EBOOK · PAR L'AUTEUR DES « 7 ERREURS À ÉVITER QUAND ON CODE AVEC L'IA »

L'IA code, tu pilotes

Faire coder l'IA sans perdre le contrôle — un système de travail complet, du premier prompt à la mise en ligne.

> CLAUDE.MD · RÈGLES DURES
>
> **NE JAMAIS** supprimer un test pour faire passer une suite au vert.
>
> **NE JAMAIS** réinitialiser la base pour résoudre un conflit de migration.
> (Cicatrice du 12/03 : 3 jours de données de test perdues.)
>
> **NE JAMAIS** logger l'e-mail d'un client — logger son id.

EXTRAIT GRATUIT · l'introduction et le chapitre 2 en entier

Le livre complet : 103 pages, 10 chapitres, 7 prompts d'installation

Gaëtan Cottrez · apprendre-la-programmation.net · édition 2026

INTRODUCTION

Change de siège

> *Ce livre commence là où le guide gratuit s'est arrêté. Les « 7 erreurs » t'ont appris à ne pas te crasher. Celui-ci t'apprend à voler.*

Il y a une différence énorme entre éviter les erreurs et avoir une méthode. Éviter les erreurs, c'est de la défense : tu connais les pièges, tu les contournes, tu survis. Une méthode, c'est de l'attaque : tu sais exactement quoi faire, dans quel ordre, avec quels outils — et tu le fais pareil un lundi matin en forme qu'un vendredi soir cramé. C'est ce qui sépare celui qui « se débrouille avec l'IA » de celui qui livre avec l'IA, semaine après semaine, sans que son projet lui échappe.

Je code depuis plus de quinze ans. Aujourd'hui, l'IA écrit une grande partie de mon code — Claude Code tourne sur mes projets comme un développeur de plus dans l'équipe. Et voilà ce que personne ne dit assez fort : **déléguer le clavier ne délègue rien d'autre.** Ni la compréhension, ni la décision, ni la responsabilité. Le jour où tu délègues les trois, tu n'es plus développeur : tu es spectateur d'un projet qui porte ton nom.

Ce livre est un système. Neuf chapitres de méthode — chacun un étage de la fusée, dans l'ordre où tu les mettras en place — plus un chapitre de terrain :

- Les chapitres 1 à 3 posent les fondations : le **contrat** entre l'IA et toi, le **cockpit** qui la configure pour TON projet, et la **base de connaissances** — la mémoire longue que vous partagez.
- Les chapitres 4 à 6 sont le **cycle de travail** : énoncé, plan, review. C'est le cœur, celui que tu répéteras dix fois par jour.
- Les chapitres 7 à 9 sont l'**industrialisation** : tes automatisations maison, tes tests, ta sécurité — ce qui tient encore debout quand tu es fatigué.
- Le chapitre 10, ce sont **trois reprises en main réelles** : la méthode appliquée à mes propres projets, chiffres et commits à l'appui.

Et tu n'es pas obligé de tout poser à la main. Chaque outil du livre a son encadré « 🔧 Installation pas à pas » — et, à la fin, l'annexe D te donne le prompt qui le pose à ta place, sur ton projet : tu le copies, tu le colles dans une session, tu réponds à cinq questions, c'est installé. Les deux chemins mènent au même endroit ; le premier t'apprend le système, le second te le monte.

Chacun des neuf chapitres de méthode se termine par une mission. Pas de la théorie à méditer : un geste à faire sur TON projet, le soir même. Si tu fais les neuf missions, à la fin de ce livre tu n'auras pas « lu un livre sur l'IA » — tu auras un système en état de marche.

Et pour que rien ne reste théorique, un projet fil rouge nous accompagne de bout en bout : **Factura**, un SaaS de facturation pour indépendants — API NestJS, front React, PostgreSQL, CI GitHub Actions. Un projet réaliste, avec du métier qui pique (numérotation légale, TVA, relances d'impayés). Chacun des

neuf chapitres de méthode se termine par « le résultat sur Factura » : pas la théorie de ce qu'il faudrait obtenir — le fichier, le plan, le rapport, assemblés et complets. Ton projet sera différent dans le détail ; la structure de ce que tu dois obtenir sera exactement celle-là.

Un dernier mot avant d'embarquer. Les outils cités ici — Claude Code en tête, parce que c'est celui qui a changé ma façon de travailler — auront changé de version avant la fin de l'année. Aucune importance : ce livre enseigne des mécanismes, pas des menus. Les fiches de poste, les plans verrouillés, les reviews par blocs, les listes de cas méchants — tout ça marchait avant l'IA, tout ça marchera avec la prochaine génération d'outils. C'est même la définition d'une méthode : ce qui reste quand les outils changent.

Allez. Change de siège.

CHAPITRE 2

Le cockpit

> *La même demande, le même outil, deux résultats sans rapport : la différence tient dans un fichier. Ce chapitre te fait écrire sa première version ce soir — puis t'ouvre un cockpit de production réel, pour que tu saches à quoi ressemble la cible.*

L'expérience des deux sessions

Fais l'expérience sur ton projet, elle prend cinq minutes. Demande quelque chose de banal — « ajoute un export CSV des factures ». Sur un projet non configuré, tu obtiens du code techniquement correct et étranger de la première à la dernière ligne : une bibliothèque ajoutée sans te demander, la logique posée dans un dossier `utils/` inventé alors que chez toi ça vit dans un service, des erreurs gérées d'une quatrième façon différente des trois déjà présentes dans ton code. Rien de faux. Rien de chez toi.

L'explication tient en une phrase, la plus importante du chapitre : **ton assistant repart de zéro à chaque session.** Ni tes conventions, ni ton architecture, ni les décisions de mardi dernier, ni les leçons du bug de la semaine passée. Tu as deux options : tout réexpliquer chaque matin, ou l'écrire une fois dans le fichier qu'il relit au démarrage de chaque session. Chez Claude Code, ce fichier s'appelle `CLAUDE.md` ; les autres outils ont leur équivalent. Tout ce que tu y écris, tu n'auras plus jamais à le répéter.

Le fichier du premier soir

La première version tient sur une page et se construit en trente minutes : la face IA du contrat en tête (chapitre 1), puis ton projet en cinq lignes, tes conventions, tes interdits, ton process, ton vocabulaire métier — le modèle complet est en annexe A. Cette version-là change déjà tes sessions du jour au lendemain : le code sort rangé au bon endroit, nommé comme chez toi.

Mais je vais être honnête avec toi : cette page n'est que le premier soir. Si je te laissais croire que le cockpit s'arrête là, je te mentirais par omission — et tu plafonnerais vite. Alors ouvrons un vrai cockpit.

Visite d'un cockpit de production

Ce qui suit est tiré des fichiers de deux plateformes en production que je pilote — dont une certifiée dans un domaine réglementaire où l'erreur ne pardonne pas — avec des agents IA qui y travaillent tous les jours. Anonymisé, mais pas édulcoré. Un cockpit mûr contient cinq choses qu'aucun tutoriel ne montre :

1 · Une carte, pas une encyclopédie. Le fichier n'essaie pas de tout contenir — il *route*. La première section est une table : « quand la question touche l'architecture → lis `docs/ARCHITECTURE.md` ; la base

de données → `docs/DATABASE.md` ; les conventions → `docs/CONVENTIONS.md` »… avec la règle qui donne tout son sens à la carte :

```
## Carte documentaire
- Ouvre le doc AVANT de deviner. Ne reconstruis jamais
  une réponse de mémoire ou en fouillant le code quand
  un doc canonique couvre le sujet.
- Cite le doc sur lequel tu t'appuies, que je puisse
  vérifier.
- Si le doc contredit le code : signale-le et propose
  une correction. Ne contourne jamais en silence.
- Ton changement rend un doc obsolète ? Tu mets à jour
  le doc dans la même livraison.
```

Retiens le principe : le cockpit d'un vrai projet ne remplace pas la documentation, il rend son usage *obligatoire*. « Ouvre le doc avant de deviner » est probablement la ligne qui économise le plus d'erreurs par mois.

2 · Un protocole de comportement. Pas seulement quoi faire — comment se comporter. Extrait :

```
## Protocole
- Avant d'agir : dis « ce que j'ai compris » puis
  « ce que je compte faire », et attends mon accord.
- Ne jamais rien supposer ni inventer : cherche, lis,
  vérifie. Un nom de branche, un chemin, une config —
  ça se vérifie, ça ne se devine pas.
- Production uniquement : pas de code d'exemple, pas
  de placeholder.
```

C'est la face IA du contrat du chapitre 1, poussée au niveau opérationnel : l'IA annonce, tu arbitres. Sur un petit projet tu allégeras ; l'important est que le *mode de collaboration* soit écrit, pas improvisé à chaque session.

3 · Des règles dures — et chacune est une cicatrice. Le cœur d'un cockpit mûr. Regarde bien la forme de ces extraits (réels, anonymisés) :

```
## Règles dures
- NE JAMAIS supprimer un fichier de test pour faire
  passer une suite au vert. Réécrire les fixtures,
  jamais la couverture.
- NE JAMAIS utiliser de mot-clé auto-fermant
  (« Closes #X ») dans une MR : ça court-circuite
  l'étape QA de notre board.
- NE JAMAIS logger de données client. Le test n'est pas
  « est-ce un secret ? » mais « serais-je à l'aise que
  toute personne ayant accès aux logs lise cette
  ligne ? ». (Leçon du 27/07 : trois endroits du code
  publiaient des données métier dans les logs.)
- Tout fichier temporaire vit dans .scratch/ gitignoré,
  JAMAIS dans /tmp. (Leçon de juin : des preuves de test
  écrites hors du repo étaient invisibles à l'audit.)
- NE JAMAIS réinitialiser la base de dev pour résoudre
  un conflit de migration. La migration s'écrit pour
  s'appliquer sur la base EXISTANTE.
```

(Si le sigle t'échappe : une **MR**, *merge request*, est la demande de fusion d'une branche — la « pull request » chez GitHub.) Tu vois le motif ? **Chaque règle dure est un incident qui ne se**

reproduira plus. La date, le contexte, le pourquoi — écrits à côté de l'interdit. C'est ça qui rend la règle inviolable pour l'IA *et* compréhensible pour l'humain qui la relit dans six mois. Un cockpit mûr se lit comme le journal des batailles du projet : chaque ligne a coûté quelque chose, une fois, et plus jamais.

Une nuance qui t'évitera de polluer ton fichier : toutes les règles dures ne naissent pas d'un incident. Certaines viennent d'un fait vérifiable de ton dépôt — un script de seed qui vide une table avant de la réécrire, un dossier de fichiers générés qu'un build écrase à chaque passage, une commande qui détruit tes volumes Docker. Celles-là sont tout aussi dures, mais elles portent leur *source* — le fichier et la ligne qui le prouvent — au lieu d'une date. Ne leur invente pas de cicatrice : dans six mois, tu chercherais un incident qui n'a jamais eu lieu.

4 · Les invariants indevinables. La catégorie la plus précieuse : ce que ni le code ni la doc ne peuvent révéler. Exemple réel anonymisé : un serveur d'échange partenaire qui *supprime chaque fichier à la première lecture* — comportement côté serveur, strictement invisible dans notre code. Une IA qui fouille le repo « prouvera » avec assurance que les fichiers restent disponibles. Faux, et destructeur. Une ligne dans le cockpit — « le serveur X est destructif à la lecture : premier lecteur gagnant, aucune relecture possible » — et cette erreur devient impossible. Recense les tiens : le webhook qui rejoue, l'API dont le statut « accepté » ne veut pas dire « en vigueur », le sandbox qui ne se comporte pas comme la prod.

5 · Une définition de « fini ». La dernière section verrouille le mot le plus dangereux du travail avec l'IA :

```
## Definition of done
Une tâche n'est finie que quand le comportement demandé
fonctionne de bout en bout — PAS quand ça compile, pas
quand un commit passe, pas quand un test unitaire isolé
est vert. Interdit de rétrécir le périmètre en silence.
« J'ai commité » n'a jamais voulu dire « ça marche ».
```

Chez moi, cette section va un cran plus loin : « fini » exige une **preuve d'exécution réelle** de bout en bout — le comportement observé sur le système qui tourne, jamais un test mocké qui se fait passer pour lui — archivée avec la livraison. Le chapitre 8 te dira ce qui compte comme preuve, et surtout ce qui n'en est pas une ; la forme, elle, dépend de ton système — chez moi, un fichier HTML horodaté avec les entrées injectées, les logs bruts et l'état final observé en base.

Sans cette section, l'IA optimise pour te dire « c'est fait » le plus vite possible. Avec elle, « fait » a une définition — la tienne.

Et quand le fichier déborde : les règles par dossier. Un cockpit de production reste court parce qu'il ne porte pas tout : les règles spécialisées vivent dans des fichiers à part, chargés automatiquement quand l'IA touche la zone concernée. Sur mes projets : une règle « couche application » qui ne se charge que quand on édite un use-case, une règle « tests » qui ne se charge que sur les fichiers de test, une règle « workflows » pour les modules asynchrones. Chez Claude Code, c'est le mécanisme des règles par chemin (`.claude/rules/`) ; peu importe l'outil, retiens l'architecture : **le cockpit est l'index per-**

manent, les règles pointues se chargent à la demande — le contexte reste léger, la précision reste chirurgicale.

La voilà, généricière pour ton projet — le mécanisme tient dans l'en-tête `paths` : ces globs déclenchent le chargement automatique quand l'IA touche un fichier correspondant. Voici `.claude/rules/tests.md` — et regarde bien le premier caractère : l'en-tête ouvre le fichier, sans ligne de titre au-dessus. Une seule ligne avant le `---` et ce n'est plus un en-tête mais du texte : la règle se charge alors en permanence, au lieu de se charger par chemin :

```
---
paths:
  - "**/*.spec.ts"
  - "**/tests/**"
---
# Règles — tests
- Un test de service se monte avec des repositories
  mockés — jamais de vraie base, jamais de container.
- Asserter sur le résultat rendu (succès/échec + la
  valeur), pas sur les détails internes.
- Tout nouveau chemin scopé par compte a son cas
  d'isolation.
- JAMAIS modifier un test pour le faire passer —
  un test rouge, c'est le code qu'on répare.
```

Où : ce fichier vit dans `.claude/rules/tests.md` (crée le dossier `.claude/rules/` s'il n'existe pas). Rien d'autre à brancher : l'en-tête `paths` suffit, la règle se charge toute seule dès que l'IA touche un fichier qui matche. Vérification : demande une modification dans un fichier de test et observe qu'elle applique ces règles sans que tu les rappelles.

De la page du premier soir au cockpit de production

Entre les deux, il n'y a pas un grand soir de rédaction — il y a une discipline d'accrétion, deux règles :

- **La règle des deux fois** : la deuxième fois que tu donnes la même consigne, elle monte dans le fichier. Pas la cinquième. La deuxième.
- **La règle de la cicatrice** : chaque incident — le test supprimé en douce, le fichier perdu, la donnée loggée — devient une règle dure, datée, avec son pourquoi. Le jour même, tant que ça fait encore mal.

C'est tout. Un cockpit de production n'est pas écrit par quelqu'un de plus malin que toi : il est écrit par quelqu'un qui a noté ses batailles au lieu de les revivre. Dernier garde-fou pour la route : chaque ligne doit corriger un comportement concret. « Écris du code propre » ne corrige rien — supprime. La ligne dont tu ne peux pas citer l'incident ou le comportement qu'elle corrige n'a rien à faire dans le fichier.

Le résultat sur Factura : le cockpit complet

Assez de morceaux — voilà le fichier assemblé, du premier au dernier caractère. C'est un cockpit réaliste pour un projet de cette taille : quelques mois de vie, deux cicatrices déjà, la structure complète. Le tien différera dans chaque détail ; il doit ressembler à ça dans chaque section.

```
# CLAUDE.md — Factura
# SaaS de facturation pour indépendants
# API NestJS (src/), front React (web/), PostgreSQL, CI GitHub Actions

## Le duo — ta place
- Tu proposes, c'est moi qui décide : aucune décision
  d'architecture, de dépendance ou de périmètre sans me
  la soumettre.
- Si tu dois t'écarter de ce qu'on a validé, arrête-toi
  et dis-le avant de continuer.
- Ne me rassure pas par défaut : signale les risques,
  dis « non vérifié » quand tu ne peux pas sourcer.

## Carte documentaire — ouvre le doc avant de deviner
- Architecture, couches, règles de dépendance
  → docs/ARCHITECTURE.md
- Décisions structurantes et leurs pourquoi
  → docs/architecture/ (ADR numérotés)
- Schéma de base, migrations → docs/DATABASE.md
Cite le doc que tu utilises. S'il contredit le code :
signale-le, ne contourne jamais en silence. Ton
changement rend un doc obsolète ? Il est mis à jour
dans la même livraison.

## Protocole
- Avant d'agir : « ce que j'ai compris », « ce que je
  compte faire », puis attends mon accord.
- Rien ne se devine : un chemin, une branche, une config
  se vérifient dans le repo.
- Production uniquement : pas d'exemple, pas de
  placeholder.

## Le projet
Factura permet à des indépendants de créer, envoyer et
suivre leurs factures, et de relancer les impayés.
Le métier vit dans src/services/ (un service par
domaine : invoice, client, payment, reminder), l'API
dans src/modules/, les tests Jest à côté des fichiers
qu'ils testent. Le front (web/) ne parle qu'à l'API.

## Conventions
- Code et commits en anglais, commentaires en français.
- Pas de dossier utils/ fourre-tout : tout code métier
  a un domaine.
- Toute erreur passe par AppError — jamais de throw nu.
- Front : composants fonction + hooks, pas de classes.
- Argent : entiers en CENTIMES partout. Jamais de float.
  Conversion à l'affichage uniquement.

## Interdits
- Ne jamais ajouter une dépendance sans me demander.
- Ne jamais toucher à src/migrations/ sans plan validé.
- Ne jamais écrire de secret (clé, token, mot de passe)
  dans un fichier versionné, même en exemple.
- Ne jamais « corriger » un test pour le faire passer.
- Ne JAMAIS réinitialiser la base de dev.
  (Cicatrice du 12/03 : 3 jours de données de test
  perdues pour « résoudre » un conflit de migration.)
- Ne jamais logger l'e-mail ou l'IBAN d'un client —
  logger son id. (Cicatrice du 02/04 : adresses en
  clair dans les logs de relance.)

## Process
- Toute feature part d'un ticket complet (restitution
  validée en description, décisions en commentaires),
  puis un plan validé (contrat de session), puis le
  code par blocs de ~30 lignes.
- Tests : npm test · Lint : npm run lint — les deux
  verts avant tout commit.

## Vocabulaire
- « Brouillon » : facture non numérotée, modifiable.
```

```
- « Émise » : facture numérotée, IMMUABLE légalement.
  Toute correction passe par un avoir, jamais par une
  modification. Numérotation continue par année, sans
  trou (FAC-2026-0001) — voir docs/architecture/ADR-0002.
- « Relance » : e-mail d'impayé, 3 niveaux (J+7, J+21,
  J+45 mise en demeure). Jamais sur facture payée.

## Invariants indevinables
- La sandbox du prestataire e-mail accepte tout et
  n'envoie RIEN : un test « vert » ne prouve pas l'envoi.
- L'API des taux de TVA répond 200 avec un corps VIDE
  hors UE : tester le contenu, jamais le statut.

## Definition of done
Une tâche est finie quand le comportement demandé
fonctionne de bout en bout — pas quand ça compile, pas
quand un commit passe, pas quand un test isolé est
vert. Interdit de rétrécir le périmètre en silence.
« J'ai commité » n'a jamais voulu dire « ça marche ».
```

🔧 Installation pas à pas

1. À la **racine de ton projet** (le dossier qui contient ton `.git`), crée un fichier nommé exactement `CLAUDE.md`. Aucune commande spéciale : c'est un fichier texte, crée-le depuis ton éditeur — ou demande à ton IA : « crée CLAUDE.md à la racine avec ce contenu ».
2. Colle ton contenu (pars de Factura ci-dessus ou du modèle de l'annexe A, adapte chaque section à TON projet).
3. **Vérifie qu'il est lu** : ouvre une NOUVELLE session (le fichier est chargé au démarrage), et demande : « quelles règles t'ai-je données dans le CLAUDE.md ? ». Elle doit te les réciter. Puis fais le vrai test : demande une petite feature et vérifie qu'elle respecte tes conventions.

Les autres outils que ce livre te donne (skills, hooks, agents) s'installent dans le dossier `.claude/` à la même racine — chacun aura son pas-à-pas au moment où tu le rencontres.

🎯 Ta mission (30 minutes ce soir, puis en continu)

Ce soir : l'expérience des deux sessions sur TON projet — session témoin, puis écris la page du premier soir (annexe A), puis même demande et compare. Cette semaine : tes deux premières cicatrices — repense aux deux dernières fois où l'IA t'a fait un coup tordu, et écris les deux règles dures correspondantes, datées, avec leur pourquoi. Ton cockpit vient de commencer sa vraie vie.

> **Ce qu'il faut retenir**
>
> - L'IA repart de zéro à chaque session : le cockpit est sa mémoire — et la tienne.
> - La page du premier soir suffit pour démarrer ; la cible, c'est le cockpit de production : carte documentaire, protocole, règles-cicatrices, invariants indevinables, definition of done.
> - Deux règles d'accrétion : la consigne répétée deux fois monte dans le fichier ; l'incident devient une règle datée le jour même.

LA SUITE

Ce que contient le livre complet

> *Tu viens d'installer ton cockpit. C'est la première pièce sur sept. Voilà les six autres, et ce que chacune t'évite.*

3 · La base de connaissances

Une documentation qui ne pourrit pas — parce qu'une machine l'en empêche. Une commande de synchronisation et deux hooks : dès qu'un commit de fonctionnalité atterrit, le rappel part ; et la session refuse de se terminer tant que la doc n'a pas suivi le code. Plus le format d'ADR qui fait survivre une décision — et sa cicatrice — aux sessions et aux agents.

4 et 5 · L'énoncé et le plan

Tu balances ton besoin comme il te vient, à 22 h, mal tapé : c'est elle qui rend la structure, toi qui corriges. Puis le livrable de plan en quatre blocs et le **checkpoint unique** — le seul moment où tu arbitres, en échange de quoi l'exécution s'enchaîne sans te redemander la permission toutes les trois minutes. Le skill complet est fourni.

6 · Reviewer, jamais greffier

Trois verdicts par bloc, les cinq odeurs du code généré, la review de branche notée sur 10 — et une boucle de correction bordée qui tourne en autonomie jusqu'à zéro problème bloquant, dans un espace de travail isolé, avec le merge final qui te reste.

7 · Tes skills, tes hooks, tes agents

L'escalier de l'industrialisation : la consigne répétée devient un skill, la règle violée deux fois devient un hook, le risque n° 1 devient un reviewer spécialisé. Avec le hook bloqueur complet — 230 lignes de production, chaque bloc payé par un incident — et son harnais de tests.

8 · Casser avant les autres

Le piège des tests générés, et la règle qui l'évite : la liste vient de toi, le code vient d'elle, le verdict revient à toi. Les cinq familles de cas méchants, et la CI qui rejoue tout à chaque push, scan de secrets compris.

9 · La sécurité en continu

L'audit devenu une étape du cycle plutôt qu'un événement annuel, la règle du coffre, l'interrogatoire des dépendances — et le gate de livraison qui **refuse** de pousser au lieu de se contenter de signaler.

10 · Trois reprises en main

Trois de mes projets repris en main pour de vrai, commits et chiffres à l'appui : ce qui a été gardé, ce qui a été jeté, ce que ça a coûté, et ce que je referais autrement.

Et les quatre annexes

Ton CLAUDE.md type, les dix prompts du pilote, les checklists — et surtout les **sept prompts d'installation** : un par chapitre, à coller dans une session pour que l'IA pose le système sur ton propre projet, avec tes vraies commandes et tes vrais fichiers.

> **Le livre complet**
>
> 103 pages, dix chapitres, quatre annexes, et les sept prompts livrés aussi en fichiers texte.
>
> **apprendre-la-programmation.net/l-ia-code-tu-pilotes**