

THE PLAYBOOK

The AI Product Builder's Playbook

AI didn't remove the process. It made it
mandatory.

Benoit Lecureur

benoitlecureur.com

Contents

00	The role: the AI Product Builder	3
01	The laws of building	5
02	The dev foundations (plain-language)	10
03	CI/CD and mechanical checks	14
04	The work cycles	16
05	Specs and scoping (the core craft)	19
06	Tests and QA: proof against intention	23
07	Data, migrations, and money	25
08	Shipping, security, and operations	27
09	Documentation as infrastructure	31
10	Design system, content, coherence	33
11	Building with AI: what actually changes	35
12	The catalogue of traps	38
13	Going further (what I haven't done yet, and what dev+++ looks like)	39
14	Glossary	42
15	The self-assessment	48
16	Case notes (anonymized)	50

The role: the AI Product Builder

What it is

The AI Product Builder owns the complete loop: from the idea, to scoping, to the spec, to the build (executed by AI), to verification, to deployment, to tests and QA, to telling the world it shipped. That used to be four jobs: a PM who specs, developers who code, QA who tests, ops who deploys. One person can now hold all of it, because AI gives them the lever to execute what they could not have coded alone.



The shift

You don't need to be a career developer. You need three things.

1. Understand the concepts (that's what this book is for), so you speak the language, make good calls, and notice when something is off.
2. Orchestrate the AI: frame it, give it context, break the work down, read what it produces.
3. Hold the quality line: the product decisions and the trade-offs stay yours, verification is non-negotiable, you are the safety rail.

One sentence to remember: the decisions are yours, the execution is delegated, the verification is mandatory.

What the role reintegrates

Scoping (turn a fuzzy want into a clear, sliced, prioritized scope). Spec (write the design doc that fixes the goal, the context, the decisions, the data model, the slices, the tests). Build (have AI execute from framed prompts).

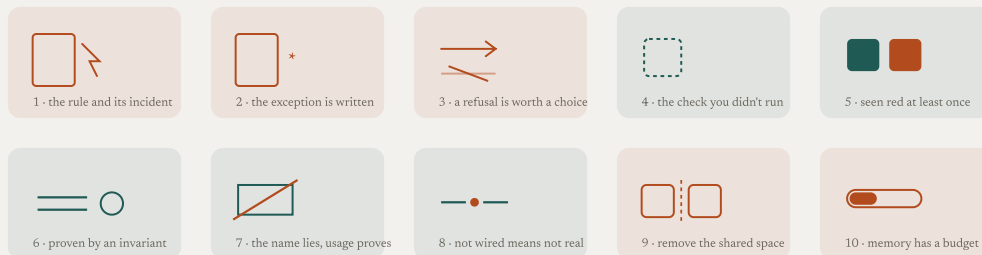
Verification (re-read, build, test in a real browser, validate each step).

Deployment (put it live, and know what each move triggers). Tests and QA (run checklists, validate end-to-end journeys, handle edge cases).

Communication (present the release: what shipped, why, a demo, what's next).

You are not smaller than a team. You are the conductor of one.

The laws of building



One pictogram per law — the ten, at a glance.

These are the principles that survive even if you remove every specific project from the story. They are the spine of everything that follows. Each one was learned the expensive way.

Law 1. A rule is written with its incident

A rule with no story gets bypassed. A rule that carries the date, the cost, and the failure that created it does not, because the reader can see exactly what they're buying by ignoring it.

So when you write a rule, write the failure next to it. And the test: if you can't find a failure behind your rule, it's probably a preference, not a rule. Preferences are fine. Just don't defend them like laws.

Law 2. An exception is documented like a rule

The normal reflex is to tolerate an exception quietly, which is exactly how exceptions multiply. Do the opposite. An exception is written with four things: the general rule, restated and explicitly protected; the cumulative criteria that allow the exception; the named list of every instance that currently uses it; and the condition under which it returns to the general rule.

"If a single criterion drops, this goes back under the general rule." That one line keeps an exception from quietly becoming the new norm.

Law 3. A rejected decision is worth as much as a chosen one

Keep a "rejected alternative" next to every real decision. It's what stops someone, six months later, from re-proposing an option you already killed, with the same arguments, because they weren't in the room the first time.

Corollary: an obsolete plan is not deleted. It's struck through and dated, with a pointer to the living decision. Deleting it just means you'll rebuild it from scratch and rediscover why it was wrong.

Law 4. A verification you didn't run is worse than none

This is the most expensive anti-pattern there is. Picture a note that says "checked: returns the right result," presented as proof, where the check was never actually run. It doesn't just fail to protect you. It closes the subject, so nobody looks again. On one project, a "verification" that was written but never executed let a batch of internal files sit exposed in public for months.

So the correct shape of a fix is never "check that it works." It's the exact command to run, and the regression criterion: the precise signal that would tell you the bug is back.

Law 5. A green gate is worth nothing until you've seen it go red

Before you trust any automatic check on new ground, make it fail on purpose. If it doesn't go red when you break the thing, it will never protect you, it will just reassure you.

There are many ways an automatic check quietly lies: it doesn't scan the area you care about; it only sees files already saved to version control, so it says "all clear" on brand-new code; its search pattern misses the real shape of the code; the target sits outside its scope; the runner counts "found nothing to

do" as a pass. Real example of the last trap: a test suite existed, over a hundred assertions, and it was wired to run nowhere. It passed by never running.

The discipline: calibrate every lock against one known-good case and one known-bad case frozen as a test. Without that, nothing tells "working" apart from "inert." And remember the human side: a lock that cries wolf gets switched off.

Law 6. A mass change is proven by an invariant, never by a re-read

When you change a thousand things at once, you cannot eyeball it. You prove it with an invariant, a fact that must hold. Three that work in practice.

Set equality: after replacing something across N pages, check that the count carrying the new version equals the count that carried the old one, and that zero of the old remain. A half-finished bump desyncs everything while looking fine.

Fingerprint the lines that must not move: before and after a big migration, prove that the specific lines you weren't supposed to touch are byte-for-byte identical. You don't just avoid touching them, you prove you didn't.

The quantified delta in production: not "the test is green," but "these exact numbers moved from X to Y on the real data, and zero items ended up misclassified." A delta beats a green checkmark.

Law 7. The name lies, the proof is the usage

Names drift from meaning. On one app, an object called "Company" actually meant a parent group, not a single site. A relationship built on the resemblance of names, rather than the reality of the data, rejected nearly eight rows out of ten. The names agreed. The data didn't.

The hardest and most useful extension: this applies to your own documents too. A doc you wrote yourself is not a source of truth, it's a dated hypothesis. Verify it against the real thing before you trust it. Two corollaries: disabled code is often richer than the live code (it tells you what was tried), and a component named in a comment is a component worth opening.

Law 8. What isn't wired to a mechanical check doesn't exist

And its twin: what isn't written in a file doesn't exist. A conversation is not a memory system.

The proof is boring and universal: of five points raised out loud in a review, four got written down somewhere, one didn't. The one that vanished was a legal line shown to users at the exact moment of their consent, and it had quietly become false. Spoken agreements evaporate. If it matters, it lives in a file or in an automatic check, or it does not really exist.

Law 9. The real fix is to remove the shared space

When two people (or two AI sessions) working in parallel keep colliding over the same shared resource, the instinct is to add a better check. Often the better move is to remove the shared thing entirely.

The classic case: parallel work kept grabbing "the next number" for a change, without seeing each other, and collided again and again. No pre-check could work, because looking at "the last number used" looks at a world where the sibling change doesn't exist yet. The fix wasn't a smarter check. It was switching to a timestamp, so there's no shared counter left to consult. Two timestamps don't collide. When a lock can't hold a shared resource, ask whether the resource should be shared at all.

Law 10. Memory has a budget and tends toward empty

Your project memory (the file that keeps the context an AI or a teammate needs) has a budget. Past it, you don't add, you distill. A good memory file is designed to shrink over time, not grow: each entry is distilled or deleted when its chapter closes.

The deeper version: a lesson written six times, each time as if it were new, is not six lessons. It's one law that never had a home. When you catch yourself relearning the same thing, don't add a seventh note. Give the law a home, and every future occurrence just points there.

The dev foundations (plain-language)

You don't need to write any of this. You need to understand it well enough to make good calls and smell trouble. Every term here is in the glossary too.

2.1 Version control, and why "not pushed" means "does not exist yet"

Version control records the history of your code: who changed what, when, and lets you go back. Its online home is the reference copy, the backup, and the meeting point.

The words you'll actually use: a repository is the versioned project. A commit is a dated snapshot of a set of changes, with a message. A branch is a parallel line of work; the main line is usually called "main" and is your production. Push and pull send your local commits up and pull others' down. Merge folds one branch into another. A pull request is a request to merge, usually reviewed first.

The lesson paid in time: a commit that isn't pushed lives only on your machine. A re-clone or a local incident erases it. The rule: push after every unit of work. Local is fragile, the push is the backup. I've lost unpushed work twice. Recovered both times, but those are hours you don't get back.

For the CTO in you

when several accounts share one machine, a push can fail with "repository not found" simply because the wrong account is active. Switch accounts before you push.

2.2 Branches and review

The classic model: main is production, sacred, always deployable. You work on a branch, verify, then merge into main. Some teams keep a long "preprod"

branch: push there, check the preview URL, then merge to main. Code review is the moment a peer re-reads before merge. With AI, review becomes two things at once: you re-read what the AI produced, and the AI does a review pass of its own. Two readers beat one.

2.3 Environments, config, and secrets

A product lives in several environments: local (your machine, to build), preprod or staging (an online copy to validate before prod), and production (what real users see). Environment variables configure the product per environment, like an API address or a public key. Two families matter. Non-sensitive config can live in the code. Secrets (API keys, powerful access keys, payment secrets) never go into version control. They're set by hand in your host's dashboard and persist across deploys.

The golden rule, worth tattooing: the powerful key, the one with full access to your database, is never in the frontend. Only server-side. The frontend gets a public, restricted key, fenced in by database rules. If a full-access key ever reaches the browser, assume the database is open to the world.

Keep an environment-variable matrix: a simple table of every variable, where it lives (in code, in a secret store, in the host dashboard, in your local file), and who sets it. Missing variables get discovered at deploy time, in the worst mood, over and over, unless you keep the map.

2.4 The backend building blocks

You won't build these from memory. You'll recognize them, and know which ones a feature touches, so you can ask the right questions.

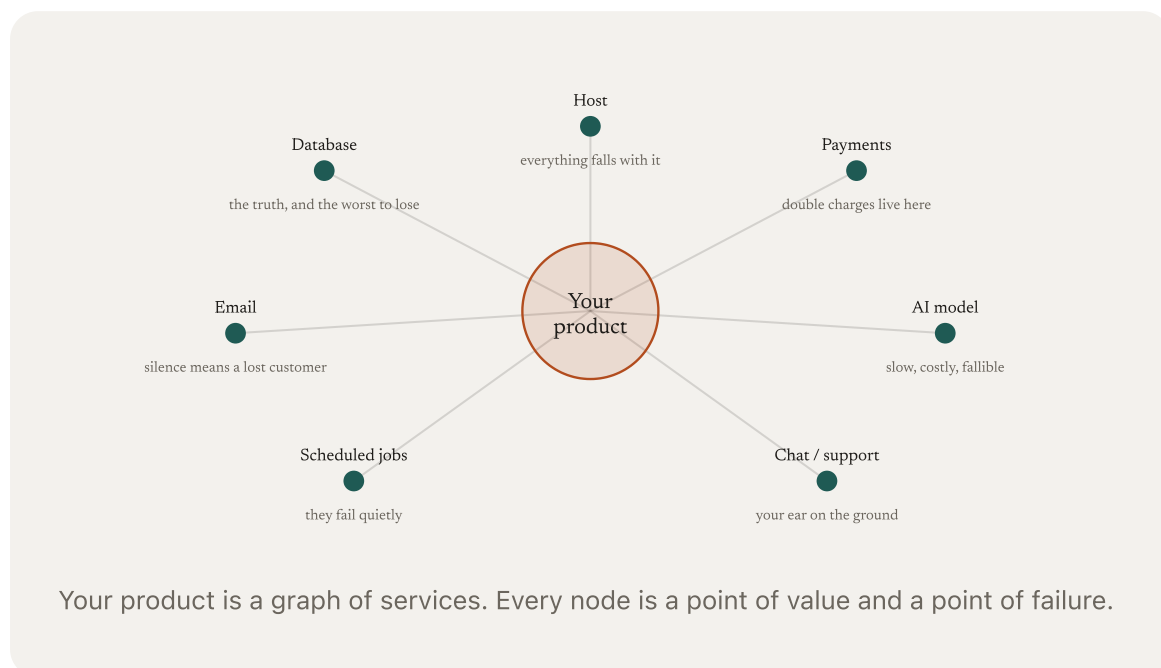
An API is the interface the frontend, or a third party, uses to talk to the backend; a request lands on an endpoint, a URL plus a method. Auth is proving who you are, usually with a signed token sent on each call and checked by the server. A webhook is a call a third party makes to you when

something happens, like a payment provider announcing "this invoice is paid." Rate limiting caps requests per period to stop abuse. Idempotence makes sure the same operation played twice doesn't double its effect, which is critical for payments and replayed webhooks. Encryption protects sensitive data at rest, like stored keys locked with a server-side secret.

2.5 Dependencies are power and risk

Every package you install is leverage you didn't have to build, and a door you now have to trust. Two habits keep it sane. Lock your versions: a lockfile pins the exact version of everything, so your build is identical today and in six months, on your machine and in production. And resist the blind auto-fix: a command that "fixes" every security warning by force-upgrading everything will happily break your app to silence a warning that never applied to you. Read what a fix changes before you run it. Fewer dependencies, chosen on purpose, beat a pile you can't account for.

2.6 Your product is a graph of services



A modern product is not one thing, it's a graph: a database, a payment provider, an email sender, an AI model, a chat integration, a host, a job

runner. Each node is a point of value and a point of failure. Draw the map, even roughly, and for each node ask two questions: what happens to my users if this is down, and what does this one cost me. You don't have to make each one unbreakable. You have to know where the single points of failure are, so an outage is a known risk and not a surprise.

CI/CD and mechanical checks

3.1 The words, first

CI, continuous integration: on every change, automatic checks run (build, tests, lint) to catch regressions early. CD, continuous deployment or delivery: once verified, code flows to preprod then prod. A pipeline is that chain of steps. A gate is any automatic check that can block the line.

3.2 What automation costs, and what it guarantees

Automation is not free. Every check costs time to run, money in compute, and attention (a noisy check gets ignored). So you choose a regime, not "all the checks." A lean regime runs the cheap, high-value checks on every change (does it build, does it type-check, does a fast style pass) and saves the expensive ones (full end-to-end, load tests) for the moments that matter.

Honest note, because this book doesn't oversell: on my own products, a full automated test suite in the pipeline is often not wired in the first version. It's a deliberate early trade-off, added after launch. What does run automatically: the site redeploys on push, and the build must pass or nothing ships. The backend deploys by hand, on purpose. That mix, automatic on the site and manual on the backend, is a stable choice, not an unfinished chore. Say what you do and what you don't.

The honest version of a trade-off has two halves: name it as a deliberate choice, and name the trigger that flips it. "No automated test suite in the pipeline yet, because early on manual QA over a small surface is faster; we add the suite the moment the surface grows past what one person can check by hand, or the first regression reaches a user." A trade-off with a trigger is a plan. A trade-off without one is a hole you're pretending is a plan.

3.3 Blocking vs informational

Not every check should block. Some stop the line: the build fails, nothing ships. Some only inform: a warning you read and decide on. Know which is which, because a check you think is protecting you might only be whispering.

3.4 The ways a gate lies, and how to fix it

This is Law 5 made practical. A gate can pass while protecting nothing. It scans the wrong area. It only sees files already saved to version control, so brand-new code reads as "all clear." Its search pattern misses the real shape of the code. The runner counts "nothing to do" as a success. A test suite exists but is wired to run nowhere, so it passes by never running.

The fix is a discipline. Before you trust a check on new ground, make it fail on purpose. Then freeze one known-good case and one known-bad case as tests, so the check can always tell "working" apart from "inert." And keep it quiet: a gate that cries wolf gets switched off by the very people it was meant to protect.

The work cycles

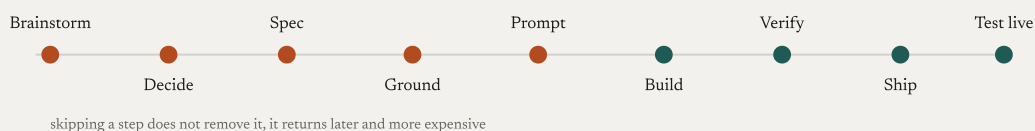
4.1 A project is not an increment

There are two sizes of work, and confusing them causes most of the mess. An increment is a small, self-contained change. A project is a larger effort with its own intent, scope, and spec. You open a project deliberately. You don't let a "quick change" quietly become a three-week rebuild without ever writing down what you're doing.

4.2 The intent spec, written before any code

Before building, write the design doc: the goal, the current state of the code, the decisions already made, the data model (new tables and columns), the slices, the list of tests, and what's explicitly deferred. It's the source of truth, and it lets anyone (another session, or an AI) pick up the thread. Writing it is not overhead. It's the cheapest place to be wrong.

4.3 The delivery chain, in order

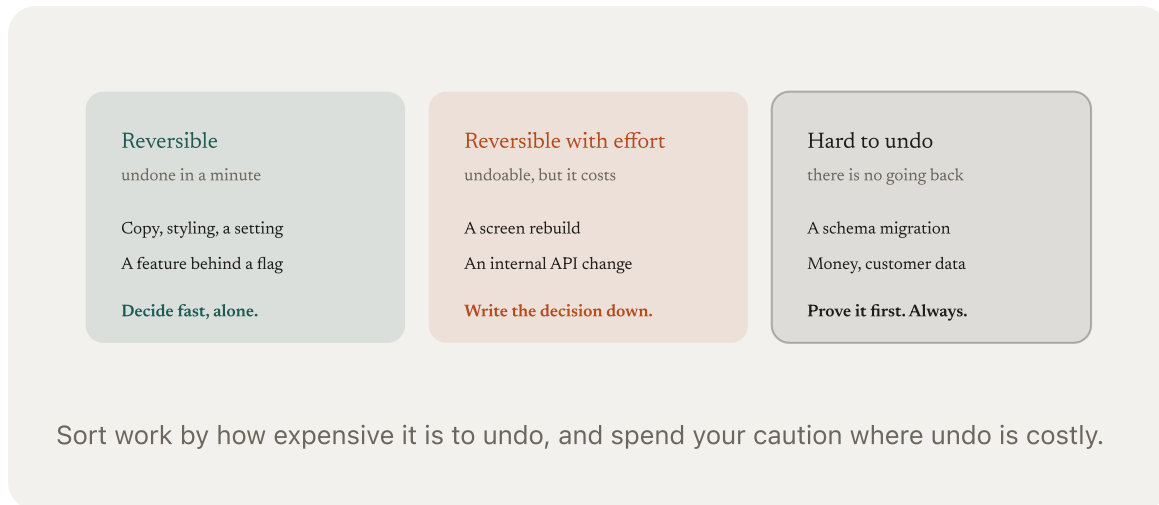


The chain runs in order. Skipping a link doesn't save time — it moves the cost downstream.

Brainstorm, then decide the real trade-offs one at a time, then spec, then ground (read the existing code so you build in the right place), then prompt the executor, then build, then verify (build, type-check, re-read, test in a real

browser), then commit, push, deploy, then test the slice. Skipping a link doesn't save time. It moves the cost downstream, where it's bigger.

4.4 Classify work by reversibility



The single most useful sorting question: if this goes wrong, how expensive is it to undo? Three verdicts. Cheaply reversible, like a visual tweak: move fast, fix forward. Reversible with effort, like a change to how data is shaped: verify harder, keep a rollback path. Hard to reverse, like moving money, a destructive data migration, or anything users can't un-see: slow down, prove it, test it for real. You don't apply the same caution everywhere. You spend it where undo is expensive.

4.5 Debugging as a method, not a vibe

When something breaks, don't poke at random. Work in phases: reproduce it reliably, isolate where it happens, form one hypothesis, test that one hypothesis, fix, then prove the fix with the exact signal the bug would show if it came back (Law 4). Random poking sometimes works and teaches you nothing. A method works and compounds.

4.6 Refactoring in waves

Don't rewrite everything at once. Change in small, provable waves. After a mass change, prove it with an invariant (Law 6), not a re-read. And never run two waves in parallel on the same area: they edit the same big files and step on each other.

Specs and scoping (the core craft)

5.1 What the builder actually holds

The build is delegated. The scope is not. Your real job is deciding what gets built, in what order, and what "done" means. Everything upstream of the code is yours: the intent, the slices, the trade-offs, the acceptance bar. Lose the scope and no amount of fast execution saves you.

5.2 The screen inventory, the best scoping tool there is



the happy path is the shortest of the four, and the only one we remember to draw

Every screen has four states. The happy path is only one of them.

Before building a feature, list every screen it touches, and for each screen, the four states it must handle: loading, empty, error, success. This one habit catches most of the scope you'd otherwise discover mid-build. A feature is not "the happy path." It's the happy path plus the three states everyone forgets. No blank screens, ever.

The deeper principle: design for the path that fails, not just the one that works. The empty state and the error state are not edge cases, they are where trust is won or lost. Most unfinished features are unfinished precisely there.

5.3 "Definition of done", in its three forms

"Done" means different things at different altitudes, and naming them prevents arguments. Done-coded: it builds, it type-checks, the change exists. Done-verified: it's been tested in a real browser, the four states work, the edge cases are handled. Done-shipped: it's in production and you've confirmed it live, the smoke test passed. A thing can be done-coded and nowhere near done-shipped. Always say which one you mean.

Definition of Done, the checklist. A slice is done when: it builds and type-checks; the four states work in a real browser; the edge cases and money paths are tested for real; it's committed and pushed; it's deployed and confirmed live (the smoke test passed); and the doc or changelog is updated. Anything short of that is "done-ish," which is another word for not done.

5.4 The go rule, and why it's literal

When you say "go," it means go: the decision is made, we don't re-debate. When something is marked delivered, it's delivered: we measure and iterate, we don't rebuild it. This sounds obvious and is constantly violated. A large share of lost time is re-litigating settled decisions. Write the decision down, with its rejected alternative (Law 3), and move.

5.5 How a request becomes real work

A request is not a task. "Can it also do X" is a fuzzy want. Turning it into work means clarifying the actual outcome wanted, sizing it (increment or project), finding where it lives in the code, deciding the trade-offs, then slicing it. The gap between "sure, easy" and the real work is where every bad estimate is born. Scope in writing, not in your head.

For the CTO in you

the distinction that prevents the most disputes is bug vs change. A bug is the product not doing what it was specified to do. A change is a new thing you now want. Different urgencies, different owners, different queues. Conflating them turns every "it should also do this" into a false emergency.

5.6 Walk the journey before you spec, and slice vertically

Before you write a spec, walk the whole user journey out loud, step by step: how does this person arrive, sign in, reach the thing, and what do they see at each step. A spec that nails the data model and the endpoints but skips "how do they actually log in" ships a feature that technically exists and practically doesn't. The question "show me the end-to-end journey" comes before the spec, not after.

And slice vertically, not horizontally. Deliver a thin complete path first (a user signs up, signs in, lands on an empty screen that works), then deepen it. Building all the plumbing horizontally before any single journey works end to end is how you accumulate a class of bugs that only appear when the pieces finally meet.

5.7 The spec an agent can't misread

This is the highest-leverage skill of the whole role, and almost no one names it. A human reads a loose spec and fills the gaps with judgment. An agent reads a loose spec and fills the gaps with a guess, confidently. So you write for the reader that will not ask. A spec an agent can't misread has: one goal in one sentence; the exact files and functions to touch, and an explicit list of what NOT to touch; the decisions already made, so it doesn't re-decide; the tests it must make pass; and one trade-off at a time, never a pile. Ambiguity

is not a small tax with an agent. It is the failure. If a sentence can be read two ways, an agent will pick one and build it well.

5.8 Decision hygiene: protecting the one scarce resource

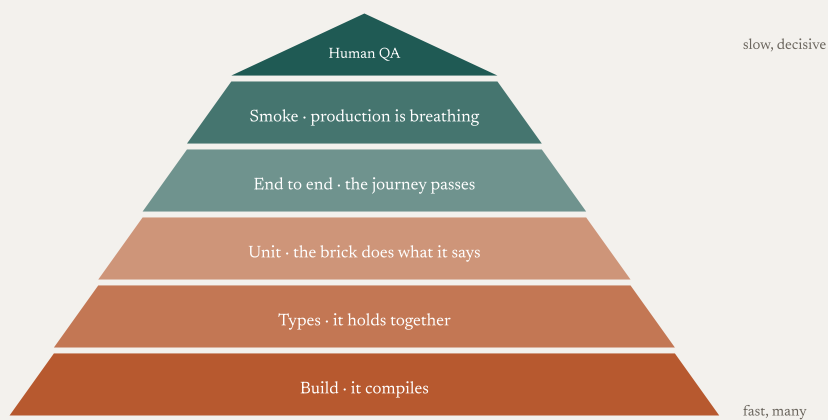
The scarce resource in this way of working is not compute, it's your attention as the decider. Protect it. Decide one trade-off at a time, so each gets real thought. Say no in writing and with a reason, so the no holds and doesn't come back next week. Batch the small reversible calls and move fast on them; slow down only for the hard-to-reverse ones (Part 4). And keep a decision log with the rejected alternative (Law 3), so you're never re-litigating a settled call while a new one waits. A builder who spends attention on everything has none left for the decisions only they can make.

Tests and QA: proof against intention

6.1 The principle, in one line

A test proves what the code does. Your intention proves nothing. The whole discipline is replacing "I'm pretty sure it works" with "here's the proof it works."

6.2 The levels that actually exist



Each tier catches what the one below lets through. The top stays human.

Build check: does it compile. If it doesn't build, it doesn't ship. Type-check: catches a class of errors before anything runs. Lint: style and quality rules. Unit test: one function in isolation. End-to-end test: a real user journey simulated in a browser. Smoke test: a quick check in real conditions right after deploy, like the endpoint answering or the checkout charging the right amount. Human QA: a person running a checklist for the complex journeys (payment, emails, scheduled jobs) before a release. You don't need all of them on day one. You need to know which ones a given change requires.

6.3 False reds, and why they cost as much as false greens

Everyone worries about a test that passes when it shouldn't, a false green (Law 5). The opposite is just as dangerous: a test that fails when nothing is wrong, a false red. A false red trains you to ignore the check. After the third time a gate cries wolf, you stop reading it, and now it protects nothing, even when it's right. A flaky check is not a minor annoyance. It's a check on its way to being disabled.

6.4 What human QA does that tests never will

Automated tests check what you thought to check. A human running the flow notices what you didn't think of: the wording that confuses, the state that feels wrong, the thing that technically works but no user would understand. On money paths especially (payments, refunds, anything with real financial effect), you validate in real test mode, with a real event, not by reading the code and reasoning that it should be fine. Reasoning is not proof.

6.5 How to prove that a guard actually guards

When you add a safety check, prove it guards the way you'd prove any invariant: feed it the bad input on purpose and confirm it blocks. A guard you've only ever seen pass is a guard you haven't tested. It's Law 5 again, aimed at your own safety code.

6.6 Test early, from the first slice

The most expensive testing mistake is testing late. If you build ten slices before you run the whole thing end to end, a break in slice one hides until slice ten, and now you're debugging through nine layers to find it. Run the real journey as early as the first thin version exists. A bug caught at slice one is a five-minute fix. The same bug caught at slice ten is an afternoon.

Data, migrations, and money

7.1 The schema, and changing it safely

Your database has a structure: tables, columns, relationships. A migration is a versioned change to that structure, one file that adds a table, a column, an index. The safe pattern, worth copying: a read-only step first that checks the current state without writing anything (do the target tables already exist, do the things I depend on exist). If the read-only step surprises you, stop before you write. Then the writing step, wrapped so it either fully applies or fully rolls back, never half.

For the CTO in you

lock the database down by default. Turn on row-level security on every table, and let only the server-side role read and write. The frontend never touches the database directly; everything goes through the backend. A table with security off is a door you forgot was open.

7.2 Money is stored in the smallest unit

Never store money as a rounded decimal. Store it in the smallest unit, as a whole number of cents. Decimals drift and round in ways that quietly create money or lose it. This is not pedantry, it's the difference between books that balance and books that don't.

7.3 Idempotence: the same event twice, one effect

Third parties will send you the same event twice: a payment webhook gets replayed, a request retries. If "payment received" runs twice, you must not credit twice. The common guard is a uniqueness constraint, a unique

identifier on the event, so the second arrival is silently ignored. Design for replay, because replay will happen.

7.4 Proving a mass data change

When you change data at scale, you prove it with an invariant, not a re-read (Law 6). After a bulk update, check that the counts match and zero of the old form remains. Before and after a big migration, prove that the rows that shouldn't have moved are byte-for-byte identical. And prove the result on the real data as a delta: these numbers moved from X to Y, and nothing landed where it shouldn't. A green test says "my code ran." A delta says "the world changed the way I intended."

Shipping, security, and operations

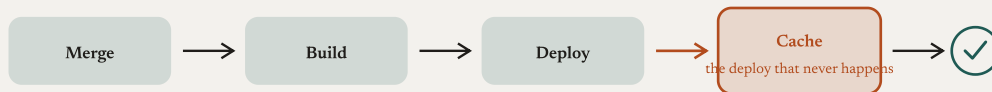
8.1 The merge is the middle of the chain, not the end

Merging your change into main feels like the finish line. It isn't. Merge deploys the code, but a lot happens after: the build runs, the deploy publishes, the cache may need clearing, the schema may need a separate migration. "Merged" is not "live and correct." Confirm live.

8.2 Deploying code is not deploying the schema

Your code and your database change on different tracks. A deploy ships code. It does not run your database migration. If the new code expects a column the migration hasn't added yet, it breaks in production while your local looks fine. Order matters: usually the schema change goes first, and stays backward compatible, then the code. Never assume one carried the other.

8.3 The cache: the invisible deploy that never happens



The merge is the middle of the chain, not the end.
until you have seen the thing alive with your own eyes, it is not shipped

The cache is the only link that lies without erroring: everything is green, and the old version is still being served.

You deploy, you check, nothing changed. The most common reason: a cache, a layer that serves a saved copy for speed, is still handing out the old version. On some surfaces you must purge the cache after each deploy, or users see

yesterday's product. The deploy that "didn't take" is almost always a cache that wasn't cleared.

8.4 Rollback

Shipping without a way back is a bet, not a plan. Before you deploy anything hard to reverse, know how you'd undo it: revert the code, or roll forward with a fix, and what to do about data that already changed. The time to figure out rollback is not during the incident.

8.5 Security as a declarative surface

Most of your security is not clever code, it's a few declarations you either made or forgot. Secrets live outside version control, set in the host. The powerful database key is server-side only; the frontend gets the restricted public key. Row-level security is on for every table. Rate limits cap abuse. These aren't features you build once and admire. They're surfaces you keep flat. One table with security off, or one full-access key in the browser, and the rest of your care doesn't matter.

8.6 Auth is two questions, not one

Authentication asks who are you. Authorization asks what are you allowed to do. They are different, and confusing them is a classic hole: a logged-in user is not automatically an allowed user. Check both, on every sensitive action, on the server. And apply least privilege everywhere: every key, every role, every integration gets the smallest set of permissions that lets it do its job, and nothing more. The blast radius of a leak is exactly the privilege you granted.

Two reflexes that cost nothing and save you: never log a secret (tokens and keys must never land in your logs, where they live forever and get shared around), and serve everything over HTTPS, always.

8.7 Privacy and personal data

If you touch personal data (a name, an email, anything that identifies a human), you inherit responsibilities, not just features. The basics a builder must hold: collect the minimum you need, know where it lives, be able to delete it on request, and never store more than you can protect. If your customers are in Europe, this is the ground floor of privacy law, and "we're a small product" is not an exemption. Privacy is a design constraint you carry from day one, not a checkbox before launch.

8.8 Observability: how you know production is fine

You cannot fix what you can't see. Three layers. Logs: the running record of what happened, which you can stream live to watch a problem as it occurs. Error tracking: a service that captures every crash with its context, so you learn about a bug from a dashboard instead of a furious user. Alerting: a signal that reaches you when a key number goes wrong, before customers notice. Early on, you often find out something broke because someone told you. Closing that gap, from "a user told me" to "my dashboard told me," is one of the first upgrades that separates a toy from a product.

8.9 The incident and rollback playbook

Things will break in production. Panic is the expensive part. Have a plan before you need it. When it breaks: first stop the bleeding (roll back to the last known-good version, or disable the feature behind its flag), then diagnose calmly, then fix forward with a real correction and its regression check (Law 4). Decide in advance what "roll back" means for you: reverting the code is easy, but data that already changed may need its own undo. A rehearsed rollback is a shrug. An improvised one is an outage.

Two cheap safety valves make this easier. Put your risky paths, anything touching money or third-party webhooks, behind a feature flag, so you can switch them off in one move without a redeploy. And sequence releases so

you validate before you promote: prove the thing works before you build the marketing pages that send people to it. Validate, then promote, never the reverse.

8.10 Versioning and releases

Give your releases names and a history. Semantic versioning is the common convention: a version like major.minor.patch, where the numbers signal how big the change is (a patch fixes, a minor adds, a major breaks). Tag each release, and keep a changelog: the dated, human-readable list of what shipped. During a beta you can suffix the version to mark the phase. The point isn't ceremony, it's being able to say exactly what's in production and to walk back to any prior point on purpose.

8.11 Cost is a first-class concern

A builder who ignores cost ships a product that can't survive its own success. Three lines to watch. Infrastructure: what your host, database, and services bill as usage grows. Third-party per-use costs: payments, email, and especially AI model calls, which are billed by the token and add up faster than you expect. And your own pricing metric: if you charge for a usage-based unit, you need to know what each unit costs you to deliver, or you're selling below cost without noticing. Cost is not an afterthought for the finance person. It's a design input, the same as speed or security.

8.12 Telling the world it shipped

Shipping isn't finished until you've told the story. A good release note answers: what shipped (the value, not the technical list), why (the problem solved), a demo or a screenshot, and what's next. Formats that work: a dated changelog oriented on value, a short demo of the real journey, build-in-public along the way, a team note about what changes for them. You speak value and impact, not jargon. The glossary exists precisely to translate the technical into plain terms for this.

Documentation as infrastructure

9.1 Three entry levels, one per reader

Good docs serve three readers at three depths: the newcomer who needs the map, the builder who needs the how, the maintainer who needs the why. One page can't serve all three well, so you layer: a top-level overview, then the practical guide, then the deep reference. Give each reader the door that fits.

9.2 One source of truth per topic

Every topic has exactly one home. When two docs describe the same thing, they drift, and the reader can't tell which is right. If information must appear in two places, one is the source and the other points to it. A fact with two owners is a fact with none.

9.3 A decision is written with its rejected alternatives

Law 3, applied to docs. A decision record isn't "we chose X." It's "we chose X over Y and Z, because, on this date." The rejected options are the valuable part: they stop the re-debate six months later, and they explain the choice to someone who wasn't there.

9.4 Freeze, don't delete

An obsolete plan or doc is not deleted. It's marked obsolete, dated, and left with a pointer to the living version. Deletion loses the reasoning, and the next person rediscovers the dead end from scratch. Freezing keeps the history cheap and the lesson intact.

9.5 The trap: text that describes code is verified by no one

A comment or a doc that says "this function returns X" is checked by nobody. The code can change and the sentence stays, now lying with total confidence. Prefer descriptions that can be checked (a test, a type, an example that runs) over prose, and treat any doc you wrote yourself as a dated hypothesis until you've re-verified it against the real thing (Law 7).

9.6 The self-sufficiency test: a cold newcomer

There's one test for whether your docs are infrastructure and not decoration: can a cold newcomer, a new teammate or a fresh AI agent with no memory of the project, become productive from the docs alone. If they have to ask you, the docs failed, and you have become the single point of failure. Write so the person who wasn't there can start without you. Self-sufficient documentation is what lets you hand off, step away, or bring someone in without re-explaining the whole world.

Design system, content, coherence

10.1 Tokens: style lives in one place

A design system starts with tokens, the named variables for colors, spacing, type. Change the token, the whole product changes with it. Hard-code a color in one screen and you've started a slow drift no one will fully fix. Consistency isn't discipline applied a thousand times, it's one source applied everywhere.

10.2 The one invariant that keeps a product coherent

Pick a small, fixed palette and hold it. A handful of colors used with meaning (one for the primary action, one for danger, one for success) is what makes a product feel like one product. The failure mode isn't ugliness, it's ten slightly different blues that each felt fine alone.

10.3 The component registry

Reuse before you recreate. Keep a registry of the building blocks (the button, the card, the input) and use them. Every one-off component is future inconsistency and future maintenance. When a file grows past a couple hundred lines, that's the signal to split it, not a nag.

10.4 Content is calibrated on a corpus, not decreed

Your product's voice (the microcopy, the empty states, the error messages) isn't invented per screen. It's calibrated on a reference: a few examples that define the tone, and everything else matches them. Content decided screen by screen ends up sounding like five different people, because it was written by five different moods.

10.5 Performance and accessibility (the light touch every builder owes)

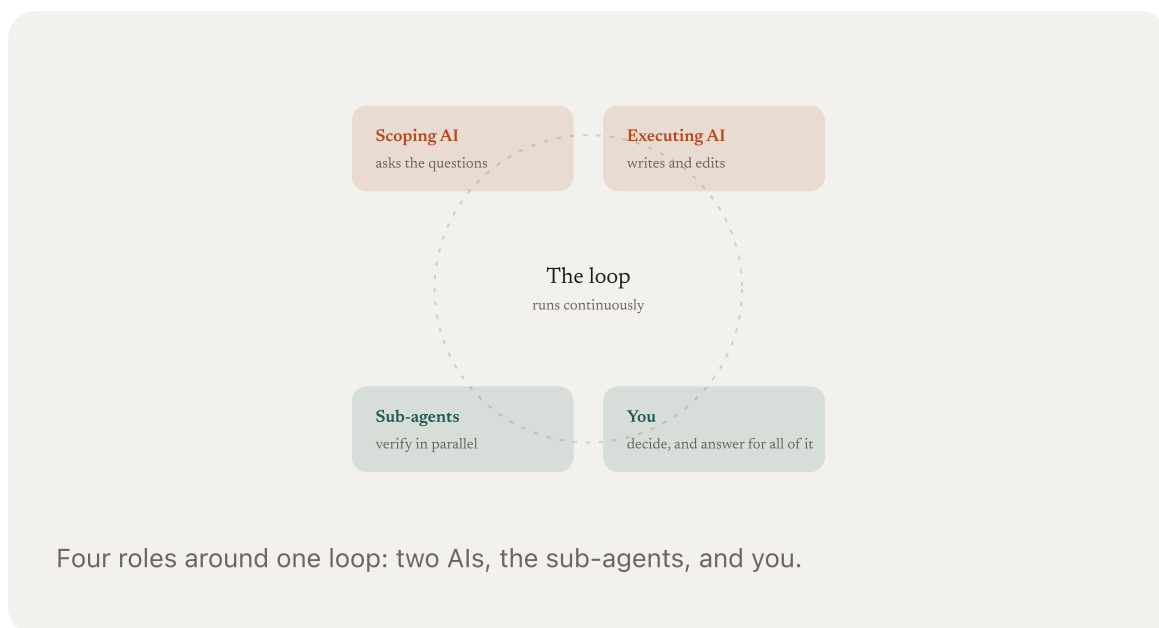
Two things you don't have to master, but must not ignore. Performance: a slow product feels broken even when it works. Watch the basics (how fast the first real content appears, how quickly the page becomes usable) and treat a heavy page as a bug, not a detail. Accessibility: some users navigate by keyboard, some use screen readers, some can't tolerate motion. The cheap wins go a long way: visible focus states so keyboard users can see where they are, honoring the "reduce motion" setting, real text alternatives for images, enough color contrast. It's not extra polish. It's whether a chunk of your users can use the thing at all.

Building with AI: what actually changes

11.1 The new practice

The deep shift: the human becomes the architect and the decider, the AI becomes the partner that explores the code, proposes options, writes the spec, generates the code, and verifies it. You don't type the code line by line. You frame, decide, re-read, validate.

11.2 The roles



Framing AI (the product partner): brainstorms, asks the trade-off questions one at a time, writes the spec, drafts the execution prompts, keeps a memory of the project for continuity. Executing AI (the coding agent): reads the repo first, edits files, writes code, runs the build and type-check, flags its blind spots. Sub-agents: explorers launched in parallel to map the code fast without cluttering the main thread. You: the product decisions, the trade-offs, the final verification, the deploy, the tests, the comms.

11.3 What the AI really does when it codes

It reads the existing code before writing (grounding): it maps the files, the conventions, the insertion points. It doesn't guess, it goes and looks. It proposes options with their trade-offs, but does not settle a product decision for you. It writes the spec then the code, reusing the patterns already in the repo. It verifies mechanically: compiles, types, syntax, style. It flags its blind spots: what it validated by reasoning and not live, the defaults it chose, the manual prerequisites (a secret to set, a migration to apply). And it deliberately does not do certain things, because they are yours: settle a business trade-off, push or deploy without your go, move real money.

In one line for the non-dev reader: the AI is a very strong executor that reads, proposes, codes and checks. The pilot and the safety rail is you.

The economics, because it's the first question a CPO asks. Your time shifts: less of it goes to writing code, more to deciding what to build and verifying what came back. Your costs shift too, from paying for hours to paying for tokens, the unit AI bills by. A session that explores a big codebase burns tokens fast, so grounding efficiently (send the agent to the right files, not the whole repo) is a cost decision, not just a speed one. The return is not free code, it's speed to a real, shippable product with one person instead of a team. Watch your token spend per shipped slice and the ratio of your decision time to the agent's execution time, and watch that ratio improve as your specs sharpen. A vague prompt is expensive twice: in tokens, and in the rework it causes.

11.4 The failure modes specific to agents

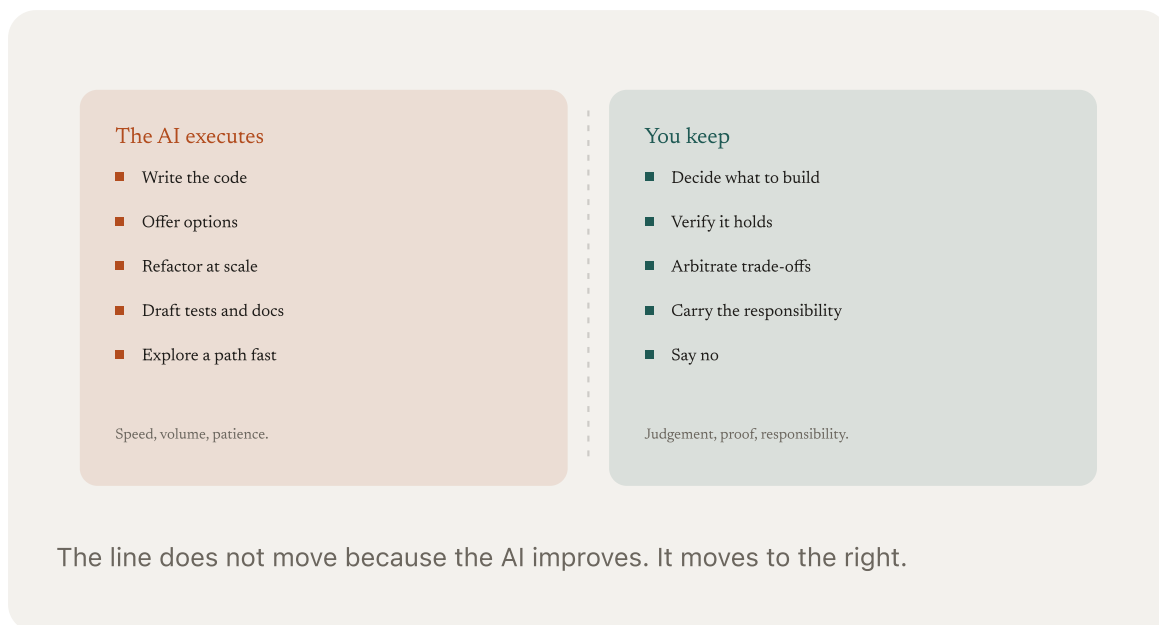
New tools, new ways to fail. An agent can be confidently wrong, stating a false fact in the same tone as a true one (Law 7). It can act on a stale doc it wrote itself. It can "verify" something it never ran (Law 4). Two agents in parallel can collide on the same files (Law 9). It can drift from the plan across

a long session as its memory fills (Law 10). None of these are reasons not to use agents. They're the reasons you keep the human checks that catch them.

11.5 What AI makes mandatory that used to be optional

Some practices were nice-to-have with a careful human and are now non-negotiable with an agent. Grounding before acting. A written spec (the agent needs the source of truth you used to keep in your head). Mechanical checks calibrated on a known-red case (you can't rely on the agent's self-report alone). A project memory with a budget. Explicit "do not touch" boundaries in prompts. The agent doesn't make these optional. It makes them the price of going fast safely.

11.6 What stays human



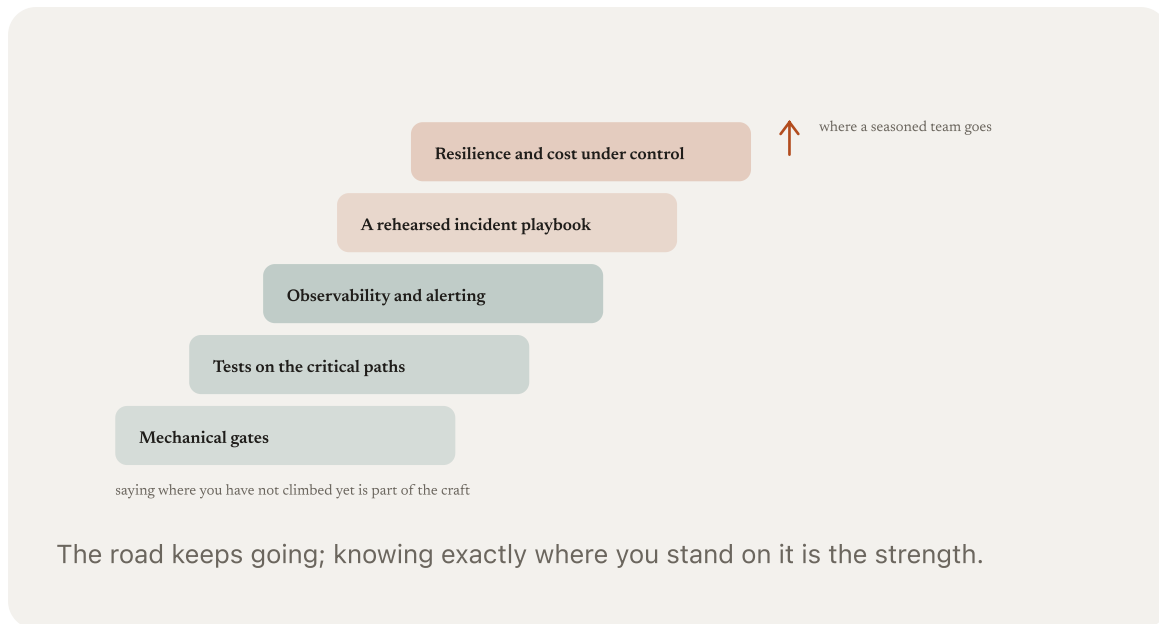
The decisions. The trade-offs. The final verification. The deploy of anything hard to reverse. The taste. And the accountability: when it ships, it's your name on it, not the model's. That's the whole deal, and it's a good one.

The catalogue of traps

A quick-reference index, not a re-telling. Each trap below is the one-line form of a principle from the laws (Part 1) or a chapter, so when you get bitten you scan this list, then go read the full law behind it. Keep your own version, add to it as you get bitten, and let each entry carry its incident (Law 1).

- **"Not pushed" is "gone."** Unpushed local work vanishes on a re-clone. Push after every unit.
- **Wrong account, "repository not found."** A failed push is often the wrong identity active, not a real error.
- **The green check that never ran.** A test suite wired to run nowhere passes by doing nothing. Make it go red on purpose first.
- **The verification written but not executed.** Worse than none: it closes the subject. Ship the command and the regression signal, not the reassurance.
- **Deploy did nothing? Clear the cache.** The most common "my deploy didn't take" is a stale cache.
- **Code shipped, schema didn't.** A deploy carries code, not your migration. Order and backward compatibility matter.
- **The name that lies.** An object named one thing means another; a relationship built on name resemblance rejects most of the rows. Prove with usage, not names.
- **Money as a decimal.** Rounding invents and loses money. Store cents as whole numbers.
- **The replayed webhook.** The same event arrives twice. Without a uniqueness guard, you double the effect.
- **The full-access key in the frontend.** One leaked powerful key opens the whole database. Server-side only, always.
- **Two agents, one file.** Parallel work on the same big files collides. One wave at a time per area.
- **The doc you trust that's a hypothesis.** Your own doc is dated and unverified until proven against the real thing.

Going further (what I haven't done yet, and what dev+++ looks like)



An honest section, on purpose. Owning the whole loop means being clear about where a seasoned engineering team would go further than I have, so far. This is not a confession. It's a map. Some of these are deliberate early trade-offs, some are next on the list, all of them are what leveling up looks like. Saying it out loud is part of the craft, and part of the trust.

The AI Product Builder gets you to a real, working, production product far faster and further than most people believe. Knowing exactly where the senior-team practices begin, and being honest that you are not all the way there yet, is a strength, not a weakness. A builder who pretends to have done everything is the one you can't trust.

Here is the ladder, roughly in the order most solo-built products should climb it. To be refined with senior engineering input.

Automated test coverage in the pipeline. Unit, integration, and end-to-end tests running on every change, blocking a merge when they fail. Many first versions ship with thin automated tests and heavy manual QA, which is

a valid early trade-off. The upgrade is a real suite wired into the pipeline, so the safety net catches you before a human has to.

Observability. Structured logs, metrics, tracing, and alerting, so you learn about a problem from a dashboard, not from an angry user. A senior setup pages you before customers notice. Early on, you often find out because someone tells you. That gap is one of the first to close.

A formal security posture. A documented threat model, dependency scanning, secret rotation, and eventually certifications like SOC 2 or ISO when customers require them. Early stage, you are honest that you don't have these yet, and you don't pretend a screenshot of a lock icon is a security program.

Load and performance testing. Proving the system holds at ten times the traffic before it happens, not after the outage or the surprise bill. Until you've tested it, "it scales" is a hope.

Disaster recovery. Backups you have actually restored, and a rehearsed plan for the day something is lost. A backup you have never restored is not a backup, it's a wish.

Infrastructure as code. The whole stack defined in files, so spinning up a fresh environment is a command, not a memory and a lucky afternoon.

Progressive delivery. Feature flags, canary releases, staged rollouts, so a bad change reaches one percent of users before it reaches everyone.

Accessibility and internationalization done to standard. Not by touch and good intentions, but against real guidelines and with real testing.

None of this makes the earlier chapters less true. You can ship a genuine, paying product without all of it. This part exists so you know the road keeps going, and so that when a senior engineer reads your work, they see someone who knows exactly where they stand.

Things I tried and killed

A book with only wins isn't credible. A few things I built and then removed, anonymized. A taxonomy system meant to organize content that added more complexity than it ever paid back, cut. A component I promoted into the shared design system that no screen ever used, deleted. A grand plan for thousands of auto-generated pages, scrapped once the effort-to-value math turned negative. Each taught the same lesson: a rejected path is not wasted if you write down why you rejected it (Law 3). The graveyard is part of the map.

Glossary

DEPLOYMENT

Gate

An automatic check that blocks until it is satisfied.

see Law 5, Law 8, Part 3

DATA

Idempotence

The same event twice produces exactly one effect.

see Part 7.3

How to read an entry: category, term, plain definition, and where it's used.

Version control and collaboration

Repository (repo)

the versioned project.

Commit

a dated snapshot of a set of changes, with a message.

Branch

a parallel line of work. Main is the production line.

Remote / origin

the reference copy hosted online.

Push / pull

send your commits up / pull others' down.

Clone

copy a repo to your machine.

Merge

fold one branch into another.

Pull request (PR)

a request to merge, usually reviewed first.

Code review

a peer (or the AI) re-reading before merge.

Working tree

the state of your files locally, committed or not.

Deployment and infrastructure

Build

compile and prepare the shippable files.

Deploy

publish the build, make it live.

Rollback

return to the previous version.

CI/CD

automation of build, then tests, then deploy.

Pipeline

the chain of automated steps.

Gate

an automatic check that can block the line.

Environment

local, preprod/staging, production.

Env var

a config value that changes per environment.

Secret

a sensitive value (a key), never in version control.

Cache

a saved copy served for speed. Purge it after some deploys or you serve the old version.

CDN

a network that serves content fast from cached copies.

Preview URL

a test URL generated for a branch before production.

Backend and data

API

the interface used to talk to the backend.

Endpoint

a URL plus a method (e.g. a create-event call).

Auth / token

proving who you are with a signed token, checked by the server.

Webhook

an incoming call a third party makes to you on an event.

Schema

the structure of the database (tables, columns, relations).

Migration

a versioned change to the schema.

Row-level security

access rules enforced per row, so the frontend can't read what it shouldn't.

Full-access key vs public key

the powerful key (server-side only) and the restricted key (safe for the frontend).

Rate limiting

capping requests per period to prevent abuse.

Idempotence

the same operation twice, one effect.

Encryption

protecting sensitive data at rest.

Cron

a task that runs on a schedule.

Frontend

Component

a reusable block of interface.

State (of a screen)

loading, empty, error, success. Handle all four.

Design token

a named style variable (color, spacing, type).

Feature flag

a switch that shows or hides a feature per environment. Off by default means not rendered, not just disabled.

Optimistic UI

updating the screen immediately, before the server confirms.

Tests and quality

Type-check

verifying the shapes of data before anything runs.

Lint

checking style and quality rules.

Unit test

verifying one function in isolation.

End-to-end test

simulating a full user journey.

Smoke test / sanity check

a quick real-conditions check after deploy.

QA

quality assurance, often a manual checklist before a release.

Regression

something that used to work and now doesn't.

False green / false red

a check that passes when it shouldn't / fails when nothing is wrong.

Invariant

a fact that must hold, used to prove a mass change.

Hotfix

an urgent fix pushed fast.

Security, operations, and cost

Authentication vs authorization

who you are, vs what you're allowed to do. Check both.

Least privilege

grant the smallest permissions that do the job. A leak's blast radius equals the privilege you gave.

PII (personal data)

anything that identifies a person. Collect the minimum, protect it, be able to delete it.

OAuth

a standard way to let a user grant limited access without sharing their password.

HTTPS

encrypted transport. Everything, always.

Observability

logs, error tracking, and alerting, so you learn about a problem from a dashboard, not a user.

Error tracking

a service that captures crashes with their context.

Alerting

a signal that reaches you when a key number goes wrong.

Incident

something broke in production; the plan is stop the bleeding, diagnose, fix forward.

Semver (semantic versioning)

major.minor.patch, signaling how big a change is.

Changelog

the dated, human-readable list of what shipped.

Lockfile

the file that pins exact dependency versions for reproducible builds.

Monorepo vs multi-repo

one repository holding several projects, vs several separate repositories.

Staging

a preprod environment that mirrors production for a final check.

Unit economics

what one unit of usage costs you to deliver, vs what you charge for it.

Building with AI

Prompt

the instruction you give the AI.

Grounding

the AI reading the existing code before it acts.

Agent / sub-agent

an AI working autonomously on a task, and explorers run in parallel.

Spec (design doc)

the document that fixes the scope and the decisions.

Wave

a small, testable, shippable unit of work.

LLM

the large language model that generates.

Token (AI)

the unit of text an LLM reads and writes, and the basis of AI billing.

Standing instructions

the house rules repeated in every prompt.

The self-assessment

Process and decisions

- ☐ Do you write a short spec before the build, not after?
- ☐ Do you decide one trade-off at a time, in writing, with the rejected option noted?
- ☐ Can you say, right now, what "done" means for you, and does it include "in production and seen working"?
- ☐ Do you slice vertically, a thin complete journey first, then depth?

Version control and shipping

- ☐ Do you push after every unit of work, always?
- ☐ Before you deploy something risky, do you know exactly how you'd roll it back?
- ☐ Do you keep a changelog and version your releases?
- ☐ Do risky paths (money, webhooks) sit behind a switch you can flip without a redeploy?

Proof over intention

- ☐ Have you seen each of your safety checks fail on purpose at least once?
- ☐ Do you test the real journey early, from the first slice, not at the end?
- ☐ On money paths, do you validate with a real test event, not just a code read?
- ☐ When you change data at scale, do you prove it with an invariant, not a re-read?

Security, data, ops

- ☐ Is your full-access key server-side only, never in the frontend?
- ☐ Do you know where every environment variable lives and who sets it?
- ☐ Is personal data collected minimally, protected, and deletable on request?
- ☐ Do you learn about a production problem from a dashboard, or from an angry user?

Documentation and AI

- ☐ Could a cold newcomer, human or agent, get productive from your docs alone?

- ☐ Are your specs written so an agent can't reasonably misread them (explicit scope, do-not-touch, one goal)?
 - ☐ Do you keep a project memory with a budget, one that tends toward empty?
 - ☐ When it ships, are you clear that the decision, the verification, and the accountability were yours?
-

Scoring

Count the yeses, out of 20.

16–20 • Building like a pro. you're building like a pro; keep the honesty about what's next (Part 13).

10–15 • Solid loop. solid loop, and the gaps are usually proof and ops, not code.

0–9 • Shipping on hope. you're shipping on hope; start with the push rule, the early test, and a real definition of done.

Case notes (anonymized)

Same lessons, no names, no addresses. Each note is a real shape of problem, stripped of anything that could map back to a live product. The point is the pattern, not the address.

A consumer-facing SaaS. A product where users send events and get automated actions back. The sharpest lessons here were about money and replay: store money as whole cents, and guard every incoming third-party event with a uniqueness constraint so a replayed webhook never doubles an effect (Part 7). And about the home screen: the biggest product win was removing a feature, not adding one, because the thing users actually needed was "what do I do now," not "here is your data" (Law 3 on rejected paths, and the taste that stays human, Part 11).

An internal tool. A back office used by a small operations team. The lessons were about data truth and mass change: an object whose name did not match its meaning caused a relationship to reject most of its rows until it was fixed by usage rather than by name (Law 7), and every bulk data change was proven by an invariant and a production delta, never by a re-read (Law 6).

A client app. A product built for someone else, where documentation and verification carried the most weight. The two most expensive lessons: a verification that was written down but never actually run, which quietly left files exposed for months (Law 4), and the discipline of writing every decision with its rejected alternatives so the same debate didn't reopen months later with someone who wasn't there (Law 3).

Keep going

Work with me – benoitlecureur.com/work-with-me

Subscribe to Off Market – benoitlecureur.com/off-market

Read the playbook online – benoitlecureur.com/playbook