

LE PLAYBOOK

# The AI Product Builder's Playbook

L'IA n'a pas supprimé le process. Elle l'a  
rendu obligatoire.

Benoit Lecureur

[benoitlecureur.com](https://benoitlecureur.com)

# Sommaire

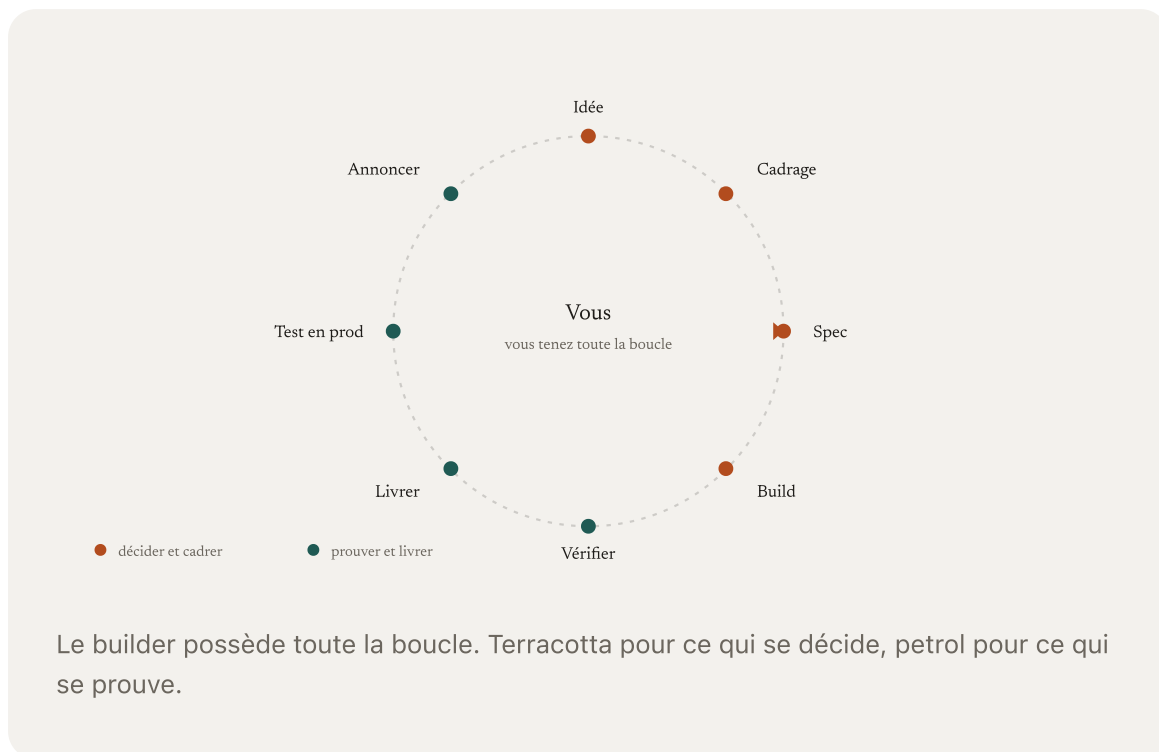
|    |                                                                                 |    |
|----|---------------------------------------------------------------------------------|----|
| 00 | Le rôle : l'AI Product Builder                                                  | 3  |
| 01 | Les lois du build                                                               | 5  |
| 02 | Les fondations dev, en langage clair                                            | 10 |
| 03 | CI/CD et contrôles mécaniques                                                   | 14 |
| 04 | Les cycles de travail                                                           | 16 |
| 05 | Specs et cadrage, le cœur du métier                                             | 19 |
| 06 | Tests et QA : la preuve contre l'intention                                      | 23 |
| 07 | Données, migrations et argent                                                   | 26 |
| 08 | Livrer, sécuriser, opérer                                                       | 28 |
| 09 | La documentation comme infrastructure                                           | 33 |
| 10 | Design system, contenu, cohérence                                               | 35 |
| 11 | Construire avec l'IA : ce qui change vraiment                                   | 37 |
| 12 | Le catalogue des pièges                                                         | 41 |
| 13 | Aller plus loin : ce que je n'ai pas encore fait, et à quoi ressemble le dev+++ | 42 |
| 14 | Glossaire                                                                       | 45 |
| 15 | L'auto-évaluation                                                               | 51 |
| 16 | Notes de cas (anonymisées)                                                      | 53 |

# Le rôle : l'AI Product Builder

## Ce que c'est

L'AI Product Builder tient la boucle complète : de l'idée au cadrage, du cadrage à la spec, de la spec au build (exécuté par l'IA), à la vérification, au déploiement, aux tests et à la QA, jusqu'à l'annonce de la mise en ligne.

C'était quatre métiers : un PM qui spécifie, des développeurs qui codent, une QA qui teste, des ops qui déploient. Une seule personne peut désormais tout porter, parce que l'IA lui donne le levier pour exécuter ce qu'elle n'aurait pas pu coder seule.



## Le basculement

Vous n'avez pas besoin d'être développeur de carrière. Vous avez besoin de trois choses.

1. Comprendre les concepts (c'est à ça que sert ce livre), pour parler la langue, faire les bons choix, et sentir quand quelque chose cloche.

2. Orchestrer l'IA : la cadrer, lui donner le contexte, découper le travail, relire ce qu'elle produit.
3. Tenir la ligne de qualité : les décisions produites et les arbitrages restent les vôtres, la vérification n'est pas négociable, vous êtes le garde-fou.

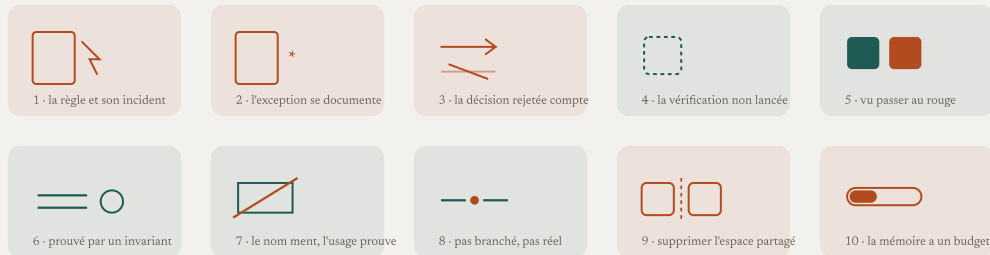
Une phrase à retenir : les décisions sont à vous, l'exécution est déléguée, la vérification est obligatoire.

## **Ce que le rôle réunit**

Le cadrage (transformer une envie floue en périmètre clair, découpé, priorisé). La spec (écrire le document qui fixe l'objectif, le contexte, les décisions, le modèle de données, les tranches, les tests). Le build (faire exécuter l'IA à partir de prompts cadrés). La vérification (relire, builder, tester dans un vrai navigateur, valider chaque étape). Le déploiement (mettre en ligne, et savoir ce que chaque geste déclenche). Les tests et la QA (dérouler les checklists, valider les parcours de bout en bout, traiter les cas limites). La communication (présenter la version : ce qui est livré, pourquoi, une démo, la suite).

Vous n'êtes pas une version réduite d'une équipe. Vous en êtes le chef d'orchestre.

# Les lois du build



Un pictogramme par loi — les dix, d'un coup d'œil.

Ce sont les principes qui survivent même si l'on retire de l'histoire tout projet particulier. Ils sont la colonne vertébrale de tout ce qui suit. Chacun a été appris à la dure.

## Loi 1. Une règle s'écrit avec son incident

Une règle sans histoire se fait contourner. Une règle qui porte la date, le coût et l'échec qui l'a créée, non, parce que le lecteur voit exactement ce qu'il achète en l'ignorant.

Alors quand vous écrivez une règle, écrivez l'échec à côté. Et le test : si vous ne trouvez aucun échec derrière votre règle, c'est probablement une préférence, pas une règle. Les préférences, c'est très bien. Ne les défendez simplement pas comme des lois.

## Loi 2. Une exception se documente comme une règle

Le réflexe normal est de tolérer une exception en silence, ce qui est exactement la façon dont les exceptions se multiplient. Faites l'inverse. Une exception s'écrit avec quatre choses : la règle générale, redite et explicitement protégée ; les critères cumulatifs qui autorisent l'exception ; la liste nommée

de tous les cas qui l'utilisent aujourd'hui ; et la condition de retour à la règle générale.

« Si un seul critère tombe, ceci repasse sous la règle générale. » Cette ligne-là empêche une exception de devenir discrètement la nouvelle norme.

### **Loi 3. Une décision rejetée vaut autant qu'une décision retenue**

Gardez une « alternative rejetée » à côté de chaque vraie décision. C'est ce qui empêche quelqu'un, six mois plus tard, de rep proposer une option que vous aviez déjà écartée, avec les mêmes arguments, parce qu'il n'était pas dans la pièce la première fois.

Corollaire : un plan obsolète ne se supprime pas. Il se barre et se date, avec un renvoi vers la décision vivante. Le supprimer signifie simplement que vous le reconstruirez de zéro et redécouvrirez pourquoi il était mauvais.

### **Loi 4. Une vérification que vous n'avez pas lancée est pire que rien**

C'est l'anti-pattern le plus coûteux qui soit. Imaginez une note qui dit « vérifié : renvoie le bon résultat », présentée comme une preuve, alors que la vérification n'a jamais été exécutée. Non seulement elle ne vous protège pas, mais elle clôt le sujet, et donc plus personne ne regarde. Sur un projet, une « vérification » écrite mais jamais lancée a laissé un lot de fichiers internes exposés en public pendant des mois.

Donc la forme correcte d'un correctif n'est jamais « vérifier que ça marche ». C'est la commande exacte à lancer, et le critère de régression : le signal précis qui vous dirait que le bug est revenu.

### **Loi 5. Un feu vert ne vaut rien tant que vous ne l'avez pas vu passer au rouge**

Avant de faire confiance à un contrôle automatique sur un terrain neuf, faites-le échouer exprès. S'il ne passe pas au rouge quand vous cassez la

chose, il ne vous protégera jamais, il vous rassurera seulement.

Un contrôle automatique a de nombreuses façons de mentir en silence : il n'analyse pas la zone qui vous intéresse ; il ne voit que les fichiers déjà enregistrés dans le versionnage, et déclare donc « tout va bien » sur du code tout neuf ; son motif de recherche rate la vraie forme du code ; la cible se trouve hors de son périmètre ; l'exécuteur compte « rien à faire » comme un succès. Exemple réel du dernier piège : une suite de tests existait, plus de cent assertions, et elle était branchée pour ne tourner nulle part. Elle passait en ne s'exécutant jamais.

La discipline : calibrez chaque verrou sur un cas connu-bon et un cas connu-mauvais, figés en tests. Sans ça, rien ne distingue « fonctionne » de « inerte ». Et n'oubliez pas le versant humain : un verrou qui crie au loup finit désactivé.

## **Loi 6. Un changement de masse se prouve par un invariant, jamais par une relecture**

Quand vous changez mille choses d'un coup, vous ne pouvez pas vérifier à l'œil. Vous le prouvez par un invariant, un fait qui doit tenir. Trois qui fonctionnent en pratique.

L'égalité d'ensembles : après avoir remplacé quelque chose sur N pages, vérifiez que le nombre de pages portant la nouvelle version égale le nombre qui portait l'ancienne, et qu'il ne reste aucune ancienne. Une montée de version à moitié faite désynchronise tout tout en ayant l'air normale.

L'empreinte des lignes qui ne doivent pas bouger : avant et après une grosse migration, prouvez que les lignes précises auxquelles vous ne deviez pas toucher sont identiques octet pour octet. Vous n'évitez pas seulement d'y toucher, vous prouvez que vous n'y avez pas touché.

Le delta chiffré en production : pas « le test est vert », mais « ces nombres précis sont passés de X à Y sur les vraies données, et zéro élément s'est

retrouvé mal classé ». Un delta vaut mieux qu'une coche verte.

### **Loi 7. Le nom ment, la preuve est l'usage**

Les noms dérivent de leur sens. Sur une app, un objet appelé « Société » désignait en réalité un groupe parent, pas un site unique. Une relation construite sur la ressemblance des noms, plutôt que sur la réalité des données, rejetait près de huit lignes sur dix. Les noms étaient d'accord. Les données, non.

L'extension la plus difficile et la plus utile : cela vaut aussi pour vos propres documents. Un document que vous avez écrit vous-même n'est pas une source de vérité, c'est une hypothèse datée. Confrontez-le au réel avant de lui faire confiance. Deux corollaires : le code désactivé est souvent plus riche que le code vivant (il dit ce qui a été tenté), et un composant nommé dans un commentaire est un composant qui mérite d'être ouvert.

### **Loi 8. Ce qui n'est pas branché sur un contrôle mécanique n'existe pas**

Et son jumeau : ce qui n'est pas écrit dans un fichier n'existe pas. Une conversation n'est pas un système de mémoire.

La preuve est banale et universelle : sur cinq points soulevés à l'oral pendant une revue, quatre ont été écrits quelque part, un non. Celui qui s'est évaporé était une mention légale affichée aux utilisateurs au moment exact de leur consentement, et elle était devenue fausse en silence. Les accords verbaux s'évaporent. Si ça compte, ça vit dans un fichier ou dans un contrôle automatique, ou ça n'existe pas vraiment.

### **Loi 9. Le vrai correctif, c'est de supprimer l'espace partagé**

Quand deux personnes (ou deux sessions d'IA) travaillant en parallèle se percutent sans arrêt sur la même ressource partagée, l'instinct est d'ajouter

un meilleur contrôle. Souvent, le meilleur geste est de supprimer entièrement la chose partagée.

Le cas classique : des travaux en parallèle attrapaient chacun « le numéro suivant » pour un changement, sans se voir, et se percutaient encore et encore. Aucune vérification préalable ne pouvait marcher, parce que regarder « le dernier numéro utilisé » revient à regarder un monde où le changement voisin n'existe pas encore. Le correctif n'était pas un contrôle plus malin. C'était de passer à un horodatage, pour qu'il n'y ait plus de compteur partagé à consulter. Deux horodatages n'entrent pas en collision. Quand un verrou n'arrive pas à tenir une ressource partagée, demandez-vous si cette ressource doit être partagée du tout.

### **Loi 10. La mémoire a un budget, et elle tend vers le vide**

La mémoire de votre projet (le fichier qui conserve le contexte dont une IA ou un coéquipier a besoin) a un budget. Au-delà, on n'ajoute plus, on distille. Un bon fichier de mémoire est conçu pour rétrécir avec le temps, pas pour grossir : chaque entrée est distillée ou supprimée quand son chapitre se referme.

La version plus profonde : une leçon écrite six fois, à chaque fois comme si elle était neuve, ce n'est pas six leçons. C'est une loi qui n'a jamais eu de maison. Quand vous vous surprenez à réapprendre la même chose, n'ajoutez pas une septième note. Donnez une maison à la loi, et chaque occurrence future n'aura plus qu'à pointer dessus.

# Les fondations dev, en langage clair

Vous n'avez rien à écrire de tout ça. Vous devez le comprendre assez bien pour faire les bons choix et sentir quand ça sent mauvais. Chaque terme se retrouve aussi dans le glossaire.

## 2.1 Le versionnage, et pourquoi « pas poussé » veut dire « n'existe pas encore »

Le versionnage enregistre l'histoire de votre code : qui a changé quoi, quand, et vous permet de revenir en arrière. Sa maison en ligne est la copie de référence, la sauvegarde et le point de rendez-vous.

Les mots que vous utiliserez vraiment : un dépôt est le projet versionné. Un commit est un instantané daté d'un ensemble de changements, avec un message. Une branche est une ligne de travail parallèle ; la ligne principale s'appelle généralement « main » et c'est votre production. Pousser et tirer envoient vos commits vers le haut et récupèrent ceux des autres. Fusionner replie une branche dans une autre. Une pull request est une demande de fusion, généralement relue avant.

La leçon payée en temps : un commit qui n'est pas poussé ne vit que sur votre machine. Un nouveau clone ou un incident local l'efface. La règle : poussez après chaque unité de travail. Le local est fragile, le push est la sauvegarde. J'ai perdu du travail non poussé deux fois. Récupéré les deux fois, mais ce sont des heures qu'on ne rattrape pas.

### Pour le CTO en vous

quand plusieurs comptes partagent une machine, un push peut échouer sur un « repository not found » simplement parce que le mauvais compte est actif. Changez de compte avant de pousser.

## 2.2 Branches et revue

Le modèle classique : main est la production, sacrée, toujours déployable. Vous travaillez sur une branche, vous vérifiez, puis vous fusionnez dans main. Certaines équipes gardent une branche « préprod » de longue durée : on y pousse, on regarde l'URL de prévisualisation, puis on fusionne dans main. La revue de code est le moment où un pair relit avant la fusion. Avec l'IA, la revue devient deux choses à la fois : vous relisez ce que l'IA a produit, et l'IA fait sa propre passe de revue. Deux lecteurs valent mieux qu'un.

## 2.3 Environnements, configuration et secrets

Un produit vit dans plusieurs environnements : le local (votre machine, pour construire), la préprod ou staging (une copie en ligne pour valider avant la prod), et la production (ce que voient les vrais utilisateurs). Les variables d'environnement configurent le produit environnement par environnement, comme une adresse d'API ou une clé publique. Deux familles comptent. La configuration non sensible peut vivre dans le code. Les secrets (clés d'API, clés d'accès puissantes, secrets de paiement) ne vont jamais dans le versionnage. Ils se posent à la main dans le tableau de bord de votre hébergeur et survivent aux déploiements.

La règle d'or, à se tatouer : la clé puissante, celle qui a tous les droits sur votre base de données, n'est jamais dans le frontend. Uniquement côté serveur. Le frontend reçoit une clé publique, restreinte, encadrée par les règles de la base. Si une clé tout-accès atteint un jour le navigateur, considérez que la base est ouverte au monde entier.

Tenez une matrice des variables d'environnement : un simple tableau de chaque variable, où elle vit (dans le code, dans un coffre à secrets, dans le tableau de bord de l'hébergeur, dans votre fichier local), et qui la pose. Les variables manquantes se découvrent au moment du déploiement, dans la pire humeur, encore et encore, sauf si vous tenez la carte.

## 2.4 Les briques du backend

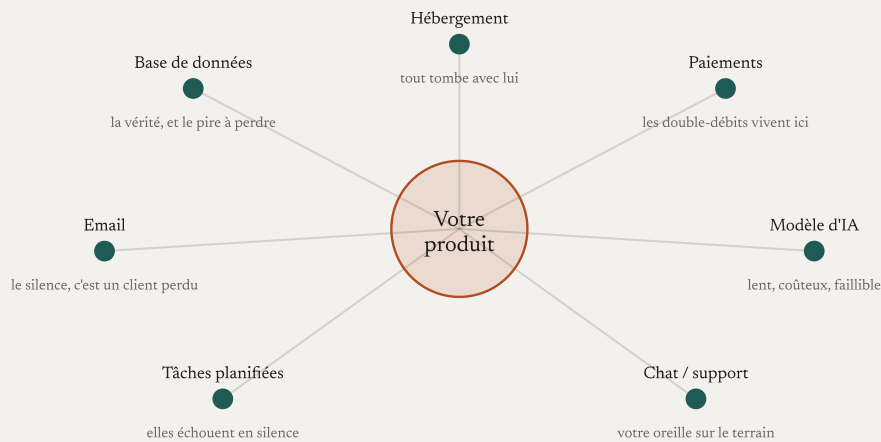
Vous ne les construirez pas de mémoire. Vous les reconnaîtrez, et vous saurez lesquelles une fonctionnalité touche, pour poser les bonnes questions.

Une API est l'interface que le frontend, ou un tiers, utilise pour parler au backend ; une requête atterrit sur un point d'entrée, une URL et une méthode. L'authentification consiste à prouver qui vous êtes, généralement avec un jeton signé envoyé à chaque appel et vérifié par le serveur. Un webhook est un appel qu'un tiers vous adresse quand quelque chose se produit, comme un prestataire de paiement qui annonce « cette facture est réglée ». La limitation de débit plafonne le nombre de requêtes par période pour empêcher les abus. L'idempotence garantit que la même opération jouée deux fois ne double pas son effet, ce qui est vital pour les paiements et les webhooks rejoués. Le chiffrement protège les données sensibles au repos, comme des clés stockées et verrouillées par un secret côté serveur.

## 2.5 Les dépendances sont un pouvoir et un risque

Chaque paquet que vous installez est un levier que vous n'avez pas eu à construire, et une porte à laquelle vous devez désormais faire confiance. Deux habitudes gardent la chose saine. Verrouillez vos versions : un fichier de verrouillage fige la version exacte de tout, pour que votre build soit identique aujourd'hui et dans six mois, sur votre machine et en production. Et résistez à la correction automatique aveugle : une commande qui « corrige » tous les avertissements de sécurité en montant tout de force cassera joyusement votre application pour faire taire un avertissement qui ne vous concernait pas. Lisez ce qu'une correction change avant de la lancer. Moins de dépendances, choisies exprès, valent mieux qu'un tas dont vous ne pouvez pas rendre compte.

## 2.6 Votre produit est un graphe de services



Votre produit est un graphe de services. Chaque nœud est un point de valeur et un point de défaillance.

Un produit moderne n'est pas une seule chose, c'est un graphe : une base de données, un prestataire de paiement, un envoyeur d'emails, un modèle d'IA, une intégration de chat, un hébergeur, un exécuter de tâches. Chaque nœud est un point de valeur et un point de panne. Dessinez la carte, même grossièrement, et pour chaque nœud posez deux questions : qu'arrive-t-il à mes utilisateurs si celui-ci tombe, et combien celui-ci me coûte. Vous n'avez pas à rendre chacun incassable. Vous devez savoir où sont les points de défaillance uniques, pour qu'une panne soit un risque connu et non une surprise.

# CI/CD et contrôles mécaniques

## 3.1 Les mots, d'abord

CI, intégration continue : à chaque changement, des contrôles automatiques se lancent (build, tests, lint) pour attraper les régressions tôt. CD, déploiement ou livraison continue : une fois vérifié, le code s'écoule vers la préprod puis la prod. Un pipeline est cette chaîne d'étapes. Un verrou est tout contrôle automatique capable de bloquer la ligne.

## 3.2 Ce que l'automatisation coûte, et ce qu'elle garantit

L'automatisation n'est pas gratuite. Chaque contrôle coûte du temps d'exécution, de l'argent en calcul, et de l'attention (un contrôle bruyant finit ignoré). Vous choisissez donc un régime, pas « tous les contrôles ». Un régime léger lance les contrôles pas chers et à forte valeur à chaque changement (est-ce que ça build, est-ce que ça type-check, une passe de style rapide) et garde les coûteux (bout en bout complet, tests de charge) pour les moments qui comptent.

Note honnête, parce que ce livre ne survend pas : sur mes propres produits, une suite de tests automatisée complète dans le pipeline n'est souvent pas branchée dans la première version. C'est un arbitrage précoce assumé, ajouté après le lancement. Ce qui tourne automatiquement : le site se redéploie à chaque push, et le build doit passer sinon rien ne sort. Le backend se déploie à la main, exprès. Ce mélange, automatique sur le site et manuel sur le backend, est un choix stable, pas une corvée inachevée. Dites ce que vous faites et ce que vous ne faites pas.

La version honnête d'un arbitrage a deux moitiés : le nommer comme un choix délibéré, et nommer le déclencheur qui le fera basculer. « Pas encore de suite de tests automatisée dans le pipeline, parce qu'au début une QA manuelle sur une petite surface va plus vite ; on ajoute la suite dès que la

surface dépasse ce qu'une personne peut vérifier à la main, ou dès que la première régression atteint un utilisateur. » Un arbitrage avec un déclencheur est un plan. Un arbitrage sans déclencheur est un trou que vous faites passer pour un plan.

### **3.3 Bloquant ou informatif**

Tous les contrôles ne doivent pas bloquer. Certains arrêtent la ligne : le build échoue, rien ne sort. D'autres se contentent d'informer : un avertissement que vous lisez et sur lequel vous tranchez. Sachez lequel est lequel, parce qu'un contrôle que vous croyez protecteur ne fait peut-être que chuchoter.

### **3.4 Les façons dont un verrou ment, et comment y remédier**

C'est la Loi 5 rendue pratique. Un verrou peut passer au vert sans rien protéger. Il analyse la mauvaise zone. Il ne voit que les fichiers déjà enregistrés dans le versionnage, donc du code tout neuf apparaît comme « tout va bien ». Son motif de recherche rate la vraie forme du code.

L'exécuteur compte « rien à faire » comme un succès. Une suite de tests existe mais est branchée pour ne tourner nulle part, donc elle passe en ne s'exécutant jamais.

Le remède est une discipline. Avant de faire confiance à un contrôle sur un terrain neuf, faites-le échouer exprès. Puis figez un cas connu-bon et un cas connu-mauvais en tests, pour que le contrôle sache toujours distinguer « fonctionne » de « inerte ». Et gardez-le silencieux : un verrou qui crie au loup finit désactivé par ceux-là mêmes qu'il devait protéger.

# Les cycles de travail

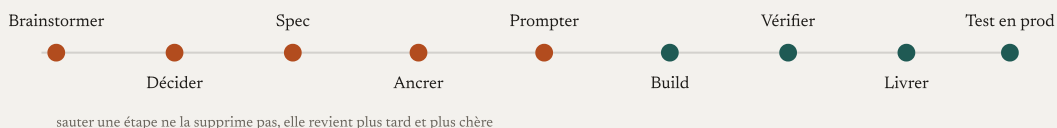
## 4.1 Un chantier n'est pas un incrément

Il y a deux tailles de travail, et les confondre cause l'essentiel du désordre. Un incrément est un petit changement autonome. Un chantier est un effort plus large, avec sa propre intention, son périmètre et sa spec. On ouvre un chantier délibérément. On ne laisse pas un « petit changement rapide » devenir en silence une refonte de trois semaines sans avoir jamais écrit ce qu'on est en train de faire.

## 4.2 La spec d'intention, écrite avant la moindre ligne de code

Avant de construire, écrivez le document de conception : l'objectif, l'état actuel du code, les décisions déjà prises, le modèle de données (nouvelles tables et colonnes), les tranches, la liste des tests, et ce qui est explicitement reporté. C'est la source de vérité, et elle permet à n'importe qui (une autre session, ou une IA) de reprendre le fil. L'écrire n'est pas du temps perdu. C'est l'endroit le moins cher où se tromper.

## 4.3 La chaîne de livraison, dans l'ordre



La chaîne se déroule dans l'ordre. Sauter un maillon ne fait pas gagner de temps — ça déplace le coût en aval.

Brainstormer, puis trancher les vrais arbitrages un par un, puis spécifier, puis ancrer (lire le code existant pour construire au bon endroit), puis prompter

l'exécutant, puis construire, puis vérifier (build, type-check, relecture, test dans un vrai navigateur), puis commiter, pousser, déployer, puis tester la tranche. Sauter un maillon ne fait pas gagner de temps. Ça déplace le coût en aval, là où il est plus gros.

#### 4.4 Classez le travail par réversibilité



La question de tri la plus utile qui soit : si ça tourne mal, combien coûte le retour en arrière ? Trois verdicts. Réversible à peu de frais, comme un ajustement visuel : allez vite, corrigez en avançant. Réversible avec effort, comme un changement de forme des données : vérifiez davantage, gardez un chemin de retour. Difficile à défaire, comme déplacer de l'argent, une migration destructrice, ou tout ce que les utilisateurs ne peuvent pas « dévoir » : ralentissez, prouvez, testez pour de vrai. On n'applique pas la même prudence partout. On la dépense là où le retour arrière coûte cher.

#### 4.5 Le débogage comme méthode, pas comme intuition

Quand quelque chose casse, ne tapotez pas au hasard. Travaillez par phases : reproduire de façon fiable, isoler l'endroit, formuler une hypothèse, tester cette hypothèse-là, corriger, puis prouver la correction avec le signal exact que le bug afficherait s'il revenait (Loi 4). Tapoter au hasard marche parfois et ne vous apprend rien. Une méthode marche et se capitalise.

## 4.6 Refactorer par vagues

Ne réécrivez pas tout d'un coup. Changez par petites vagues prouvables.

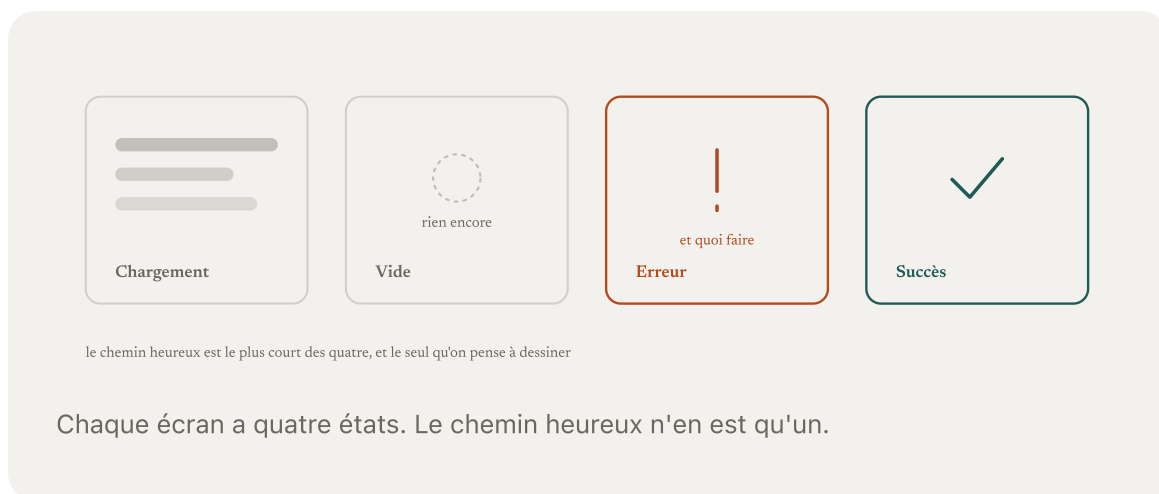
Après un changement de masse, prouvez-le par un invariant (Loi 6), pas par une relecture. Et ne lancez jamais deux vagues en parallèle sur la même zone : elles modifient les mêmes gros fichiers et se marchent dessus.

# Specs et cadrage, le cœur du métier

## 5.1 Ce que le builder tient vraiment

Le build est délégué. Le périmètre, non. Votre vrai métier consiste à décider ce qui se construit, dans quel ordre, et ce que « terminé » veut dire. Tout ce qui est en amont du code vous appartient : l'intention, les tranches, les arbitrages, le niveau d'acceptation. Perdez le périmètre, et aucune vitesse d'exécution ne vous sauvera.

## 5.2 L'inventaire d'écrans, le meilleur outil de cadrage qui existe



Avant de construire une fonctionnalité, listez chaque écran qu'elle touche, et pour chaque écran, les quatre états qu'il doit gérer : chargement, vide, erreur, succès. Cette seule habitude attrape l'essentiel du périmètre que vous découvririez sinon en plein build. Une fonctionnalité, ce n'est pas « le chemin heureux ». C'est le chemin heureux plus les trois états que tout le monde oublie. Jamais d'écran blanc.

Le principe plus profond : concevez pour le chemin qui échoue, pas seulement pour celui qui marche. L'état vide et l'état d'erreur ne sont pas des cas limites, c'est là que la confiance se gagne ou se perd. La plupart des fonctionnalités inachevées le sont précisément là.

### 5.3 « Terminé », dans ses trois formes

« Terminé » ne veut pas dire la même chose selon l'altitude, et nommer les niveaux évite les disputes. Terminé-codé : ça build, ça type-check, le changement existe. Terminé-vérifié : ça a été testé dans un vrai navigateur, les quatre états fonctionnent, les cas limites sont traités. Terminé-livré : c'est en production et vous l'avez confirmé en ligne, le test de fumée est passé. Une chose peut être terminée-codée et très loin d'être terminée-livrée. Dites toujours de laquelle vous parlez.

La Definition of Done, en checklist. Une tranche est terminée quand : elle build et type-check ; les quatre états fonctionnent dans un vrai navigateur ; les cas limites et les chemins d'argent sont testés pour de vrai ; c'est engagé et poussé ; c'est déployé et confirmé en ligne (test de fumée passé) ; et la doc ou le changelog est à jour. Tout ce qui est en deçà, c'est « à peu près terminé », autrement dit pas terminé.

### 5.4 La règle du go, et pourquoi elle est littérale

Quand vous dites « go », ça veut dire go : la décision est prise, on ne redébat pas. Quand quelque chose est marqué livré, c'est livré : on mesure et on itère, on ne le reconstruit pas. Ça paraît évident et c'est violé en permanence. Une grande part du temps perdu vient du rejugement de décisions déjà tranchées. Écrivez la décision, avec son alternative rejetée (Loi 3), et avancez.

### 5.5 Comment une demande devient du vrai travail

Une demande n'est pas une tâche. « Est-ce que ça pourrait aussi faire X » est une envie floue. La transformer en travail, c'est clarifier le résultat réellement attendu, le dimensionner (incrément ou chantier), trouver où il vit dans le code, trancher les arbitrages, puis le découper. L'écart entre « oui, facile » et le vrai travail est là où naît chaque mauvaise estimation. Cadrez par écrit, pas dans votre tête.

#### Pour le CTO en vous

la distinction qui évite le plus de conflits est bug ou évolution. Un bug, c'est le produit qui ne fait pas ce qui avait été spécifié. Une évolution, c'est une chose nouvelle que vous voulez maintenant. Urgences différentes, responsables différents, files différentes. Les confondre transforme chaque « il faudrait aussi que » en fausse urgence.

### 5.6 Parcourez le trajet avant de spécifier, et découpez verticalement

Avant d'écrire une spec, parcourez tout le trajet de l'utilisateur à voix haute, étape par étape : comment cette personne arrive, se connecte, atteint la chose, et que voit-elle à chaque étape. Une spec qui verrouille parfaitement le modèle de données et les points d'entrée mais saute le « comment se connecte-t-elle » livre une fonctionnalité qui existe techniquement et pas concrètement. La question « montre-moi le trajet de bout en bout » vient avant la spec, pas après.

Et découpez verticalement, pas horizontalement. Livrez d'abord un chemin complet et fin (un utilisateur s'inscrit, se connecte, atterrit sur un écran vide qui fonctionne), puis approfondissez. Construire toute la plomberie à l'horizontale avant qu'un seul trajet ne fonctionne de bout en bout, c'est la meilleure façon d'accumuler une classe de bugs qui n'apparaissent qu'au moment où les morceaux se rencontrent enfin.

### 5.7 La spec qu'un agent ne peut pas mal lire

C'est la compétence au plus fort effet de levier de tout le rôle, et presque personne ne la nomme. Un humain lit une spec approximative et comble les trous avec son jugement. Un agent lit une spec approximative et comble les trous avec une supposition, en toute confiance. Vous écrivez donc pour le lecteur qui ne posera pas de question. Une spec qu'un agent ne peut pas mal lire contient : un objectif en une phrase ; les fichiers et fonctions exacts à

toucher, et une liste explicite de ce qu'il ne faut PAS toucher ; les décisions déjà prises, pour qu'il ne les reprenne pas ; les tests qu'il doit faire passer ; et un arbitrage à la fois, jamais un tas. L'ambiguïté n'est pas une petite taxe avec un agent. C'est l'échec lui-même. Si une phrase peut se lire de deux façons, un agent en choisira une et la construira très bien.

### **5.8 L'hygiène de décision : protéger la seule ressource rare**

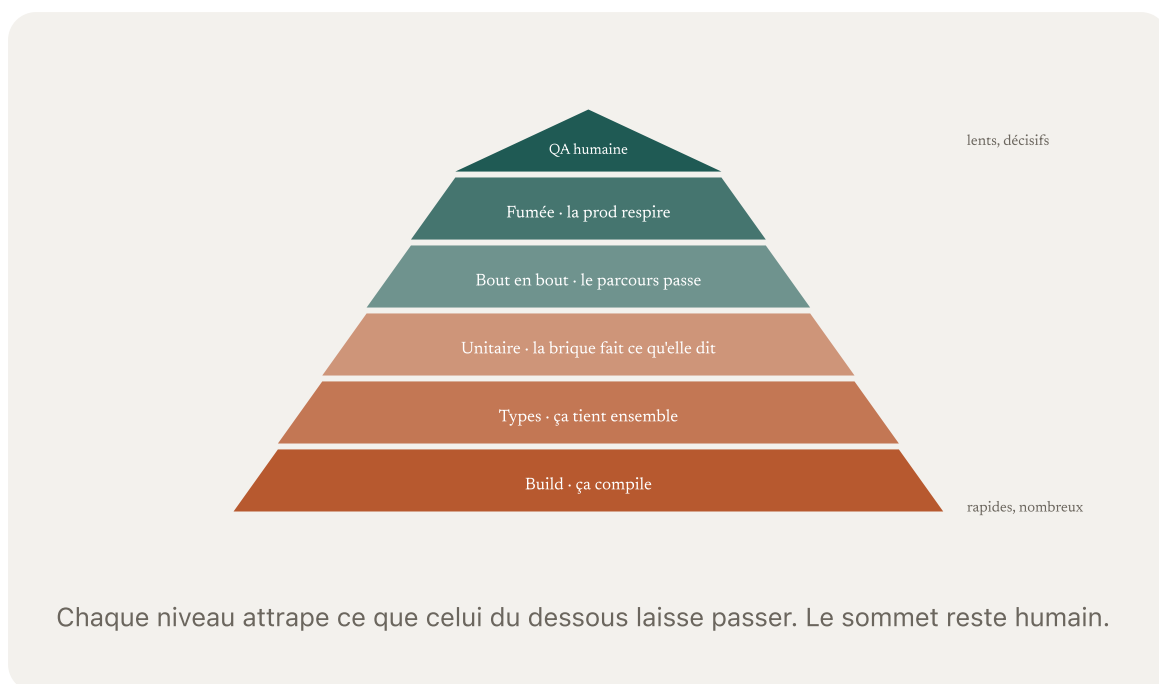
La ressource rare dans cette façon de travailler n'est pas la puissance de calcul, c'est votre attention de décideur. Protégez-la. Tranchez un arbitrage à la fois, pour que chacun reçoive une vraie réflexion. Dites non par écrit et avec une raison, pour que le non tienne et ne revienne pas la semaine suivante. Groupez les petits choix réversibles et allez vite dessus ; ralentissez uniquement sur ceux qui sont difficiles à défaire (Partie 4). Et tenez un journal de décisions avec l'alternative rejetée (Loi 3), pour ne jamais rejurer une décision acquise pendant qu'une nouvelle attend. Un builder qui dépense son attention sur tout n'en a plus pour les décisions que lui seul peut prendre.

# Tests et QA : la preuve contre l'intention

## 6.1 Le principe, en une ligne

Un test prouve ce que le code fait. Votre intention ne prouve rien. Toute la discipline consiste à remplacer « je suis à peu près sûr que ça marche » par « voici la preuve que ça marche ».

## 6.2 Les niveaux qui existent vraiment



Le contrôle de build : est-ce que ça compile. Si ça ne build pas, ça ne part pas. Le type-check : attrape une classe d'erreurs avant que quoi que ce soit ne s'exécute. Le lint : règles de style et de qualité. Le test unitaire : une fonction isolée. Le test de bout en bout : un vrai trajet utilisateur simulé dans un navigateur. Le test de fumée : une vérification rapide en conditions réelles juste après le déploiement, comme le point d'entrée qui répond ou le paiement qui débite le bon montant. La QA humaine : une personne qui déroule une checklist sur les trajets complexes (paiement, emails, tâches planifiées) avant une mise en ligne. Vous n'avez pas besoin de tous dès le

premier jour. Vous avez besoin de savoir lesquels un changement donné exige.

### **6.3 Les faux rouges, et pourquoi ils coûtent aussi cher que les faux verts**

Tout le monde s'inquiète du test qui passe alors qu'il ne devrait pas, le faux vert (Loi 5). L'inverse est tout aussi dangereux : un test qui échoue alors que rien ne va mal, le faux rouge. Un faux rouge vous entraîne à ignorer le contrôle. Après la troisième fois où un verrou crie au loup, vous cessez de le lire, et il ne protège plus rien, même quand il a raison. Un contrôle instable n'est pas un désagrément mineur. C'est un contrôle en route vers la désactivation.

### **6.4 Ce que la QA humaine fait et que les tests ne feront jamais**

Les tests automatisés vérifient ce que vous avez pensé à vérifier. Une personne qui déroule le parcours remarque ce à quoi vous n'avez pas pensé : la formulation qui embrouille, l'état qui semble faux, la chose qui fonctionne techniquement mais qu'aucun utilisateur ne comprendrait. Sur les chemins d'argent en particulier (paiements, remboursements, tout ce qui a un effet financier réel), vous validez en mode test réel, avec un vrai événement, pas en lisant le code et en raisonnant que ça devrait aller. Reasonner n'est pas prouver.

### **6.5 Comment prouver qu'une garde garde vraiment**

Quand vous ajoutez un contrôle de sécurité, prouvez qu'il garde comme vous prouveriez n'importe quel invariant : donnez-lui la mauvaise entrée exprès et confirmez qu'il bloque. Une garde que vous n'avez jamais vue qu'en train de passer est une garde que vous n'avez pas testée. C'est encore la Loi 5, braquée cette fois sur votre propre code de sécurité.

## 6.6 Testez tôt, dès la première tranche

L'erreur de test la plus coûteuse, c'est de tester tard. Si vous construisez dix tranches avant de faire tourner l'ensemble de bout en bout, une casse dans la tranche un reste cachée jusqu'à la tranche dix, et vous voilà à déboguer à travers neuf couches pour la trouver. Faites tourner le vrai trajet dès que la première version fine existe. Un bug attrapé à la tranche un, c'est cinq minutes de correction. Le même bug attrapé à la tranche dix, c'est un après-midi.

# Données, migrations et argent

## 7.1 Le schéma, et comment le changer sans danger

Votre base de données a une structure : des tables, des colonnes, des relations. Une migration est un changement versionné de cette structure, un fichier qui ajoute une table, une colonne, un index. Le motif sûr, à copier : d'abord une étape en lecture seule qui vérifie l'état actuel sans rien écrire (les tables cibles existent-elles déjà, les choses dont je dépends existent-elles). Si l'étape de lecture vous surprend, arrêtez-vous avant d'écrire. Ensuite l'étape d'écriture, enveloppée pour s'appliquer entièrement ou se défaire entièrement, jamais à moitié.

### Pour le CTO en vous

verrouillez la base par défaut. Activez la sécurité au niveau des lignes sur chaque table, et laissez uniquement le rôle côté serveur lire et écrire. Le frontend ne touche jamais directement la base ; tout passe par le backend. Une table dont la sécurité est désactivée est une porte que vous aviez oublié d'avoir laissée ouverte.

## 7.2 L'argent se stocke dans la plus petite unité

Ne stockez jamais de l'argent sous forme de décimal arrondi. Stockez-le dans la plus petite unité, en nombre entier de centimes. Les décimaux dérivent et s'arrondissent de façons qui créent ou perdent de l'argent en silence. Ce n'est pas du pédantisme, c'est la différence entre des comptes qui tombent juste et des comptes qui ne tombent pas juste.

### **7.3 L'idempotence : le même événement deux fois, un seul effet**

Les tiers vous enverront deux fois le même événement : un webhook de paiement est rejoué, une requête se relance. Si « paiement reçu » s'exécute deux fois, vous ne devez pas créditer deux fois. La garde habituelle est une contrainte d'unicité, un identifiant unique sur l'événement, pour que la seconde arrivée soit ignorée en silence. Concevez pour le rejeu, parce que le rejeu arrivera.

### **7.4 Prouver un changement de données en masse**

Quand vous changez des données à l'échelle, vous le prouvez par un invariant, pas par une relecture (Loi 6). Après une mise à jour en masse, vérifiez que les décomptes correspondent et qu'il ne reste rien de l'ancienne forme. Avant et après une grosse migration, prouvez que les lignes qui n'auraient pas dû bouger sont identiques octet pour octet. Et prouvez le résultat sur les vraies données sous forme de delta : ces nombres sont passés de X à Y, et rien n'a atterri là où il ne fallait pas. Un test vert dit « mon code s'est exécuté ». Un delta dit « le monde a changé comme je le voulais ».

# Livrer, sécuriser, opérer

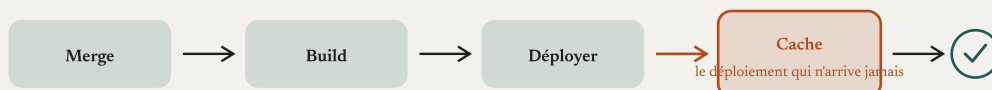
## 8.1 Le merge est le milieu de la chaîne, pas la fin

Fusionner votre changement dans main donne l'impression d'être la ligne d'arrivée. Ça ne l'est pas. Le merge déploie le code, mais beaucoup de choses arrivent après : le build s'exécute, le déploiement publie, le cache doit parfois être vidé, le schéma doit parfois être migré à part. « Fusionné » n'est pas « en ligne et correct ». Confirmez en ligne.

## 8.2 Déployer le code n'est pas déployer le schéma

Votre code et votre base de données changent sur des voies différentes. Un déploiement livre du code. Il n'exécute pas votre migration de base de données. Si le nouveau code attend une colonne que la migration n'a pas encore ajoutée, ça casse en production pendant que votre local a l'air parfait. L'ordre compte : en général le changement de schéma passe en premier, et reste rétrocompatible, puis le code. Ne supposez jamais que l'un a emporté l'autre.

## 8.3 Le cache : le déploiement invisible qui n'a jamais lieu



Le merge est le milieu de la chaîne, pas la fin.

tant que vous n'avez pas vu la chose vivante de vos yeux, ce n'est pas livré

Le cache est le seul maillon qui ment sans erreur : tout est vert, et l'ancienne version est toujours servie.

Vous déployez, vous vérifiez, rien n'a changé. La raison la plus fréquente : un cache, une couche qui sert une copie enregistrée pour aller plus vite, distribue encore l'ancienne version. Sur certaines surfaces, il faut purger le cache après chaque déploiement, sinon les utilisateurs voient le produit d'hier. Le déploiement qui « n'a pas pris » est presque toujours un cache qui n'a pas été vidé.

## **8.4 Le rollback**

Livrer sans chemin de retour est un pari, pas un plan. Avant de déployer quoi que ce soit de difficile à défaire, sachez comment vous l'annuleriez : revenir sur le code, ou avancer avec un correctif, et que faire des données déjà modifiées. Le moment de réfléchir au rollback n'est pas pendant l'incident.

## **8.5 La sécurité comme surface déclarative**

L'essentiel de votre sécurité n'est pas du code astucieux, ce sont quelques déclarations que vous avez faites ou oubliées. Les secrets vivent hors du versionnage, posés chez l'hébergeur. La clé puissante de la base est uniquement côté serveur ; le frontend reçoit la clé publique restreinte. La sécurité au niveau des lignes est active sur chaque table. Les limitations de débit plafonnent les abus. Ce ne sont pas des fonctionnalités qu'on construit une fois puis qu'on admire. Ce sont des surfaces qu'on garde planes. Une seule table sans sécurité, ou une seule clé toutaccès dans le navigateur, et tout le reste de votre soin ne compte plus.

## **8.6 L'authentification pose deux questions, pas une**

L'authentification demande qui vous êtes. L'autorisation demande ce que vous avez le droit de faire. Ce sont deux choses différentes, et les confondre est un trou classique : un utilisateur connecté n'est pas automatiquement un utilisateur autorisé. Vérifiez les deux, à chaque action sensible, côté serveur. Et appliquez le moindre privilège partout : chaque clé, chaque rôle, chaque

intégration reçoit le plus petit jeu de permissions qui lui permet de faire son travail, et rien de plus. Le rayon d'explosion d'une fuite est exactement le privilège que vous avez accordé.

Deux réflexes qui ne coûtent rien et vous sauvent : ne journalisez jamais un secret (jetons et clés ne doivent jamais atterrir dans vos logs, où ils vivent éternellement et circulent), et servez tout en HTTPS, toujours.

## **8.7 Vie privée et données personnelles**

Si vous touchez à des données personnelles (un nom, un email, tout ce qui identifie un humain), vous héritez de responsabilités, pas seulement de fonctionnalités. Les bases qu'un builder doit tenir : ne collecter que le minimum nécessaire, savoir où ça vit, pouvoir le supprimer sur demande, et ne jamais stocker plus que ce que vous pouvez protéger. Si vos clients sont en Europe, c'est le rez-de-chaussée du droit de la vie privée, et « nous sommes un petit produit » n'est pas une exemption. La vie privée est une contrainte de conception qu'on porte dès le premier jour, pas une case à cocher avant le lancement.

## **8.8 L'observabilité : comment vous savez que la production va bien**

On ne répare pas ce qu'on ne voit pas. Trois couches. Les logs : le registre de ce qui s'est passé, que vous pouvez diffuser en direct pour observer un problème pendant qu'il se produit. Le suivi d'erreurs : un service qui capture chaque plantage avec son contexte, pour que vous appreniez l'existence d'un bug par un tableau de bord plutôt que par un utilisateur furieux. Les alertes : un signal qui vous atteint quand un chiffre clé dérape, avant que les clients ne le remarquent. Au début, on découvre souvent qu'une chose a cassé parce que quelqu'un l'a dit. Refermer cet écart, passer de « un utilisateur m'a prévenu » à « mon tableau de bord m'a prévenu », est l'une des premières montées en gamme qui séparent un jouet d'un produit.

## 8.9 Le playbook d'incident et de rollback

Des choses casseront en production. La panique est la partie coûteuse. Ayez un plan avant d'en avoir besoin. Quand ça casse : arrêtez d'abord l'hémorragie (revenez à la dernière version connue bonne, ou désactivez la fonctionnalité derrière son interrupteur), puis diagnostiquez calmement, puis corrigez en avançant avec une vraie correction et son contrôle de régression (Loi 4). Décidez à l'avance ce que « revenir en arrière » signifie chez vous : revenir sur le code est facile, mais les données déjà modifiées peuvent exiger leur propre annulation. Un rollback répété est un haussement d'épaules. Un rollback improvisé est une panne.

Deux soupapes de sécurité peu coûteuses facilitent tout ça. Mettez vos chemins risqués, tout ce qui touche à l'argent ou aux webhooks tiers, derrière un interrupteur, pour pouvoir les couper d'un geste sans redéployer. Et séquencez vos mises en ligne pour valider avant de promouvoir : prouvez que la chose fonctionne avant de construire les pages marketing qui y envoient les gens. Valider, puis promouvoir, jamais l'inverse.

## 8.10 Versionnage et mises en ligne

Donnez des noms et une histoire à vos versions. Le versionnage sémantique est la convention courante : une version du type majeure.mineure.correctif, où les nombres signalent l'ampleur du changement (un correctif répare, une mineure ajoute, une majeure casse). Étiquetez chaque version, et tenez un changelog : la liste datée et lisible de ce qui est sorti. Pendant une bêta, vous pouvez suffixer la version pour marquer la phase. Le but n'est pas le cérémonial, c'est de pouvoir dire exactement ce qu'il y a en production et de revenir volontairement à n'importe quel point antérieur.

## 8.11 Le coût est une préoccupation de premier plan

Un builder qui ignore le coût livre un produit incapable de survivre à son propre succès. Trois lignes à surveiller. L'infrastructure : ce que votre

hébergeur, votre base et vos services facturent à mesure que l'usage grandit. Les coûts tiers à l'usage : paiements, emails, et surtout les appels aux modèles d'IA, facturés au jeton et qui montent plus vite qu'on ne le croit. Et votre propre métrique de prix : si vous facturez une unité liée à l'usage, vous devez savoir ce que chaque unité vous coûte à délivrer, sinon vous vendez à perte sans le voir. Le coût n'est pas un sujet qu'on refile au financier. C'est une donnée de conception, au même titre que la vitesse ou la sécurité.

## **8.12 Annoncer que c'est en ligne**

Livrer n'est pas fini tant que vous n'avez pas raconté l'histoire. Une bonne note de version répond à : ce qui est sorti (la valeur, pas la liste technique), pourquoi (le problème résolu), une démo ou une capture, et la suite. Les formats qui marchent : un changelog daté et orienté valeur, une courte démo du vrai trajet, du build in public en chemin, une note à l'équipe sur ce qui change pour elle. Vous parlez valeur et impact, pas jargon. Le glossaire existe précisément pour traduire le technique en langage clair à ce moment-là.

# La documentation comme infrastructure

## 9.1 Trois niveaux d'entrée, un par lecteur

Une bonne documentation sert trois lecteurs à trois profondeurs : le nouveau venu qui a besoin de la carte, le builder qui a besoin du comment, le mainteneur qui a besoin du pourquoi. Une seule page ne peut pas bien servir les trois, alors on superpose : une vue d'ensemble au sommet, puis le guide pratique, puis la référence détaillée. Donnez à chaque lecteur la porte qui lui convient.

## 9.2 Une seule source de vérité par sujet

Chaque sujet a exactement une maison. Quand deux documents décrivent la même chose, ils divergent, et le lecteur ne sait plus lequel a raison. Si une information doit apparaître à deux endroits, l'un est la source et l'autre pointe dessus. Un fait avec deux propriétaires est un fait sans propriétaire.

## 9.3 Une décision s'écrit avec ses alternatives rejetées

La Loi 3, appliquée à la documentation. Un enregistrement de décision, ce n'est pas « on a choisi X ». C'est « on a choisi X plutôt que Y et Z, parce que, à cette date ». Les options rejetées sont la partie précieuse : elles empêchent le redébat six mois plus tard, et elles expliquent le choix à quelqu'un qui n'était pas là.

## 9.4 Geler, pas supprimer

Un plan ou un document obsolète ne se supprime pas. Il se marque obsolète, se date, et reste avec un renvoi vers la version vivante. La suppression perd le raisonnement, et la personne suivante redécouvre l'impasse depuis zéro. Le gel garde l'histoire à bas coût et la leçon intacte.

## **9.5 Le piège : le texte qui décrit du code n'est vérifié par personne**

Un commentaire ou un document qui dit « cette fonction renvoie X » n'est contrôlé par personne. Le code peut changer et la phrase reste, désormais mensongère avec une totale assurance. Préférez les descriptions vérifiables (un test, un type, un exemple qui s'exécute) à la prose, et traitez tout document que vous avez écrit vous-même comme une hypothèse datée jusqu'à l'avoir reconfirmée au réel (Loi 7).

## **9.6 Le test d'autosuffisance : le nouveau venu à froid**

Il existe un seul test pour savoir si votre documentation est une infrastructure et non une décoration : un nouveau venu à froid, un nouveau coéquipier ou un agent IA sans aucune mémoire du projet, peut-il devenir productif à partir de la seule documentation. S'il doit vous demander, la documentation a échoué, et vous êtes devenu le point de défaillance unique. Écrivez pour que la personne qui n'était pas là puisse démarrer sans vous. Une documentation autosuffisante est ce qui vous permet de passer la main, de vous absenter, ou de faire entrer quelqu'un sans réexpliquer le monde entier.

# Design system, contenu, cohérence

## 10.1 Les tokens : le style vit à un seul endroit

Un design system commence par les tokens, les variables nommées des couleurs, des espacements, de la typographie. Changez le token, tout le produit change avec lui. Codez une couleur en dur sur un écran et vous avez lancé une lente dérive que personne ne corrigera entièrement. La cohérence, ce n'est pas de la discipline appliquée mille fois, c'est une source unique appliquée partout.

## 10.2 L'invariant qui garde un produit cohérent

Choisissez une palette réduite et fixe, et tenez-la. Une poignée de couleurs utilisées avec du sens (une pour l'action principale, une pour le danger, une pour le succès) est ce qui fait qu'un produit ressemble à un seul produit. Le mode d'échec n'est pas la laideur, ce sont dix bleus légèrement différents qui semblaient tous très bien, isolément.

## 10.3 Le registre de composants

Réutilisez avant de recréer. Tenez un registre des briques (le bouton, la carte, le champ) et servez-vous-en. Chaque composant unique est une incohérence future et une maintenance future. Quand un fichier dépasse les deux cents lignes, c'est le signal de le découper, pas une réprimande.

## 10.4 Le contenu se calibre sur un corpus, il ne se décrète pas

La voix de votre produit (les micro-textes, les états vides, les messages d'erreur) ne s'invente pas écran par écran. Elle se calibre sur une référence : quelques exemples qui définissent le ton, et tout le reste s'y aligne. Un contenu décidé écran par écran finit par sonner comme cinq personnes différentes, parce qu'il a été écrit dans cinq humeurs différentes.

## 10.5 Performance et accessibilité, le minimum que tout builder doit

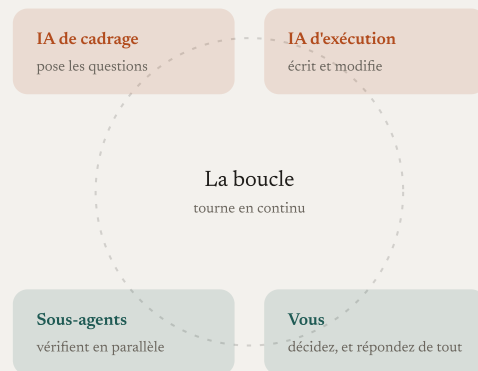
Deux choses que vous n'avez pas à maîtriser, mais que vous ne devez pas ignorer. La performance : un produit lent semble cassé même quand il fonctionne. Surveillez les bases (à quelle vitesse le premier vrai contenu apparaît, en combien de temps la page devient utilisable) et traitez une page lourde comme un bug, pas comme un détail. L'accessibilité : certains utilisateurs naviguent au clavier, d'autres avec un lecteur d'écran, d'autres ne supportent pas les animations. Les gains faciles portent loin : des états de focus visibles pour que les utilisateurs au clavier sachent où ils sont, le respect du réglage « réduire les animations », de vraies alternatives textuelles aux images, un contraste de couleurs suffisant. Ce n'est pas du vernis supplémentaire. C'est la question de savoir si une partie de vos utilisateurs peut simplement se servir de la chose.

# Construire avec l'IA : ce qui change vraiment

## 11.1 La nouvelle pratique

Le basculement profond : l'humain devient l'architecte et le décideur, l'IA devient le partenaire qui explore le code, propose des options, écrit la spec, génère le code et le vérifie. Vous ne tapez pas le code ligne par ligne. Vous cadrez, vous décidez, vous relisez, vous validez.

## 11.2 Les rôles



Quatre rôles autour d'une boucle : deux IA, les sous-agents, et vous.

L'IA de cadrage (la partenaire produit) : elle brainstorme, pose les questions d'arbitrage une par une, écrit la spec, rédige les prompts d'exécution, tient une mémoire du projet pour la continuité. L'IA d'exécution (l'agent qui code) : elle lit d'abord le dépôt, modifie les fichiers, écrit le code, lance le build et le type-check, signale ses angles morts. Les sous-agents : des explorateurs lancés en parallèle pour cartographier le code vite sans encombrer le fil principal. Vous : les décisions produit, les arbitrages, la vérification finale, le déploiement, les tests, la communication.

### 11.3 Ce que l'IA fait réellement quand elle code

Elle lit le code existant avant d'écrire (l'ancrage) : elle cartographie les fichiers, les conventions, les points d'insertion. Elle ne devine pas, elle va voir. Elle propose des options avec leurs arbitrages, mais ne tranche pas une décision produite à votre place. Elle écrit la spec puis le code, en réutilisant les motifs déjà présents dans le dépôt. Elle vérifie mécaniquement : compilation, types, syntaxe, style. Elle signale ses angles morts : ce qu'elle a validé par raisonnement et non en vrai, les valeurs par défaut qu'elle a choisies, les prérequis manuels (un secret à poser, une migration à appliquer). Et elle s'abstient délibérément de certaines choses, parce qu'elles vous appartiennent : trancher un arbitrage métier, pousser ou déployer sans votre go, déplacer de l'argent réel.

En une phrase pour le lecteur non développeur : l'IA est un exécutant très solide qui lit, propose, code et contrôle. Le pilote et le garde-fou, c'est vous.

L'économie du build, parce que c'est la première question d'un CPO. Votre temps se déplace : moins vers l'écriture de code, plus vers la décision de quoi construire et la vérification de ce qui revient. Vos coûts se déplacent aussi, de payer des heures à payer des jetons, l'unité que l'IA facture. Une session qui explore un gros codebase brûle des jetons vite, donc bien ancrer (envoyer l'agent sur les bons fichiers, pas sur tout le dépôt) est une décision de coût, pas seulement de vitesse. Le retour n'est pas du code gratuit, c'est la vitesse jusqu'à un vrai produit livrable, avec une personne au lieu d'une équipe. Surveillez votre dépense en jetons par tranche livrée et le rapport entre votre temps de décision et le temps d'exécution de l'agent, et regardez ce rapport s'améliorer à mesure que vos specs s'affûtent. Un prompt vague coûte deux fois : en jetons, et en reprises qu'il provoque.

## **11.4 Les modes d'échec propres aux agents**

Nouveaux outils, nouvelles façons d'échouer. Un agent peut se tromper avec assurance, énoncer un fait faux sur le même ton qu'un fait vrai (Loi 7). Il peut agir d'après un document périmé qu'il a lui-même écrit. Il peut « vérifier » quelque chose qu'il n'a jamais exécuté (Loi 4). Deux agents en parallèle peuvent se percuter sur les mêmes fichiers (Loi 9). Il peut dériver du plan au fil d'une longue session à mesure que sa mémoire se remplit (Loi 10). Rien de tout cela n'est une raison de ne pas utiliser d'agents. Ce sont les raisons de garder les contrôles humains qui les attrapent.

## **11.5 Ce que l'IA rend obligatoire et qui était optionnel**

Certaines pratiques étaient un confort avec un humain soigneux et deviennent non négociables avec un agent. L'ancrage avant d'agir. Une spec écrite (l'agent a besoin de la source de vérité que vous gardiez dans votre tête). Des contrôles mécaniques calibrés sur un cas connu-rouge (on ne peut pas se fier au seul rapport de l'agent sur lui-même). Une mémoire de projet avec un budget. Des frontières explicites « ne pas toucher » dans les prompts. L'agent ne rend pas ces choses optionnelles. Il en fait le prix de la vitesse en sécurité.

## 11.6 Ce qui reste humain



Les décisions. Les arbitrages. La vérification finale. Le déploiement de tout ce qui est difficile à défaire. Le goût. Et la responsabilité : quand c'est en ligne, c'est votre nom dessus, pas celui du modèle. C'est tout le marché, et c'est un bon marché.

# Le catalogue des pièges

Un index de référence rapide, pas un récit. Chaque piège ci-dessous est la forme en une ligne d'un principe issu des lois (Partie 1) ou d'un chapitre. Quand vous vous faites mordre, vous parcourez cette liste, puis vous allez lire la loi complète derrière. Tenez votre propre version, complétez-la au fil des morsures, et laissez chaque entrée porter son incident (Loi 1).

- **« Pas poussé » veut dire « perdu ».** Le travail local non poussé disparaît au prochain clone. Poussez après chaque unité.
- **Mauvais compte, « repository not found ».** Un push qui échoue est souvent la mauvaise identité active, pas une vraie erreur.
- **La coche verte qui n'a jamais tourné.** Une suite de tests branchée pour ne s'exécuter nulle part passe en ne faisant rien. Faites-la d'abord passer au rouge exprès.
- **La vérification écrite mais non exécutée.** Pire que rien : elle clôt le sujet. Livrez la commande et le signal de régression, pas la réassurance.
- **Le déploiement n'a rien changé ? Videz le cache.** Le « mon déploiement n'a pas pris » le plus courant est un cache périmé.
- **Le code est parti, le schéma non.** Un déploiement emporte du code, pas votre migration. L'ordre et la rétrocompatibilité comptent.
- **Le nom qui ment.** Un objet nommé d'une façon en désigne une autre ; une relation bâtie sur la ressemblance des noms rejette la plupart des lignes. Prouvez par l'usage, pas par les noms.
- **L'argent en décimal.** L'arrondi invente et perd de l'argent. Stockez les centimes en nombres entiers.
- **Le webhook rejoué.** Le même événement arrive deux fois. Sans garde d'unicité, vous doublez l'effet.
- **La clé tout-accès dans le frontend.** Une seule clé puissante qui fuite ouvre toute la base. Côté serveur uniquement, toujours.
- **Deux agents, un fichier.** Le travail en parallèle sur les mêmes gros fichiers entre en collision. Une vague à la fois par zone.
- **Le document auquel vous faites confiance et qui est une hypothèse.** Votre propre document est daté et non vérifié tant qu'il n'a pas été confronté au réel.

# Aller plus loin : ce que je n'ai pas encore fait, et à quoi ressemble le dev+++



Une section honnête, exprès. Tenir toute la boucle, c'est être clair sur les endroits où une équipe d'ingénierie aguerrie irait plus loin que moi, à ce jour. Ce n'est pas une confession. C'est une carte. Certains de ces points sont des arbitrages précoces assumés, d'autres sont les prochains sur la liste, et tous dessinent ce que monter d'un cran veut dire. Le dire à voix haute fait partie du métier, et de la confiance.

L'AI Product Builder vous emmène jusqu'à un produit réel, fonctionnel, en production, bien plus vite et bien plus loin que la plupart des gens ne le croient. Savoir exactement où commencent les pratiques d'équipe senior, et reconnaître qu'on n'y est pas encore tout à fait, est une force et non une faiblesse. Un builder qui prétend avoir tout fait est celui à qui on ne peut pas se fier.

Voici l'échelle, à peu près dans l'ordre où la plupart des produits construits en solo devraient la gravir. À affiner avec un regard d'ingénierie senior.

**Une couverture de tests automatisée dans le pipeline.** Des tests unitaires, d'intégration et de bout en bout qui tournent à chaque changement et bloquent une fusion quand ils échouent. Beaucoup de premières versions sortent avec des tests automatisés légers et une QA manuelle lourde, ce qui est un arbitrage précoce valable. La montée en gamme, c'est une vraie suite branchée dans le pipeline, pour que le filet vous rattrape avant qu'un humain n'ait à le faire.

**L'observabilité.** Des logs structurés, des métriques, du traçage et des alertes, pour apprendre l'existence d'un problème par un tableau de bord et non par un utilisateur en colère. Une installation senior vous réveille avant que les clients ne remarquent quoi que ce soit. Au début, on l'apprend souvent parce que quelqu'un le dit. Cet écart est l'un des premiers à refermer.

**Une posture de sécurité formelle.** Un modèle de menaces documenté, l'analyse des dépendances, la rotation des secrets, et à terme des certifications comme SOC 2 ou ISO quand les clients l'exigent. Au démarrage, on assume de ne pas les avoir, et on ne fait pas passer une capture d'écran de cadenas pour un programme de sécurité.

**Les tests de charge et de performance.** Prouver que le système tient à dix fois le trafic avant que ça n'arrive, pas après la panne ou la facture surprise. Tant que vous ne l'avez pas testé, « ça scale » est un espoir.

**Le plan de reprise après sinistre.** Des sauvegardes que vous avez réellement restaurées, et un plan répété pour le jour où quelque chose est perdu. Une sauvegarde jamais restaurée n'est pas une sauvegarde, c'est un vœu.

**L'infrastructure en tant que code.** Toute la stack définie dans des fichiers, pour que monter un environnement neuf soit une commande, et non un souvenir et un après-midi chanceux.

**La livraison progressive.** Interrupteurs de fonctionnalité, versions canari, déploiements par paliers, pour qu'un mauvais changement atteigne un pour cent des utilisateurs avant de les atteindre tous.

**Accessibilité et internationalisation aux standards.** Pas au feeling et à la bonne intention, mais face à de vraies recommandations et avec de vrais tests.

Rien de tout cela ne rend les chapitres précédents moins vrais. Vous pouvez livrer un vrai produit, avec de vrais clients payants, sans tout ça. Cette partie existe pour que vous sachiez que la route continue, et pour qu'un ingénieur senior qui lit votre travail voie quelqu'un qui sait exactement où il en est.

## **Ce que j'ai tenté et supprimé**

Un livre qui ne raconte que des victoires n'est pas crédible. Quelques choses que j'ai construites puis retirées, anonymisées. Un système de taxonomie censé organiser le contenu, qui ajoutait plus de complexité qu'il n'en a jamais remboursé : coupé. Un composant que j'avais promu dans le design system partagé et qu'aucun écran n'a jamais utilisé : supprimé. Un grand plan de milliers de pages générées automatiquement : abandonné dès que le calcul effort/valeur est devenu négatif. Chacun a enseigné la même leçon : un chemin rejeté n'est pas perdu si vous écrivez pourquoi vous l'avez rejeté (Loi 3). Le cimetière fait partie de la carte.

# Glossaire

## DÉPLOIEMENT

### Verrou

Un contrôle automatique qui bloque la ligne.

voir Loi 5, Loi 8, Partie 3

## DONNÉES

### Idempotence

La même opération deux fois un seul effet.

voir Partie 7.3

Comment lire une entrée : catégorie, terme, définition simple, et où on l'utilise.

## Versionnage et collaboration

### Dépôt (repo)

le projet versionné.

### Commit

un instantané daté d'un ensemble de changements, avec un message.

### Branche

une ligne de travail parallèle. Main est la ligne de production.

### Remote / origin

la copie de référence hébergée en ligne.

### Push / pull

envoyer vos commits vers le haut / récupérer ceux des autres.

### Clone

copier un dépôt sur votre machine.

### Merge (fusion)

replier une branche dans une autre.

### Pull request (PR)

une demande de fusion, généralement relue avant.

### Revue de code

un pair (ou l'IA) qui relit avant la fusion.

---

## Arbre de travail

l'état de vos fichiers en local, commités ou non.

---

## Déploiement et infrastructure

---

### Build

compiler et préparer les fichiers livrables.

---

### Déploiement

publier le build, le mettre en ligne.

---

### Rollback

revenir à la version précédente.

---

### CI/CD

automatisation du build, puis des tests, puis du déploiement.

---

### Pipeline

la chaîne d'étapes automatisées.

---

### Verrou (gate)

un contrôle automatique capable de bloquer la ligne.

---

### Environnement

local, préprod/staging, production.

---

### Variable d'environnement

une valeur de configuration qui change selon l'environnement.

---

### Secret

une valeur sensible (une clé), jamais dans le versionnage.

---

### Cache

une copie enregistrée servie pour la vitesse. Purgez-la après certains déploiements, sinon vous servez l'ancienne version.

---

### CDN

un réseau qui sert le contenu rapidement depuis des copies en cache.

---

### URL de prévisualisation

une URL de test générée pour une branche avant la production.

---

## Backend et données

---

### API

l'interface utilisée pour parler au backend.

---

**Point d'entrée (endpoint)**

une URL et une méthode (par exemple un appel de création d'événement).

---

**Authentification / jeton**

prouver qui vous êtes avec un jeton signé, vérifié par le serveur.

---

**Webhook**

un appel entrant qu'un tiers vous adresse lors d'un événement.

---

**Schéma**

la structure de la base de données (tables, colonnes, relations).

---

**Migration**

un changement versionné du schéma.

---

**Sécurité au niveau des lignes**

des règles d'accès appliquées ligne par ligne, pour que le frontend ne lise pas ce qu'il ne doit pas.

---

**Clé tout-accès et clé publique**

la clé puissante (côté serveur uniquement) et la clé restreinte (sans danger pour le frontend).

---

**Limitation de débit**

plafonner les requêtes par période pour empêcher les abus.

---

**Idempotence**

la même opération deux fois, un seul effet.

---

**Chiffrement**

protéger les données sensibles au repos.

---

**Cron**

une tâche qui s'exécute selon un calendrier.

---

**Frontend**

---

**Composant**

un bloc d'interface réutilisable.

---

**État (d'un écran)**

chargement, vide, erreur, succès. Traitez les quatre.

---

**Token de design**

une variable de style nommée (couleur, espacement, typographie).

---

**Interrupteur de fonctionnalité**

un commutateur qui affiche ou masque une fonctionnalité selon l'environnement. Désactivé par défaut signifie non rendu, pas seulement grisé.

---

---

## Interface optimiste

mettre l'écran à jour immédiatement, avant la confirmation du serveur.

---

## Tests et qualité

---

### Type-check

vérifier la forme des données avant que quoi que ce soit ne s'exécute.

---

### Lint

contrôler les règles de style et de qualité.

---

### Test unitaire

vérifier une fonction isolée.

---

### Test de bout en bout

simuler un trajet utilisateur complet.

---

### Test de fumée

une vérification rapide en conditions réelles après le déploiement.

---

### QA

assurance qualité, souvent une checklist manuelle avant une mise en ligne.

---

### Régression

quelque chose qui fonctionnait et ne fonctionne plus.

---

### Faux vert / faux rouge

un contrôle qui passe alors qu'il ne devrait pas / qui échoue alors que rien ne va mal.

---

### Invariant

un fait qui doit tenir, utilisé pour prouver un changement de masse.

---

### Hotfix

un correctif urgent poussé rapidement.

---

## Sécurité, opérations et coût

---

### Authentification et autorisation

qui vous êtes, et ce que vous avez le droit de faire. Vérifiez les deux.

---

### Moindre privilège

accorder les plus petites permissions qui font le travail. Le rayon d'explosion d'une fuite égale le privilège accordé.

---

### Données personnelles

tout ce qui identifie une personne. Collectez le minimum, protégez-le, sachez le supprimer.

---

## **OAuth**

une façon standard de laisser un utilisateur accorder un accès limité sans partager son mot de passe.

---

## **HTTPS**

transport chiffré. Partout, toujours.

---

## **Observabilité**

logs, suivi d'erreurs et alertes, pour apprendre un problème par un tableau de bord et non par un utilisateur.

---

## **Suivi d'erreurs**

un service qui capture les plantages avec leur contexte.

---

## **Alertes**

un signal qui vous atteint quand un chiffre clé dérape.

---

## **Incident**

quelque chose a cassé en production ; le plan est : arrêter l'hémorragie, diagnostiquer, corriger en avançant.

---

## **Semver (versionnage sémantique)**

majeure.mineure.correctif, qui signale l'ampleur d'un changement.

---

## **Changelog**

la liste datée et lisible de ce qui est sorti.

---

## **Fichier de verrouillage**

le fichier qui fige les versions exactes des dépendances pour des builds reproductibles.

---

## **Monorepo et multi-repo**

un dépôt qui contient plusieurs projets, ou plusieurs dépôts séparés.

---

## **Staging**

un environnement de préprod qui reflète la production pour une dernière vérification.

---

## **Économie unitaire**

ce qu'une unité d'usage vous coûte à délivrer, face à ce que vous la facturez.

---

## **Construire avec l'IA**

---

### **Prompt**

l'instruction que vous donnez à l'IA.

---

### **Ancrage (grounding)**

l'IA qui lit le code existant avant d'agir.

---

### **Agent / sous-agent**

une IA qui travaille en autonomie sur une tâche, et des explorateurs lancés en parallèle.

---

---

**Spec (document de conception)**

le document qui fixe le périmètre et les décisions.

---

**Vague**

une unité de travail petite, testable et livrable.

---

**LLM**

le grand modèle de langage qui génère.

---

**Jeton (IA)**

l'unité de texte qu'un LLM lit et écrit, et la base de la facturation IA.

---

**Instructions permanentes**

les règles maison répétées dans chaque prompt.

# L'auto-évaluation

## Processus et décisions

- ☐ Écrivez-vous une spec courte avant le build, et non après ?
- ☐ Tranchez-vous un arbitrage à la fois, par écrit, avec l'option rejetée notée ?
- ☐ Pouvez-vous dire, là maintenant, ce que « terminé » veut dire chez vous, et cela inclut-il « en production et vu fonctionner » ?
- ☐ Découpez-vous verticalement, un trajet complet et fin d'abord, puis la profondeur ?

## Versionnage et mise en ligne

- ☐ Poussez-vous après chaque unité de travail, systématiquement ?
- ☐ Avant de déployer quelque chose de risqué, savez-vous exactement comment vous reviendriez en arrière ?
- ☐ Tenez-vous un changelog et versionnez-vous vos mises en ligne ?
- ☐ Vos chemins risqués (argent, webhooks) sont-ils derrière un interrupteur que vous pouvez couper sans redéployer ?

## La preuve avant l'intention

- ☐ Avez-vous vu chacun de vos contrôles de sécurité échouer exprès au moins une fois ?
- ☐ Testez-vous le vrai trajet tôt, dès la première tranche, et non à la fin ?
- ☐ Sur les chemins d'argent, validez-vous avec un vrai événement de test, et pas seulement par une lecture du code ?
- ☐ Quand vous changez des données à l'échelle, le prouvez-vous par un invariant plutôt que par une relecture ?

## Sécurité, données, opérations

- ☐ Votre clé tout-accès est-elle uniquement côté serveur, jamais dans le frontend ?
- ☐ Savez-vous où vit chaque variable d'environnement et qui la pose ?
- ☐ Les données personnelles sont-elles collectées au minimum, protégées, et supprimables sur demande ?
- ☐ Apprenez-vous un problème de production par un tableau de bord, ou par un utilisateur en colère ?

## Documentation et IA

- ☐ Un nouveau venu à froid, humain ou agent, pourrait-il devenir productif à partir de votre seule documentation ?
  - ☐ Vos specs sont-elles écrites pour qu'un agent ne puisse pas raisonnablement les mal lire (périmètre explicite, ne-pas-toucher, un seul objectif) ?
  - ☐ Tenez-vous une mémoire de projet avec un budget, qui tend vers le vide ?
  - ☐ Quand c'est en ligne, est-il clair que la décision, la vérification et la responsabilité étaient les vôtres ?
- 

## Score

Comptez les oui, sur 20.

**16–20 • Vous construisez comme un pro.** vous construisez comme un pro ; gardez l'honnêteté sur ce qui vient ensuite (Partie 13).

**10–15 • Une boucle solide.** une boucle solide, et les manques sont en général du côté de la preuve et des opérations, pas du code.

**0–9 • Vous livrez à l'espoir.** vous livrez à l'espoir ; commencez par la règle du push, le test précoce, et une vraie définition de terminé.

## Notes de cas (anonymisées)

Les mêmes leçons, sans noms, sans adresses. Chaque note est une vraie forme de problème, débarrassée de tout ce qui pourrait remonter à un produit en ligne. Ce qui compte, c'est le motif, pas l'adresse.

**Un SaaS grand public.** Un produit où les utilisateurs envoient des événements et reçoivent en retour des actions automatisées. Les leçons les plus tranchantes portaient sur l'argent et le jeu : stocker l'argent en centimes entiers, et garder chaque événement entrant d'un tiers derrière une contrainte d'unicité pour qu'un webhook rejoué ne double jamais un effet (Partie 7). Et sur l'écran d'accueil : la plus grande victoire produit a été de retirer une fonctionnalité, pas d'en ajouter une, parce que ce dont les utilisateurs avaient réellement besoin était « qu'est-ce que je fais maintenant », pas « voici vos données » (Loi 3 sur les chemins rejetés, et le goût qui reste humain, Partie 11).

**Un outil interne.** Un back-office utilisé par une petite équipe d'opérations. Les leçons portaient sur la vérité des données et le changement de masse : un objet dont le nom ne correspondait pas au sens faisait rejeter à une relation la plupart de ses lignes, jusqu'à ce qu'on la corrige par l'usage plutôt que par le nom (Loi 7), et chaque changement de données en masse était prouvé par un invariant et un delta en production, jamais par une relecture (Loi 6).

**Une application client.** Un produit construit pour quelqu'un d'autre, où la documentation et la vérification pesaient le plus lourd. Les deux leçons les plus coûteuses : une vérification écrite mais jamais réellement exécutée, qui a laissé des fichiers exposés pendant des mois en silence (Loi 4), et la discipline d'écrire chaque décision avec ses alternatives rejetées pour que le même débat ne se rouvre pas des mois plus tard avec quelqu'un qui n'était pas là (Loi 3).

# Continuer

---

Travailler avec moi – [benoitlecureur.com/fr/work-with-me](https://benoitlecureur.com/fr/work-with-me)

---

S'abonner à Off Market – [benoitlecureur.com/fr/off-market](https://benoitlecureur.com/fr/off-market)

---

Lire le playbook en ligne – [benoitlecureur.com/fr/playbook](https://benoitlecureur.com/fr/playbook)