire la documentation avec les mêmes outils et le même flux de travail que le logiciel : outils de suivi des tickets, gestion de versions (généralement Git), balisage en texte brut comme Markdown ou AsciiDoc, revue de code et tests automatisés. Les sections précédentes ont produit la matière : vues et diagrammes C4, UML là où la précision compte, ADR et structure arc42. Cette section porte sur la manière de garder cette matière exacte pendant toute la vie du système, et c'est là que la plupart des documentations d'architecture échouent. Les raisons de cet échec ont été exposées dans pourquoi documenter, et dans quelle mesure : des wikis périmés, de gros documents dont personne n'est responsable et des diagrammes qui ne correspondent plus au système.

Pourquoi mettre les diagrammes en texte

Gestion de versions. Un diagramme en texte a un historique. Vous pouvez voir qui l'a modifié, quand et pourquoi, et comparer deux versions quelconques ligne par ligne ; une image binaire ou un fichier au format d'un outil de dessin montre seulement que quelque chose a changé. Vous pouvez aussi créer une branche : le diagramme d'une modification proposée vit sur la même branche que le code qui la met en œuvre.

Relecture dans les pull requests. Quand un développeur ajoute une nouvelle dépendance entre des services, la modification du diagramme apparaît dans la même pull request que le code, sous forme de diff textuel lisible. Les relecteurs peuvent demander si la nouvelle flèche est voulue : c'est une revue d'architecture au moment où elle coûte le moins cher.

Cohérence. L'outil applique les mêmes formes, polices et règles de mise en page à chaque diagramme, si bien que des diagrammes dessinés par vingt personnes ressemblent toujours à un ensemble unique. Avec un langage de modélisation (voir plus bas), le même élément apparaît avec le même nom et la même description dans chaque vue.

Génération. Du texte peut être produit par des programmes. Les diagrammes peuvent être générés à partir du code lui-même (graphes de dépendances), de l'infrastructure en tant que code (diagrammes de déploiement), des spécifications d'API ou d'un modèle central, puis vérifiés automatiquement par rapport au système.

Les compromis sont réels. La mise en page automatique est pratique mais parfois laide, et lutter contre un moteur de mise en page pour placer une seule boîte peut prendre plus de temps que de la dessiner ; la plupart des outils proposent des indications, comme la direction ou le regroupement, pas des positions exactes. Les langages textuels demandent un apprentissage aux personnes qui ne codent pas, comme les product managers. Et un beau croquis au tableau blanc pendant une discussion de conception reste la manière la plus rapide de réfléchir ; les diagrammes en tant que code sont destinés aux diagrammes que vous comptez conserver.

Quatre langages, un exemple

Quatre langages dominent. Chaque exemple ci-dessous décrit une partie de la même plateforme de livraison de repas.

PlantUML, créé par Arnaud Roques et publié pour la première fois en 2009, est le plus ancien des quatre. Il couvre tous les diagrammes UML courants (séquence, composants, déploiement, états, activités, classes) et bien d'autres, et utilise Graphviz pour mettre en page la plupart des types de diagrammes. La bibliothèque C4-PlantUML ajoute les formes C4. Ses diagrammes de séquence sont particulièrement réussis, ce qui en fait un choix naturel pour les scénarios d'exécution de l'essentiel d'UML.

```
@startuml
title Place an order (happy path)
actor Customer
participant "Customer app" as App
participant "Ordering service" as Ord
participant "Payment provider" as Pay
queue "Order events topic" as MB

Customer -> App : Tap "Pay"
App -> Ord : POST /orders (HTTPS)
Ord -> Pay : Authorise payment
Pay --> Ord : Authorised
Ord ->> MB : OrderPlaced (async)
Ord --> App : 201 Created
@enduml
```

PlantUML : un diagramme de séquence de la prise de commande ; `->>` dessine un message asynchrone et `-->` une réponse

Mermaid, créé par Knut Sveidqvist en 2014, est écrit en JavaScript et s'affiche dans le navigateur. Son principal atout est sa portée : GitHub affiche les blocs Mermaid dans les fichiers Markdown depuis février 2022, et GitLab ainsi que de nombreux wikis et outils de prise de notes font de même, si bien qu'un diagramme dans un README ne demande aucune étape de build. Il prend en charge, entre autres, les organigrammes et les diagrammes de séquence, de classes, d'états et entité-association, et propose une prise en charge expérimentale de C4. Sa maîtrise de la mise en page est limitée pour les grands diagrammes.

```
flowchart LR
    app["Customer app<br/>Kotlin, Swift"]
    gw["API gateway"]
    ord["Ordering service<br/>Java"]
    db[("Order database<br/>PostgreSQL")]
    topic[["Order events<br/>Kafka topic"]]
    disp["Dispatch service<br/>Go"]
    notif["Notification service<br/>Node.js"]

    app -->|HTTPS/JSON| gw
    gw -->|HTTPS/JSON| ord
    ord -->|SQL| db
    ord -->|publishes OrderPlaced| topic
    disp -->|consumes| topic
    notif -->|consumes| topic
```

Mermaid : un organigramme au niveau des conteneurs ; `[( )]` dessine un cylindre de base de données et les libellés se placent sur les flèches ; le topic est dessiné comme un conteneur, comme le recommande la section 4.3

Structurizr DSL, créé par Simon Brown en parallèle du modèle C4, est d'une autre nature. Il ne décrit pas un diagramme ; il décrit un *modèle* (personnes, systèmes logiciels, conteneurs, composants et relations entre eux), puis définit des *vues* de ce modèle. Chaque élément est défini une seule fois, avec son nom, sa description et sa technologie, et apparaît dans chaque vue qui l'inclut. Les relations entre conteneurs sont automatiquement déduites au niveau du système, si bien que la vue de contexte n'a pas besoin de flèches séparées. Le DSL peut aussi rattacher au workspace une documentation Markdown et un dossier d'ADR, et l'outillage peut exporter les vues vers PlantUML, Mermaid, D2 et d'autres formats.

```
workspace "Food delivery" {
    model {
        customer = person "Customer" "Orders food from local restaurants."
        payments = softwareSystem "Payment provider" "Takes card payments." "External"
        platform = softwareSystem "Food-delivery platform" {
            app = container "Customer app" "Ordering and tracking." "Kotlin, Swift"
            gateway = container "API gateway" "Authenticates and routes requests."
"Managed API gateway"
            ordering = container "Ordering service" "Accepts and tracks orders." "Java"
            db = container "Order database" "Orders and their states." "PostgreSQL"
"Database"
            orderEvents = container "Order events" "Order lifecycle events." "Kafka
topic" "Topic"
            dispatch = container "Dispatch service" "Assigns couriers." "Go"
        }
        customer → app "Places and tracks orders using"
        app → gateway "Makes API calls to" "JSON/HTTPS"
        gateway → ordering "Routes requests to" "JSON/HTTPS"
        ordering → db "Reads from and writes to" "SQL/TCP"
        ordering → payments "Takes card payments using" "JSON/HTTPS"
        ordering → orderEvents "Publishes order events to" "Kafka"
        dispatch → orderEvents "Consumes order events from" "Kafka"
    }
    views {
        systemContext platform "Context" {
            include *
            autoLayout lr
        }
        container platform "Containers" {
            include *
            autoLayout lr
        }
        styles {
            element "Database" {
                shape cylinder
            }
            element "Topic" {
                shape pipe
            }
        }
    }
}
```

Structurizr DSL : un modèle et deux vues de la plateforme de livraison de repas ; la vue de contexte est générée à partir des mêmes définitions

D2 (Declarative Diagramming), publié en open source par Terrastruct en 2022, est le plus récent. Il vise un rendu net et moderne et une syntaxe lisible, prend en charge les conteneurs imbriqués (écrits avec des accolades et désignés par des noms à points), et offre le choix entre

plusieurs moteurs de mise en page : dagre et ELK sont fournis et gratuits, et TALA, conçu pour les diagrammes d'architecture logicielle, est un moteur commercial distinct.

```
direction: right

customer: Customer {shape: person}
payments: Payment provider

platform: Food-delivery platform {
  app: Customer app
  ordering: Ordering service
  db: Order database {shape: cylinder}
  events: Order events topic {shape: queue}
  dispatch: Dispatch service

  app → ordering: Places order (HTTPS)
  ordering → db: SQL
  ordering → events: Publishes OrderPlaced
  dispatch → events: Consumes OrderPlaced
}

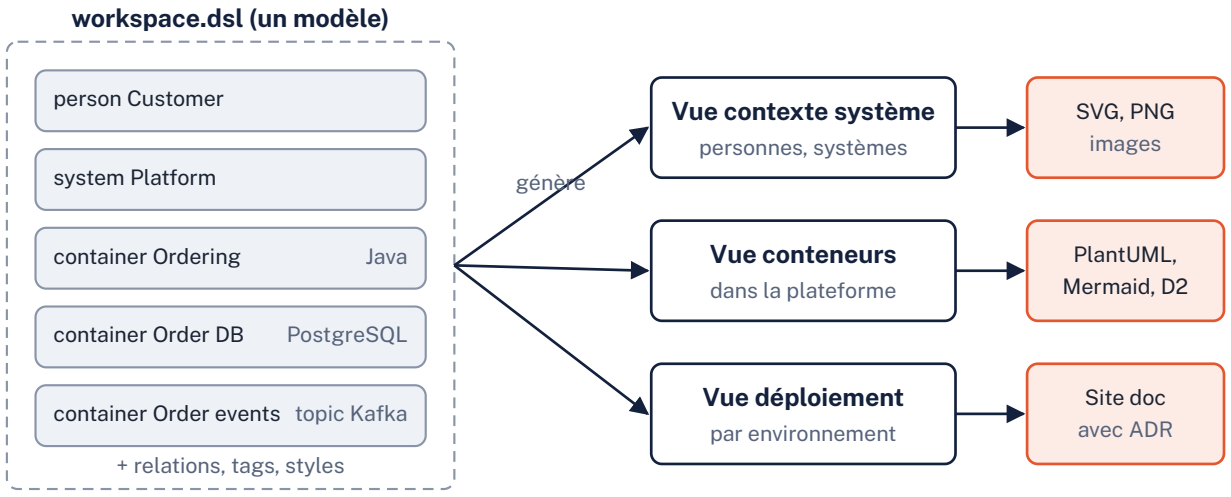
customer → platform.app: Places orders using
platform.ordering → payments: Authorise (HTTPS)
```

D2 : conteneurs imbriqués, formes et connexions libellées

Comparaison des quatre

	PlantUML	Mermaid	Structurizr DSL	D2
Décrit	Un diagramme par fichier	Un diagramme par bloc	Un modèle et des vues de ce modèle	Un diagramme par fichier
Point fort	UML, en particulier les diagrammes de séquence	Diagrammes dans le Markdown, sans étape de build	Diagrammes C4 cohérents à plusieurs niveaux	Diagrammes d'architecture généraux et attrayants
Rendu	Outil ou serveur Java ; Graphviz pour la plupart des mises en page	Dans le navigateur ; natif sur GitHub et GitLab	Outils Structurizr ; exports vers d'autres formats	Outil en ligne de commande ; choix du moteur de mise en page
Points de vigilance	Aspect par défaut daté ; mise en page guidée seulement par des indications	Maîtrise limitée de la mise en page des grands diagrammes	Concepts C4 uniquement ; outillage spécifique nécessaire	Le meilleur moteur de mise en page est commercial

Un modèle, de nombreuses vues



Renommez Ordering une fois dans le modèle : toutes les vues et exports suivent.

Un langage de modélisation définit chaque élément une seule fois ; les vues et les exports sont générés, ils ne peuvent donc pas se contredire.

Outils de dessin et outils de modélisation

La différence entre Structurizr et les trois autres illustre la distinction entre outils de dessin et outils de modélisation présentée à la section 4.3. Un outil de dessin (diagrams.net, Excalidraw, Visio, un tableau blanc en ligne, mais aussi PlantUML, Mermaid et D2) connaît des formes, des lignes et du texte. Une boîte libellée «Service Commandes» dans un diagramme n'a aucun lien avec la boîte portant le même libellé dans un autre. Renommez le service, et vous devez retrouver et corriger chaque diagramme à la main ; faites une seule faute de frappe, et vous avez, pour l'outil, deux services.

Un outil de modélisation (Structurizr, Archi pour les modèles ArchiMate, Sparx Enterprise Architect pour UML et SysML, Capella) conserve un modèle : un référentiel d'éléments et de relations dotés de propriétés. Les diagrammes sont des vues sur ce modèle. Renommez un élément, et toutes les vues changent. Le modèle peut être interrogé («quels conteneurs dépendent du broker ?»), vérifié («aucun composant de l'app client ne doit appeler la base de données») et comparé au système en fonctionnement. Le prix à payer est davantage de structure en amont et, avec les outils commerciaux lourds, des licences, de la formation et une tendance à modéliser bien plus que ce que quiconque lit.

Une règle pratique : utilisez des outils de dessin, textuels ou graphiques, pour les croquis, les explications ponctuelles et les diagrammes autonomes, comme un diagramme de séquence isolé dans un ADR. Utilisez un modèle quand les mêmes éléments apparaissent dans de nombreux diagrammes, quand plusieurs équipes contribuent, ou quand vous voulez vérifier la documentation par rapport au code. Beaucoup d'équipes combinent les deux : un modèle Structurizr pour la structure C4, et PlantUML ou Mermaid pour quelques diagrammes de séquence et d'états.

SI VOUS DÉBUTEZ

Pas à pas : un renommage, trois fichiers

L'idée la plus difficile de cette section est qu'écrire des diagrammes sous forme de texte ne suffit pas, à lui seul, à les garder cohérents. La section 4.3 a réparti les outils entre outils de dessin et outils de modélisation, et cette section ajoute que PlantUML, Mermaid et D2 sont des outils de dessin, même si on les tape au clavier, et que seul un outil de modélisation comme Structurizr sait que deux boîtes désignent la même chose.

- L'équipe de la plateforme de livraison de repas tient trois diagrammes en texte : le diagramme de séquence PlantUML de la prise de commande, l'organigramme des conteneurs en Mermaid et le diagramme D2. Chaque fichier écrit lui-même le libellé « Ordering service ».
- Supposons que l'équipe renomme le service « Order service ». Quelqu'un doit retrouver chaque fichier qui le mentionne et modifier chaque libellé à la main, car rien ne relie les trois boîtes : ce n'est que du texte qui se trouve être identique.
- Oubliez un fichier, et les diagrammes se contredisent. Faites une faute de frappe dans un libellé, par exemple « Order servce », et pour cet outil il s'agit d'un autre service. Le pipeline rend quand même chaque diagramme, car chaque fichier est valide en lui-même.
- Dans le fichier Structurizr DSL, le service Commandes est défini une seule fois, dans le modèle, avec son nom, sa description et sa technologie. La vue des conteneurs, et toute autre vue qui l'inclut, est générée à partir de cette définition unique.
- Changez le nom à cet endroit, et chaque vue le reprend au build suivant. Comme le modèle contient les éléments et les relations, on peut aussi lui demander quels conteneurs dépendent du topic des événements de commande, ou le vérifier par rapport à une règle.
- **La leçon :** le texte donne aux diagrammes un historique, une relecture dans les pull requests et un rendu automatique ; seul un modèle fait qu'un même élément signifie la même chose partout. Utilisez-en un quand les mêmes parties apparaissent dans de nombreux diagrammes.

Générer la documentation dans le pipeline

Dès que la documentation est du texte dans le dépôt, le pipeline d'intégration continue vu au chapitre 0 peut la construire et la vérifier comme du code. Un pipeline type rend chaque diagramme (en faisant échouer le build en cas d'erreur de syntaxe), vérifie les liens, construit un site de documentation statique avec un générateur comme MkDocs, Antora ou Docusaurus,

et le publie quand les modifications arrivent sur la branche principale. Les lecteurs voient alors toujours la documentation qui correspond au code de main, et personne n'a besoin de penser à exporter une image pour l'envoyer sur un wiki.

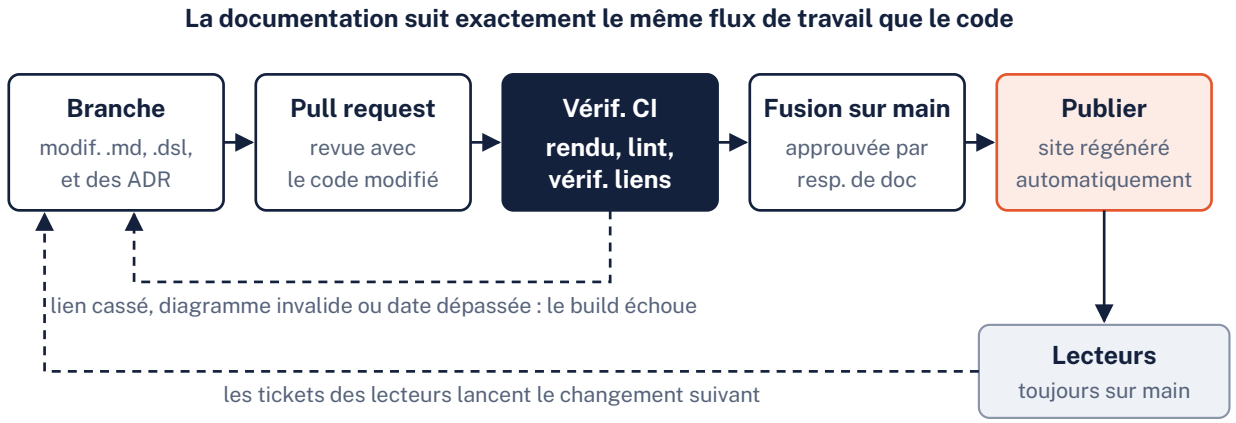
D'autres vérifications peuvent s'ajouter à mesure que l'équipe gagne en maturité : un correcteur orthographique et un linter de rédaction (Vale est populaire), une vérification que chaque ADR a un statut valide, un avertissement quand la date de «dernière revue» d'une page dépasse son intervalle de revue, et des vérifications d'architecture qui comparent les dépendances documentées avec les dépendances réelles du code, ce qui est une forme de fonction d'aptitude. Certaines équipes génèrent une partie de la documentation à partir du système lui-même : graphes de dépendances issus du build, référence d'API issue des spécifications OpenAPI, diagrammes de déploiement issus de l'infrastructure en tant que code.

```
name: architecture-docs
on:
  pull_request:
    paths: ["docs/**"]
  push:
    branches: [main]

jobs:
  docs:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      # assumes plantuml, d2, lychee and mkdocs are available on the runner
      - name: Render diagrams (fails on syntax errors)
        run: |
          plantuml -tsvg docs/diagrams/*.puml
          for f in docs/diagrams/*.d2; do d2 "$f" "${f%.d2}.svg"; done
      - name: Check links
        run: lychee --offline docs/
      - name: Build site (warnings such as broken links fail the build)
        run: mkdocs build --strict
      - name: Publish
        if: github.ref == 'refs/heads/main'
        run: ./scripts/publish-docs.sh
```

Un workflow GitHub Actions qui rend les diagrammes, vérifie les liens, construit le site de documentation à chaque pull request et publie depuis main

Le flux de travail de la documentation en tant que code



Les modifications de la documentation suivent le même chemin que celles du code : branche, relecture, vérifications automatisées, fusion et publication automatique.

Garder la documentation à jour

Ce ne sont pas les outils qui gardent la documentation exacte, ce sont les habitudes. Cinq habitudes font la différence.

Responsabilité. Chaque document a un responsable nommé, normalement l'équipe responsable du code qu'il décrit. Dans GitHub et GitLab, un fichier `CODEOWNERS` peut faire de cette équipe un relecteur obligatoire pour les modifications de son dossier `docs/`. Les documents qui appartiennent à «tout le monde» ne sont maintenus par personne.

Relecture avec les modifications du code. Un modèle de pull request qui demande «Cette modification touche-t-elle une interface, une dépendance, un déploiement ou une décision ? Si oui, quel document ou quel ADR avez-vous mis à jour ?» attrape la plupart des dérives au moment où elles se produisent. Certaines équipes vont plus loin et bloquent la fusion des fonctionnalités dépourvues de documentation, une pratique que mentionne le guide de Write the Docs.

Dates de dernière revue. Un petit en-tête sur chaque page indique son responsable, la date de sa dernière vérification par rapport à la réalité et la fréquence à laquelle elle doit être revue. Les lecteurs peuvent juger dans quelle mesure s'y fier, et le pipeline peut signaler les pages en retard. Une page qui indique «dernière revue : 2023» est honnête ; une page qui décrit en silence le système de 2023 est un piège.

Relier documentation et code. Les pages pointent vers le code qu'elles décrivent (chemins, modules), les commentaires du code pointent vers les ADR, et les messages de commit citent les enregistrements qu'ils mettent en œuvre. Des liens dans les deux sens rendent évident quels documents une modification peut affecter.

Supprimer ce qui est périmé. Une documentation fausse est pire que pas de documentation du tout, car les lecteurs agissent en fonction d'elle. Quand une page ne décrit plus le système et que personne ne la corrigera, supprimez-la ou déplacez-la dans une archive clairement signalée. Les ADR font exception : ce sont des traces de l'histoire, ils sont donc remplacés ou dépréciés, pas supprimés. C'est l'idée de documentation vivante que Cyrille Martraire a développée dans son livre de 2019 portant ce titre : une documentation qui évolue au même rythme que le système, idéalement parce qu'une grande partie en est dérivée du système lui-même.

```
---
title: "Ordering service: building block view (level 2)"
owner: "@ordering-team"
last_reviewed: 2026-09-14
review_every: 6 months
code: src/ordering/
decisions: [ADR-0007, ADR-0012]
---
```

Front matter en tête d'une page Markdown : responsable, date et intervalle de revue, et liens vers le code et les décisions qu'elle décrit

Liste de contrôle : une documentation qui reste à jour

Utilisez-la lors de la mise en place de la documentation d'architecture d'un dépôt, et revenez-y tous les quelques mois.

- [ ] **La documentation vit dans le dépôt qu'elle décrit**
  Pages Markdown ou AsciiDoc, sources des diagrammes et ADR dans `docs/`, pas dans un wiki séparé que personne ne met à jour dans la même modification.
- [ ] **Les sources des diagrammes sont du texte et sont rendues par le pipeline**
  Pas d'images exportées à la main qui prennent discrètement du retard sur leurs sources ; le build échoue sur un diagramme invalide.
- [ ] **Chaque élément est défini une seule fois là où les diagrammes se recoupent**
  Utilisez un modèle (comme Structurizr DSL) quand les mêmes conteneurs apparaissent dans plusieurs diagrammes.
- [ ] **Chaque page a un responsable et une date de dernière revue**
  Garanti par `CODEOWNERS` et par une vérification du pipeline qui avertit quand l'intervalle de revue est dépassé.
- [ ] **Les pull requests posent la question de la documentation**
  Le modèle demande si une interface, une dépendance, un déploiement ou une décision a changé, et quel document a été mis à jour.
- [ ] **Les liens sont vérifiés automatiquement**
  Les liens internes cassés et les références à des fichiers supprimés font échouer le build.
- [ ] **La documentation pointe vers le code et le code vers les décisions**
  Des chemins dans la documentation, des numéros d'ADR dans les commentaires du code et les messages de commit.
- [ ] **Le site publié est reconstruit à partir de main**
  Les lecteurs voient toujours la version qui correspond au code déployé, sans étape de publication manuelle.
- [ ] **Les pages périmées sont supprimées ou archivées**
  Les pages que personne ne corrigera sont retirées ; les ADR sont remplacés ou dépréciés, jamais supprimés.
- [ ] **Au moins une vérification compare la documentation à la réalité**
  Par exemple, un test qui échoue si une dépendance apparaît dans le code sans que le diagramme de conteneurs la montre.

ATTENTION

Erreurs courantes

- Choisir un outil avant de décider quoi documenter. L'outil compte bien moins que les habitudes de responsabilité et de relecture.
- Tout mettre dans une seule source de diagramme géante que personne ne peut lire, ni comme texte ni comme image.
- Committer des images rendues sans leurs sources, ou des sources sans build qui les rend, si bien que les deux divergent.
- Prendre les diagrammes en tant que code comme prétexte pour se passer de légendes, de titres et de libellés : les règles de notation C4 s'appliquent toujours aux diagrammes générés.
- Rendre facultatives les modifications de documentation lors de la relecture, si bien qu'elles sont toujours les premières sacrifiées sous la pression des délais.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Que peut voir un relecteur dans une pull request quand un diagramme est du code, et qu'il ne peut pas voir quand c'est une image exportée ?

Un diff lisible, ligne par ligne, de ce qui a changé, comme une nouvelle relation entre deux services, dans la même pull request que le code qui l'a introduite.

▼ Pourquoi renommer un service demande-t-il plus de travail avec un outil de dessin qu'avec un outil de modélisation ?

Dans un outil de dessin, chaque diagramme contient sa propre copie du libellé, si bien qu'il faut retrouver et modifier chaque diagramme ; dans un outil de modélisation, l'élément est défini une seule fois et chaque vue reprend le nouveau nom.

▼ Pourquoi faut-il supprimer la documentation périmée plutôt que la laisser en place, et quelle est l'exception ?

Les lecteurs agissent en fonction de ce qu'ils lisent, si bien qu'une documentation fausse entraîne de mauvaises décisions. Les ADR font exception : ils consignent l'histoire et sont plutôt marqués comme remplacés ou dépréciés.

▼ **Citez deux vérifications qu'un pipeline de documentation peut exécuter en plus du rendu des diagrammes.**

La vérification des liens, le linting de la rédaction ou de l'orthographe, un build strict du site, un avertissement pour les dates de revue dépassées, et la comparaison des dépendances documentées avec celles du code.

▼ **À vous : Votre équipe a huit diagrammes dans un outil de dessin. Après le renommage d'un service, trois affichent encore l'ancien nom, et une flèche d'affectation diffère entre deux d'entre eux. Que changeriez-vous, dans les outils et dans les habitudes, et pourquoi ?**

Les outils de dessin ne connaissent que des formes : chaque image dérive donc séparément. Passez à un modèle, comme le Structurizr DSL, conservé dans le dépôt : définissez chaque élément une seule fois et générez les vues, de sorte qu'un renommage atteigne tous les diagrammes. Produisez-les dans le pipeline à chaque pull request. Les outils seuls ne suffisent pas : nommez donc un responsable, relisez la documentation avec les changements de code, affichez une date de dernière revue et supprimez les pages périmées. PlantUML ou Mermaid conviendraient à des diagrammes de séquence ponctuels.

SOURCES

- Docs as Code, Write the Docs: https://www.writethedocs.org/guide/docs-as-code/
- Structurizr DSL language reference: https://docs.structurizr.com/dsl/language
- Mermaid documentation: https://mermaid.js.org/
- PlantUML: https://plantuml.com/
- Include diagrams in your Markdown files with Mermaid, GitHub Blog (2022): https://github.blog/2022-02-14-include-diagrams-markdown-files-mermaid/

Communiquer l'architecture à différents publics

AVANT DE LIRE

Cette section s'appuie sur :

- 1.5 Parties prenantes et préoccupations
- 1.6 Décisions et compromis
- 4.3 Le modèle C4

Termes déjà vus qu'elle utilise : Compromis, Préoccupation, Modèle C4, Matrice de décision pondérée, Dernier moment responsable

L'IDÉE EN BREF

Un guide fait visiter la même cathédrale différemment à des écoliers, à des architectes et à des pèlerins : des faits différents, des mots différents, des parcours différents à travers l'édifice. Les architectes logiciels doivent faire de même. Une directrice financière veut savoir ce que coûte une conception et ce qui pourrait mal tourner ; un développeur veut savoir comment les éléments s'assemblent ; un auditeur veut des preuves. Une bonne communication commence par se demander qui écoute et ce que ces personnes doivent décider, puis leur montre exactement cela, clairement et honnêtement, inconvénients compris.

CE QU'IL FAUT SAVOIR

- Communiquer l'architecture est une tâche architecturale essentielle ; commencez chaque explication en nommant le public et ce qu'il doit décider ou faire ensuite.
- Les dirigeants ont besoin de la recommandation, du coût, du risque et de la décision attendue ; les développeurs ont besoin de la structure, des interfaces et des raisons ; l'exploitation, la sécurité et les auditeurs ont chacun besoin de vues sur les défaillances, les frontières de confiance et les preuves.
- Expliquez les compromis sous forme d'options, de critères, de recommandation et de conséquences, en commençant par la réponse et en disant clairement ce à quoi l'on renonce.
- Un résumé d'architecture en une page et un récit numéroté qui suit une requête rendent n'importe quel système compréhensible ; libellez chaque boîte et chaque ligne quand vous dessinez, et consignez les conclusions après la séance au tableau blanc.

- Les documents de conception et les RFC proposent une modification par écrit, pour commentaires, avant qu'elle soit construite ; les ADR consignent ensuite les décisions de façon permanente.
- Des diagrammes clairs ont un titre, une légende, une notation cohérente, peu de couleurs mais porteuses de sens, un seul sens de lecture et des flèches libellées ; les désaccords se règlent en convenant des critères, en réunissant des preuves, en recherchant un consensus approximatif et en consignant les avis divergents.

Communiquer, c'est le métier

Une architecture que seul son architecte comprend a échoué, si bonne soit-elle. Les décisions façonnent ce que des dizaines de personnes construisent, exploitent, financent et auditent, et elles ne fonctionnent que si ces personnes les comprennent assez bien pour agir en conséquence et pour remarquer quand la réalité s'en écarte. C'est pourquoi le programme de l'iSAQB compte la communication de l'architecture parmi les tâches essentielles de l'architecte, et pourquoi le rôle de l'architecte décrit au chapitre 1 repose autant sur les compétences relationnelles.

Les sections précédentes de ce chapitre ont fourni la matière : vues, diagrammes C4 et UML, ADR, structure arc42 et un pipeline qui les tient à jour. Cette section porte sur l'usage de cette matière avec des personnes. La compétence centrale est la même dans toutes les situations : déterminer qui est le public, déterminer ce qu'il doit faire après vous avoir entendu, et lui donner exactement ce que cela exige, sous une forme qu'il peut assimiler. L'image de l'*ascenseur de l'architecte* (architect elevator) de Gregor Hohpe (*The Software Architect Elevator*, 2020) résume cette amplitude : l'architecte circule entre la salle du conseil, où la stratégie se définit en termes métier, et la salle des machines, où les systèmes sont construits, et doit être compris à chaque étage.

S'adapter au public

Chaque public a ses propres préoccupations, exactement au sens des parties prenantes et préoccupations, ainsi que son propre vocabulaire et sa propre capacité d'attention. Le tableau est un point de départ ; la vraie compétence consiste à se demander, avant de préparer quoi que ce soit : «que va décider ou faire différemment cette personne après cette conversation ?»

Ce dont les différents publics ont besoin

Public	Ce qui l'intéresse	Ce qu'il faut montrer	Ce qu'il faut éviter
Dirigeants et sponsors	Coût, risque, délai de mise sur le marché, options stratégiques, ce qu'ils doivent décider	Une page : la recommandation, l'impact métier, le coût, les principaux risques et la décision que vous attendez d'eux ; tout au plus un diagramme de contexte système	Des noms de technologies sans signification métier ; des diagrammes de composants ; un tour des options avant la réponse
Product managers	Ce qui devient possible ou plus difficile, les dates de livraison, le comportement visible par les utilisateurs	Les capacités et les contraintes en termes de produit ; l'effet de l'architecture sur la feuille de route ; le récit d'exécution d'un parcours utilisateur	La structure interne pour elle-même
Développeurs	Comment construire dans le cadre de l'architecture : frontières, interfaces, conventions, pourquoi les choses sont comme elles sont	Diagrammes C4 de conteneurs et de composants, diagrammes de séquence, ADR, exemples de code, journal des décisions	Des principes vagues sans exemples ; des diagrammes qui ne correspondent pas au code
Exploitation et SRE	Déploiement, modes de défaillance, supervision, capacité, reprise	Diagrammes de déploiement, scénarios de défaillance à l'exécution, SLO, runbooks, cartes des dépendances	Des flux limités au chemin nominal
Sécurité	Frontières de confiance, flux de données, identités, surface d'attaque	Diagrammes de flux de données avec les frontières de confiance indiquées, flux d'authentification et d'autorisation, modèle de menaces	Des diagrammes qui cachent où vont les données personnelles ou de paiement
Auditeurs et régulateurs	La preuve que les contrôles existent et sont appliqués ; la traçabilité	Documents versionnés avec responsables et dates de revue, ADR, correspondance entre exigences et contrôles	Des croquis informels sans date, ni responsable, ni version

Deux habitudes aident avec tous les publics. **Commencez par la réponse.** Les lecteurs pressés doivent connaître la conclusion dès la première phrase, puis les raisons qui l'étayent, puis le détail ; c'est la structure que Barbara Minto enseignait aux consultants sous le nom de Pyramid Principle (principe de la pyramide), et c'est la même idée que la pyramide inversée de Fowler pour les ADR. **Employez leurs mots.** Pour une directrice financière, «cohérence à terme» n'est que du bruit ; «le restaurant voit une nouvelle commande jusqu'à deux secondes après le client, et rien n'est perdu» est un fait qu'elle peut évaluer.

Expliquer les compromis

L'essentiel de la communication architecturale consiste, au fond, à expliquer un compromis. Les non-spécialistes se méfient des recommandations qui semblent n'avoir aucun inconvénient, et ils ont raison. Une structure claire, employée de façon constante, rend les compromis faciles à suivre : les **options** (y compris ne rien faire), les **critères** qui comptent pour ce public, la façon dont chaque option s'en sort, la **recommandation** et ses **conséquences**, y compris ce à quoi vous renoncez. C'est l'anatomie d'une décision vue au chapitre 1 et le contenu d'un ADR, remodelés pour un auditoire.

EXEMPLE COMMENTÉ

Exemple commenté : expliquer un compromis à l'équipe de direction

La plateforme de livraison de repas a subi deux pannes un vendredi soir, causées par un service de notification lent. L'architecte dispose de cinq minutes en réunion de direction pour faire approuver l'ADR-0012 (événements de commande asynchrones).

Commencer par la réponse	« Je recommande de changer la façon dont les commandes circulent entre nos systèmes, afin qu'une défaillance des notifications ou de l'affectation des livreurs ne puisse plus empêcher les clients de commander. Cela coûte environ six semaines de travail d'une équipe et quelques centaines de livres par mois. »
Pourquoi maintenant	« Le vendredi 21 août, une panne des notifications a bloqué toutes les commandes pendant 23 minutes, en plein pic du vendredi. À environ 40 commandes par seconde, cela représente à peu près 55 000 commandes qui n'ont pas pu être passées, suivies de remboursements et de réclamations. »
Les options, dans leurs termes	« Nous pourrions (A) garder la conception actuelle et ajouter des limites de sécurité, ce qui est peu coûteux mais laisse encore un service lent ralentir chaque commande ; (B) adopter le changement recommandé ; ou (C) laisser d'autres systèmes lire directement nos données de commande, ce qui est rapide mais rend chaque modification future plus risquée. »
Ce à quoi nous renonçons	« Les restaurants verront les nouvelles commandes une ou deux secondes après la confirmation reçue par le client, et nous aurons un service managé de plus à exploiter. Nous jugeons ces deux points acceptables ; l'exploitation est d'accord. »
La décision attendue	« J'ai besoin de votre accord pour consacrer ce trimestre six semaines de travail d'une équipe, ce qui retarde de deux semaines la fonctionnalité de fidélité. »

Aucune technologie n'est nommée : « courtier de messages » et « outbox transactionnelle » restent dans l'ADR. Le chiffre des commandes (40 commandes par seconde pendant 23 minutes) est vérifié sur des données réelles avant la réunion.

Le coût de la recommandation et les alternatives sont exposés clairement, afin que les dirigeants puissent être en désaccord sur le fond plutôt que sur la confiance.

Le résumé d'architecture en une page

Tout système gagne à avoir un résumé d'architecture en une page : une page unique qu'une nouvelle recrue, un dirigeant ou un relecteur d'une autre équipe peut lire en cinq minutes. Il ne remplace pas la documentation complète, il en est la porte d'entrée. Une structure utile est la

suivante : ce que fait le système et pour qui (deux phrases) ; les trois principaux objectifs de qualité ; un diagramme de contexte système ; un diagramme de conteneurs ou une liste des principaux blocs de construction ; les cinq décisions environ qui façonnent tout le reste (une ligne chacune, avec un lien vers les ADR) ; les principaux risques ; et où en savoir plus. La page arc42 allégée de arc42 et modèles est un tel résumé.

L'exercice qui consiste à faire tenir l'architecture sur une page est précieux en soi. Si les idées principales ne peuvent pas être exposées en une page, elles ne sont probablement pas claires non plus pour l'équipe.

Raconter : suivre une requête

On retient mieux les histoires que les structures. Une technique efficace pour expliquer n'importe quel système consiste à suivre une requête : choisissez un seul événement, concret et important, comme un client qui passe commande, et parcourez pas à pas ce qui lui arrive à travers l'architecture. Un diagramme statique de boîtes montre ce qui existe ; le récit montre pourquoi cela existe et comment les parties coopèrent. Il fait aussi apparaître naturellement les moments intéressants : où le client attend, où les données franchissent une frontière de confiance, ce qui se passe si le prestataire de paiement est lent.

Choisissez le récit en fonction du public. Pour les développeurs, le chemin nominal, puis une défaillance («maintenant, le prestataire de paiement dépasse son délai»). Pour l'exploitation, la défaillance et la reprise. Pour les dirigeants, l'expérience du client, avec seulement les moments qui comptent pour l'activité. Numérotez les étapes sur le diagramme pour pouvoir les montrer du doigt, et gardez le même diagramme à l'écran pendant que vous parlez plutôt que de passer d'une diapositive à l'autre.

Suivre une commande à travers la plateforme



Flèches pleines : appels synchrones. Pointillés : événements asynchrones. Numéros : ordre du récit.

Un diagramme de récit numéroté : les étapes 1 à 4 se déroulent pendant que le client attend ; les étapes 5 à 8 sont asynchrones et ne peuvent pas bloquer la commande.

Bien utiliser le tableau blanc et le croquis

Une grande partie de la communication architecturale se fait en direct, devant un tableau blanc ou sur un canevas partagé en ligne, et quelques habitudes rendent les croquis bien plus utiles. **Dites ce que vous dessinez avant de le dessiner** («ceci est le niveau conteneurs : les éléments que nous déployons séparément»), afin que le public connaisse le niveau d'abstraction. **Commencez par le contexte** : les utilisateurs et les systèmes externes sur les bords, puis l'intérieur. **Libellez chaque boîte et chaque ligne** au fur et à mesure, y compris le sens et ce qui circule ; une flèche sans libellé sera lue de trois façons différentes par trois personnes. **Dessinez une petite légende** dans un coin quand vous utilisez plus d'un type de ligne ou de boîte. **Laissez de la place**, car les meilleures idées arrivent en pleine discussion et ont besoin d'un endroit où aller.

Les croquis sont des outils de réflexion, faits pour être jetés, mais pas les conclusions. Photographiez le tableau et, dans la journée, transformez tout ce qui comptera en vrai diagramme, en ADR ou en modification de la documentation. Simon Brown, créateur du modèle C4, a observé que les diagrammes au tableau blanc de nombreuses équipes sont des boîtes et des traits ambigus précisément parce qu'il leur manque un ensemble d'abstractions convenu ; utiliser les niveaux C4 au tableau blanc est l'un des moyens les plus simples d'y remédier.

Documents de conception et RFC

De nombreuses organisations d'ingénierie utilisent un document de conception écrit, souvent appelé RFC (request for comments, appel à commentaires), pour proposer une modification importante avant de la construire. Le nom fait écho à la série des RFC d'Internet, qui a commencé avec la RFC 1 de Steve Crocker en avril 1969. L'auteur expose le problème, le contexte, les objectifs et les non-objectifs, la conception proposée, les alternatives envisagées et les préoccupations transversales comme la sécurité, la confidentialité et l'observabilité ; le document est diffusé aux personnes concernées, qui le commentent par écrit ; puis il est approuvé ou révisé. La description très lue des design docs chez Google par Malte Ubl (2020) énumère exactement ces sections et suggère dix à vingt pages pour les grands projets, et une à trois pages pour les petits «mini design docs». Gergely Orosz a décrit comment Uber a utilisé un processus de RFC dès ses premières années pour diffuser les connaissances et détecter les problèmes tôt (2018).

Amazon pousse l'écrit plus loin : sa lettre aux actionnaires de 2017 explique que l'entreprise n'utilise pas de présentations à base de diapositives en réunion, mais des «mémos de six pages à structure narrative», que les participants lisent en silence au début. L'avantage commun à toutes ces pratiques est qu'écrire oblige à penser clairement, que les commentaires écrits permettent aux collègues plus discrets ou à distance de contribuer, et que le document devient une trace. Les risques sont eux aussi communs : de longs documents que personne ne finit, des processus de relecture qui deviennent des barrières bureaucratiques, et des documents de conception jamais mis à jour après la mise en œuvre.

Documents de conception et ADR se complètent. Un document de conception explore un problème et propose une conception complète, couvrant souvent plusieurs décisions ; une fois qu'il est approuvé, les décisions importantes qu'il contient sont consignées sous forme d'ADR, plus courts, permanents et indexés. Une règle raisonnable : le document de conception pour réfléchir et convaincre, l'ADR pour se souvenir.

Présenter lors d'une revue d'architecture

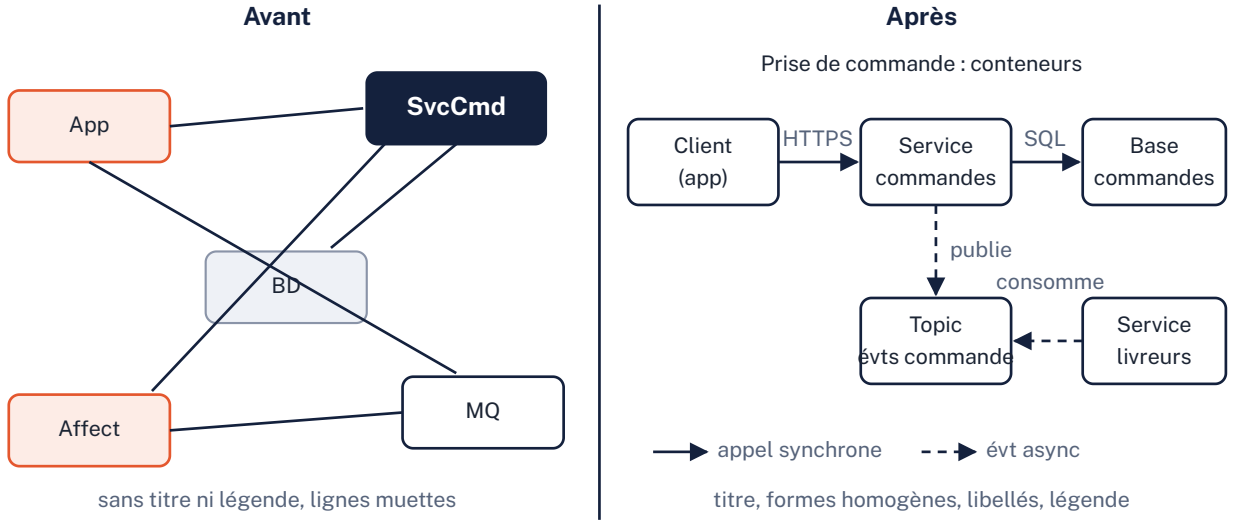
Une revue d'architecture est une réunion au cours de laquelle d'autres personnes expérimentées examinent une conception pour en trouver les risques avant qu'ils ne deviennent coûteux. Les méthodes structurées comme l'ATAM et les revues plus légères sont traitées au chapitre 11 : revues légères ; il s'agit ici de bien présenter. Envoyez le matériel au moins deux jours à l'avance : le résumé en une page, les diagrammes clés, les ADR pertinents et les scénarios de qualité. Dites dès le début ce que vous attendez de la revue : une approbation, une liste de risques ou un choix entre deux options. Présentez les facteurs avant la conception, car un relecteur ne peut pas juger une conception sans savoir quelles qualités elle doit atteindre. Déroulez un ou deux scénarios avec la technique qui consiste à suivre une requête, y

compris une défaillance. Montrez les compromis que vous avez faits et les risques que vous connaissez ; un relecteur qui trouve un risque que vous avez déjà listé vous fait davantage confiance, pas moins. Consignez chaque point soulevé, avec un responsable, et envoyez la liste ensuite.

Règles de conception visuelle des diagrammes

Les diagrammes communiquent avant que quiconque ait lu un mot, leur conception visuelle compte donc. Les règles ci-dessous s'inspirent des recommandations de notation C4 présentées dans le modèle C4 et de la conception de l'information en général. Le diagramme avant/après les applique aux mêmes cinq éléments.

Le même diagramme, avant et après



À gauche, la couleur et les abréviations ne veulent rien dire et les lignes se croisent sans libellé ; à droite, un titre, des formes cohérentes, un flux de gauche à droite, des flèches libellées et une légende.

Règles pour des diagrammes d'architecture clairs

1. **Donnez un titre à chaque diagramme**
   Indiquez le type et le périmètre, par exemple «Conteneurs : plateforme de livraison de repas», pour que le lecteur connaisse le niveau d'abstraction.

2. **Utilisez une seule notation, de façon cohérente**
   La même forme pour le même type d'élément dans tous les diagrammes ; ne mélangez pas les symboles C4 et UML sur un même diagramme sans le dire.

3. **Ajoutez une légende**
   Expliquez chaque forme, couleur, style de ligne et pointe de flèche qui porte un sens. Si vous ne pouvez pas l'expliquer, supprimez-le.

4. **Limitez les couleurs et donnez-leur un sens**
   Deux ou trois couleurs, chacune avec une signification (externe, nouveau, à risque). Ne comptez jamais sur la couleur seule : les diagrammes sont imprimés en noir et blanc et de nombreux lecteurs ont un trouble de la vision des couleurs.

5. **Orientez le flux de gauche à droite ou de haut en bas**
   Suivez le sens de lecture de votre public, avec des requêtes qui circulent dans un seul sens, pour que l'œil suive le récit.

6. **Libellez chaque élément et chaque flèche**
   Des noms sans abréviations inexpliquées ; des flèches à sens unique, avec ce qui circule et, entre conteneurs, le protocole.

7. **Évitez autant que possible les croisements de lignes**
   Réorganisez d'abord les boîtes ; s'il reste des croisements, divisez le diagramme. Un diagramme avec de nombreux croisements essaie généralement d'en montrer trop.

8. **Tenez sur un seul écran**
   Une à deux douzaines d'éléments au maximum, à peu près. Les vues plus larges relèvent d'un niveau inférieur ou de plusieurs diagrammes.

SI VOUS DÉBUTEZ

Pas à pas : décider quand tout le monde n'est pas d'accord

L'idée qui demande le plus d'attention ici est le consensus approximatif (rough consensus) : une décision peut être prise dès que chaque objection a été équitablement entendue et a reçu une réponse argumentée, même si certains préfèrent encore une autre option. Imaginez le déroulement d'une proposition.

- L'équipe Commandes ouvre l'ADR-0012, qui publie les événements de commande via un courtier de messages, comme enregistrement proposé dans une pull request. Les équipes restaurant, livreurs et notifications, ainsi que les SRE, sont invités à commenter.
- Deux objections arrivent. Un ingénieur de l'équipe notifications préfère garder les appels synchrones avec délais d'expiration et nouvelles tentatives, l'option qui demande le moins de nouvelle infrastructure. Une ingénieure de l'équipe restaurant craint que les tablettes affichent les nouvelles commandes en retard.
- Chaque objection reçoit une réponse écrite. L'enregistrement explique qu'avec des appels synchrones, un service lent retarde toujours chaque commande, et reconnaît qu'une tablette de restaurant peut avoir une ou deux secondes de retard sur la confirmation reçue par le client : un coût qu'il consigne.
- L'ingénieur de l'équipe notifications n'est toujours pas convaincu. Son objection a été équitablement entendue et a reçu une réponse, mais elle n'est pas prise en compte : la conception ne change pas pour lui donner satisfaction. Pour un consensus approximatif, cela suffit.
- Avant de trancher, les décideurs demandent : « Y a-t-il quelqu'un qui ne peut pas vivre avec cela, et pourquoi ? » Aucun nouveau point bloquant n'apparaît. Si une objection était restée sans réponse, il n'y aurait pas encore de consensus approximatif, quel que soit le nombre de personnes favorables.
- L'enregistrement est accepté, avec le désaccord de l'ingénieur et ses raisons consignés dedans. L'ingénieur s'engage à faire fonctionner la solution ; si l'avenir lui donne raison, l'enregistrement indique ce qu'il faut réexaminer.
- **La leçon :** visez à répondre à chaque objection, pas à mettre tout le monde d'accord. Consignez le désaccord avec la décision, et convenez à l'avance de qui décide si le consensus ne peut pas être atteint.

Gérer les désaccords et construire un consensus

Les désaccords sur l'architecture sont normaux et sains : ils signifient généralement que les gens voient des risques différents. Le but n'est pas de gagner, mais d'aboutir à une décision que les personnes qui devront vivre avec comprennent et soutiendront. Plusieurs pratiques aident.

Distinguez les intérêts des positions. «Nous devons utiliser Kafka» est une position ; «je crains que nous perdions des événements pendant les déploiements» est un intérêt, et un intérêt peut souvent être satisfait de plusieurs façons. **Convenez des critères avant de comparer les options**, comme dans une matrice de décision pondérée ; la plupart des discussions animées sont en réalité des désaccords sur ce qui compte, et elles s'apaisent dès que c'est explicite. **Levez l'incertitude par des preuves** : un spike (investigation limitée dans le temps) de deux jours ou un test de charge tranche bien des débats que les réunions ne peuvent pas régler. **Mettez-le par écrit** : un ADR proposé ou un document de conception permet à chacun de commenter à son rythme et donne la parole aux collègues plus discrets.

Visez le consentement plutôt que l'unanimité. La RFC 7282 de l'IETF (Pete Resnick, 2014) décrit le consensus approximatif : une décision est atteinte quand chaque objection a été traitée, c'est-à-dire examinée et a reçu une réponse, sans forcément avoir été prise en compte. Demander «y a-t-il quelqu'un qui ne peut pas vivre avec cela, et pourquoi ?» fait apparaître les points bloquants plus vite que de demander si tout le monde est d'accord. Quand le consensus n'est pas possible, quelqu'un doit décider, et il faut savoir à l'avance qui. La lettre aux actionnaires de Jeff Bezos de 2016 a popularisé le *disagree and commit* (exprimer son désaccord, puis s'engager) : chacun exprime ouvertement son désaccord, puis s'engage pleinement dans la décision une fois qu'elle est prise. Consignez le désaccord et ses raisons dans l'ADR ; si ceux qui n'étaient pas d'accord ont raison, l'enregistrement indique ce qu'il faut réexaminer et pourquoi.

Deux dérives méritent une mise en garde. L'architecte qui gagne chaque discussion par ancienneté n'entend plus parler des problèmes. L'équipe qui ne peut pas décider sans unanimité laisse des décisions importantes se prendre par défaut, ce qui est l'inverse du dernier moment responsable.

Liste de contrôle : une présentation d'architecture

Utilisez-la avant toute présentation d'architecture, du point de cinq minutes avec la direction jusqu'à la revue formelle.

- [ ] **Public et résultat attendu nommés**
  Vous pouvez dire en une phrase qui sera dans la salle et ce que ces personnes devront décider ou faire ensuite.
- [ ] **Commencer par la réponse**
  La recommandation ou le message principal figure sur la première diapositive ou dans le premier paragraphe.
- [ ] **Les facteurs avant la conception**
  Les objectifs métier, les objectifs de qualité et les contraintes sont présentés avant la solution.
- [ ] **Niveau d'abstraction adapté au public**
  Niveau contexte pour les dirigeants, niveaux conteneurs et composants pour les ingénieurs ; pas plus de deux niveaux par séance.
- [ ] **Une histoire racontée de bout en bout**
  Un parcours qui suit une requête, avec une défaillance pour les publics techniques.
- [ ] **Compromis et risques montrés honnêtement**
  Les options, ce que coûte chacune et ce à quoi vous renoncez ; les risques connus listés avant que quelqu'un d'autre ne les trouve.
- [ ] **Chaque diagramme a un titre, une légende et des libellés**
  Une notation cohérente, peu de couleurs mais porteuses de sens, lisible depuis le fond de la salle.
- [ ] **Matériel envoyé à l'avance**
  Pour les revues et les décisions, le résumé et les ADR partent au moins deux jours avant, afin que la réunion serve à la discussion.
- [ ] **La demande est explicite**
  Une approbation, des risques, un choix entre options ou simplement une information ; dites lequel, et pour quand.
- [ ] **Suites consignées**
  Questions, objections et actions consignées avec des responsables ; décisions transformées en ADR ; documentation mise à jour.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **À quelle question devez-vous répondre avant de préparer toute explication d'architecture ?**

Qui est le public, et ce qu'il doit décider ou faire différemment ensuite ; le contenu, le niveau d'abstraction et le vocabulaire en découlent.

▼ **Pourquoi raconter un récit qui suit une requête plutôt que de montrer seulement un diagramme de boîtes ?**

Un diagramme de boîtes montre ce qui existe ; le récit montre comment les parties coopèrent et pourquoi, et fait apparaître naturellement les attentes, les frontières de confiance et les points de défaillance qu'une image statique cache.

▼ **En quoi un document de conception et un ADR diffèrent-ils dans leur objectif ?**

Un document de conception explore un problème et propose une conception complète, pour commentaires, avant la construction ; un ADR est un enregistrement court et permanent d'une seule décision. Un document de conception approuvé produit généralement plusieurs ADR.

▼ **Que signifie le consensus approximatif dans la RFC 7282 ?**

Chaque objection a été examinée et a reçu une réponse, sans forcément avoir été prise en compte ; ce n'est ni l'unanimité ni un vote à la majorité.

▼ **À vous : Un architecte dit à l'équipe de direction : « Il nous faut un transactional outbox et un courtier pour découpler notre chemin de commande synchrone. » Reformulez cela pour elle en trois phrases environ, avec la recommandation, le coût et ce qu'elle doit décider.**

« Je recommande de changer la façon dont les commandes circulent entre nos systèmes, afin qu'une panne des notifications ne puisse plus empêcher les clients de commander, comme cela s'est produit deux fois récemment. Il faudrait environ six semaines-équipe et un petit coût d'exploitation mensuel ; la contrepartie est que certaines mises à jour arriveraient quelques secondes plus tard. Merci de l'approuver aujourd'hui. » La réponse vient en premier, en termes métier, avec les coûts et l'inconvénient clairement énoncés et sans jargon.

SOURCES

- C4 model: notation: https://c4model.com/diagrams/notation

- Design Docs at Google, Malte Ubl (2020): https://www.industrialempathy.com/posts/design-docs-at-google/
- Scaling Engineering Teams via RFCs: Writing Things Down, Gergely Orosz: https://blog.pragmaticengineer.com/scaling-engineering-teams-via-writing-things-down-rfcs/
- RFC 7282: On Consensus and Humming in the IETF: https://www.rfc-editor.org/rfc/rfc7282.html
- Amazon 2017 letter to shareholders: https://www.aboutamazon.com/news/company-news/2017-letter-to-shareholders

Lectures complémentaires

- *Documenting Software Architectures: Views and Beyond, 2nd edition*, Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord and Judith Stafford (2010), Addison-Wesley. Le traitement le plus complet de la documentation d'architecture : les vues, les styles, les sept règles d'une bonne documentation et un modèle de documentation.
  https://www.sei.cmu.edu/library/documenting-software-architectures-views-and-beyond-second-edition/
- *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, 2nd edition*, Nick Rozanski and Eóin Woods (2012), Addison-Wesley. Un catalogue pratique de sept points de vue et de perspectives comme la sécurité et les performances, destiné aux grands systèmes d'information.
  https://www.viewpoints-and-perspectives.info/
- *Architectural Blueprints: The "4+1" View Model of Software Architecture*, Philippe Kruchten (1995), IEEE Software 12(6). L'article court qui a popularisé la description d'architecture multivue et la validation par scénarios.
  https://arxiv.org/abs/2006.04975
- *ISO/IEC/IEEE 42010:2022 Software, systems and enterprise: Architecture description*, ISO/IEC JTC 1/SC 7 and IEEE (2022), ISO. La norme internationale qui définit les parties prenantes, les préoccupations, les points de vue, les vues et les types de modèles.
  https://www.iso.org/standard/74393.html
- *The C4 Model: Visualizing Software Architecture*, Simon Brown (2026), O'Reilly Media. La présentation par son créateur des abstractions, des diagrammes et de la notation du C4 ; le site web en couvre gratuitement l'essentiel.
  https://c4model.com/
- *UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd edition*, Martin Fowler (2003), Addison-Wesley. Un guide court et pratique des parties d'UML qui valent la peine d'être connues, et la source de la distinction entre croquis, plan détaillé et langage de programmation.
- *Living Documentation: Continuous Knowledge Sharing by Design*, Cyrille Martraire (2019), Addison-Wesley. Comment tirer la documentation du système lui-même et limiter le contenu écrit à la main à ce que seules les personnes peuvent apporter.

Glossaire

arc42

Modèle gratuit en douze sections pour documenter et communiquer une architecture logicielle, créé par Gernot Starke et Peter Hruschka.

Bloc de construction (building block)
Dans arc42, toute partie de la structure statique d'un système, comme un sous-système, un service, un composant ou un module, représentée dans la vue des blocs de construction.

Cadre de description d'architecture
Dans ISO/IEC/IEEE 42010:2022, ensemble établi de conventions pour produire des descriptions d'architecture, comme des points de vue, des modèles et des règles.

Composant (C4)
Dans le modèle C4, groupe de fonctionnalités liées derrière une interface claire, s'exécutant à l'intérieur d'un conteneur et déployé avec lui.

Composant de vue
Dans l'ISO/IEC/IEEE 42010:2022, une partie d'une vue, comme un diagramme, un tableau ou un modèle, régie par un type de modèle. L'édition de 2011 l'appelait modèle d'architecture.

Concept transversal
Règle ou solution qui s'applique à de nombreuses parties d'un système, comme la sécurité, la gestion des erreurs, la journalisation ou la persistance.

Consensus approximatif (rough consensus)
Décision obtenue lorsque chaque objection a été examinée et a reçu une réponse, sans nécessairement avoir été prise en compte ; décrite dans la RFC 7282 de l'IETF.

Conteneur (C4)
Dans le modèle C4, unité exécutable ou déployable séparément qui exécute du code ou stocke des données, comme une application web, une app mobile, un service ou une base de données. Ce n'est pas un conteneur Docker.

Correspondance
Relation consignée entre des éléments d'une description d'architecture, par exemple entre un conteneur dans une vue et le serveur sur lequel il s'exécute dans une autre.

Description de la conception logicielle (SDD)
Document décrivant en détail une conception logicielle, dont le contenu est spécifié par la norme IEEE Std 1016-2009 ; courant dans les projets régis par un contrat.

Diagramme d'activité
Diagramme UML qui représente un flux de travail sous forme d'actions, de décisions, de bifurcations et de jonctions parallèles, souvent divisé en couloirs (swimlanes) indiquant qui fait quoi.

Diagramme d'états-transitions
Diagramme UML qui montre les états dans lesquels un objet peut se trouver et les transitions entre eux, chacune déclenchée par un événement.

Diagramme de conteneurs

Diagramme C4 montrant les applications, les services et les stockages de données à l'intérieur d'un système, avec leurs technologies et la manière dont ils communiquent.

Diagramme de contexte système

Diagramme C4 qui montre un système sous la forme d'une seule boîte, les personnes qui l'utilisent et les autres systèmes avec lesquels il interagit.

Diagramme de déploiement

Diagramme (en C4 ou en UML) montrant comment le logiciel est mis en correspondance avec l'infrastructure dans un environnement, y compris les nœuds, les réplicas et les chemins réseau.

Diagramme de séquence

Diagramme UML qui montre les messages échangés entre des participants au fil du temps pour un scénario, le temps s'écoulant du haut vers le bas de la page.

Diagramme dynamique

Diagramme C4 complémentaire montrant comment les éléments collaborent à l'exécution dans un scénario donné, généralement sous forme d'interactions numérotées.

Diagrammes en tant que code (diagrams as code)

Écriture de diagrammes sous forme de texte dans un petit langage, comme PlantUML, Mermaid, D2 ou Structurizr DSL, qu'un outil met en page et dessine.

Document de conception (RFC)

Proposition écrite pour une modification importante, couvrant le problème, les objectifs, la conception et les alternatives, diffusée pour commentaires avant que la modification ne soit construite.

Documentation en tant que code (docs as code)

Production de la documentation avec les mêmes outils et le même flux de travail que le logiciel : fichiers en texte brut dans la gestion de versions, revue de code, vérifications automatisées et publication.

Documentation juste suffisante

Documentation dimensionnée selon ce dont ses lecteurs ont réellement besoin, de sorte qu'en écrire et en maintenir davantage coûterait plus que cela ne rapporte.

Documentation vivante

Documentation qui évolue avec le système qu'elle décrit, souvent parce qu'elle est générée à partir du système lui-même ou vérifiée par rapport à lui.

Entité d'intérêt

Dans ISO/IEC/IEEE 42010:2022, ce dont on décrit l'architecture : un système logiciel, un système de systèmes, une ligne de produits ou une entreprise.

Facteur de décision
Force ou critère, comme un objectif de qualité ou une contrainte, auquel une décision d'architecture doit satisfaire et à l'aune duquel les options sont comparées.

Garde (guard)
Condition entre crochets sur une transition ou un flux, qui doit être vraie pour que celle-ci soit empruntée, comme [le code correspond].

Interface fournie
Dans un diagramme de composants UML, une interface qu'un composant offre aux autres, représentée par un petit cercle (une boule) au bout d'une tige.

Interface requise
Dans un diagramme de composants UML, une interface dont un composant a besoin qu'un autre composant la fournisse, représentée par un demi-cercle (une prise).

Journal des décisions (decision log)
Collection ordonnée des ADR d'un système, généralement accompagnée d'un index indiquant le numéro, le titre, le statut et la date de chaque enregistrement.

Langage de modélisation unifié (UML)
Langage de modélisation graphique généraliste normalisé par l'Object Management Group. La version actuelle est la 2.5.1 (2017).

Ligne de vie
Dans un diagramme de séquence, la ligne verticale en pointillés sous un participant, le long de laquelle ses messages et ses activations sont montrés dans l'ordre chronologique.

MADR
Markdown Architectural Decision Records : modèle d'ADR très utilisé, comprenant les facteurs de décision, les options envisagées, le résultat, les conséquences et la confirmation.

Modèle C4
Approche de Simon Brown pour les diagrammes d'architecture : une hiérarchie de diagrammes de contexte système, de conteneurs, de composants et de code, plus quelques diagrammes complémentaires.

Modèle de domaine
Petit diagramme ou petite description des concepts métier clés d'un système, de leurs attributs et de leurs relations, partagé par les développeurs et les experts du domaine.

Modèle de vues 4+1
Approche de Philippe Kruchten (1995) qui décrit un système à travers des vues logique, de processus, de développement et physique, reliées entre elles par un petit ensemble de scénarios.

Multiplicité
Dans un diagramme de classes UML, le nombre d'objets qui peuvent participer à chaque extrémité d'une association, par exemple 1, 0..1, 0..* ou 1..*.

Outil de dessin
Outil de création de diagrammes qui ne connaît que des formes, des traits et du texte, de sorte qu'un même élément dessiné dans deux diagrammes n'a aucun lien d'un diagramme à l'autre.

Outil de modélisation
Outil qui stocke des éléments et des relations dans un modèle et génère des diagrammes comme autant de vues de ce modèle, en assurant la cohérence de toutes les vues.

Perspective
Dans l'approche de Rozanski et Woods, une préoccupation de qualité, comme la sécurité ou les performances, appliquée à l'ensemble des vues plutôt que représentée comme une vue à part.

Processus de consultation (advice process)
Mode de prise de décision décentralisé dans lequel chacun peut décider, à condition d'avoir d'abord demandé l'avis des personnes concernées et des experts compétents.

Résumé d'architecture en une page
Une page unique qui présente l'objectif d'un système, ses objectifs de qualité, son contexte, ses principaux blocs de construction, ses décisions clés et ses risques, comme porte d'entrée vers une documentation plus complète.

Structurizr
Les outils de Simon Brown pour le modèle C4, dans lesquels un modèle textuel génère de nombreux diagrammes cohérents.

Suivre une requête
Explication d'un système en suivant un seul événement concret, comme une commande, étape par étape à travers chaque partie qu'il touche.

Système logiciel (C4)
Dans le modèle C4, le niveau d'abstraction le plus élevé : quelque chose qui apporte de la valeur à ses utilisateurs, généralement détenu par une seule équipe ou organisation.

Type de modèle (model kind)
Dans l'ISO/IEC/IEEE 42010, les conventions propres à un type de modèle, comme un diagramme de séquence UML ou un diagramme de conteneurs C4.

Vue composants et connecteurs
Vue de ce qui existe à l'exécution, comme les services, les clients, les bases de données et les files, et des connecteurs (appels, messages, données partagées) qui les relient.

Vue d’allocation

Vue qui met en correspondance le logiciel avec des éléments qui lui sont extérieurs : l’infrastructure (déploiement), les fichiers et les builds (implémentation) ou les équipes (affectation du travail).

Vue des modules

Vue des unités d’implémentation (paquets, bibliothèques, couches) et de leurs dépendances, telles qu’elles existent au moment du développement.

Y-statement (déclaration en Y)

Enregistrement de décision en une phrase, proposé par Olaf Zimmermann : dans le contexte de…, face à…, nous avons choisi… et écarté…, pour obtenir…, en acceptant que…