AI Builds, You own

Book I

The Backlog that thinks

What to settle before the first user story is refined

Haider Lasani

Contents

Preface

Copyright

First edition.

Who this book is for

This book is for the people who decide what gets built.

The founder who knows the team is shipping faster with AI and cannot quite explain why the customer-support queue keeps getting longer.

The CTO who has watched two teams adopt the same AI tools and get opposite results, and wants to know what the difference was.

The product owner who is now drafting tickets that AI agents will build from, and is starting to suspect that the work of writing a ticket is no longer the same job it used to be.

The engineer who is reviewing code they did not write, on a schedule that did not slow down to match.

The book assumes you are working today. It assumes the AI tools you use change every six months and the bottleneck does not. It assumes you have read enough hot takes about AI and would prefer something specific.

The book does not assume you write code. The chapters on review, ownership, and acceptance criteria are written so that a non-coding PO can apply them with the same precision as a senior engineer. The chapters on the underlying mechanics are written so that an engineer can read them without being bored.

How to use this book

This is a practical field guide, written through stories.

You can read it in order. The chapters build on each other. Each chapter ends with a bridge that names the next problem honestly, not with a teaser. By the time you reach the synthesis chapter, the parts will assemble into a system the team can run.

You can also read any chapter standalone. Each one is built to land on its own. If you are about to refine a ticket, the chapter on the eight decisions is enough on its own. If your team is arguing about lanes, the chapter on the process the work deserves answers that argument. If a story has been stuck for three sprints, the chapter on slicing is the one to read.

The case studies in the back are illustrative composites of real engagements. They show the system in motion. Read them after the chapters they reference.

The glossary is for words that show up across multiple chapters in a specific sense. If a term feels unfamiliar in a chapter, the glossary is short enough to scan in under two minutes.

There is no required sequence. There is no certification. There is nothing you need to buy to understand this book. Some teams will choose to wire their existing tools together; the discipline itself lives in the habits. The contribution is the names and the ordering. If the names land, the habits become repeatable. If the habits are repeatable, they compound.

A note on evidence

Many examples in this book are composites that combine patterns observed across teams, so a lesson can be shown without exposing any one company's private work. Where a number comes from a published study, it is named in *Sources and Notes*. Where a number comes from field experience, read it as an observation, not a benchmark.

A note on scope

This book discusses software delivery, security, privacy, and compliance through examples, for educational purposes. It is not legal, security, or compliance advice. Teams should consult qualified professionals for their own regulatory and risk requirements.

A note on the series

This is the first of five books on the same idea: the parts of building software that the team owns, and the parts AI does on top of that ownership. The series is *AI Builds, You Own*. The remaining four cover the layers downstream of the backlog: the architecture decisions made before the first commit, the codebase the team has to live with after the code outlives its author, the release that protects what the team shipped, and production once the build is alive.

Each layer is a habit. Each book is one layer. The backlog is the first one, because everything else compounds whatever the backlog decides.

You do not need the other books to use this one. This one stands alone. If it changes how your team refines a story, it has done its job.

Part I

The Problem

The first two chapters describe what is broken and why it costs more than it used to. The demo that lied. The sentence the team typed without realizing it was a hundred-decision bill. The averaging that happens when an AI is asked to fill in what the team did not.

These chapters set the stakes. Everything else in the book is the response.

The Demo That Lied

Tuesday

It is Tuesday. The founder is showing the new feature on a shared screen, and the room is unusually quiet because the demo is going well. The form submits. The dashboard updates. The chart animates the way the chart is supposed to animate. The investor on the call says the word “impressive,” which is a word investors say when they want to say “fine” without sounding rude. Everyone smiles. Someone types a celebration emoji in the chat. The recording gets clipped and shared.

Three weeks later, the same feature is on fire.

A customer hits an edge case. In retrospect, an obvious one. The kind anyone on the team would have caught if they had paused for ten seconds. The engineer is asked about it and says the case was never in the ticket. The PO opens the story and reads it twice. It looks like a complete user story: every field filled in, acceptance criteria written out, a clean format. The case is not

in any of it. The PO remembers writing the ticket. It was one sentence in chat, handed to the AI. The AI wrote the rest of the story around the sentence. It did not put in what was never said.

You have seen this movie. You may be in it right now.

A demo runs once, on the path you chose.
A product runs every day, on every path the customer chose.

Why this keeps happening

The standard story about AI in product delivery goes like this. The tools are getting better fast. Models are smarter. Agents are more autonomous. Therefore delivery will get faster. Therefore the bottleneck is which tools you adopt and how many seats you buy.

That story is not exactly wrong. It is looking in the wrong place.

The bottleneck is not the model. The bottleneck is what you hand the model. AI does not get faster when you write a vague ticket at it. AI gets confidently wrong faster. The demo on Tuesday looked the way it did because the happy path is the easiest thing in software to generate. The product collapsed in week three because the parts of the work that did not fit on the happy path were never written down anywhere a person or a machine could read them.

In other words, the failure was not in the build. The failure was upstream of the build, in the place where a sentence was supposed to become a contract and did not.

The numbers caught up

This was opinion until recently. The research now complicates the simple story in both directions.

In a randomized controlled trial with ninety-six Google engineers, AI assistance shortened the time developers spent on a complex, enterprise-grade task by about 21 percent. The authors stressed the confidence interval was wide and warned against assuming the result generalizes across tools and settings.

In a separate randomized trial, the research group METR found that experienced open-source developers, working in repositories they already knew well, took about 19 percent longer with early-2025 AI tools, even though they had expected to be faster.

The two findings are not in conflict. They are the same lesson from two sides. AI is not a fixed multiplier on delivery. What surrounds it (the clarity of the task, the context the tool can read, the workflow it runs in) decides whether it speeds the team up or quietly slows them down.

The 2025 DORA report names it directly: "AI's primary role is as an amplifier, magnifying an organization's existing strengths and weaknesses." Read in operating terms: AI behaves less like a fixed productivity multiplier and more like an amplifier of whatever upstream discipline a team already has. The demo on Tuesday

is what the amplifier looks like in week one. Week three is what the amplifier looks like when the upstream work was missing.

A benchmark from outside software makes the same point in a different vocabulary. Mercor's APEX-Agents tested AI agents on long, multi-step professional tasks drawn from investment banking, consulting, and law. The best agent finished under a quarter of them on the first attempt. APEX is not a coding benchmark. On software-specific benchmarks such as SWE-bench, agents now resolve a far higher share of tasks, and that contrast is exactly the point. Agents do best where the constraints are hard and verifiable: code runs or it does not, a test passes or it does not. They do worst where the work is long, ambiguous, and multi-step. A vague backlog turns coding work into the second kind.

This is why the book is needed now in a way it would not have been needed two years ago. The cost of vague work used to be paid by an engineer who interpreted it. The cost is now paid by an AI that builds against it, a reviewer who has to catch what the AI got wrong, and a customer who hits what nobody caught. The amplifier did not change the rules. It changed the bill.

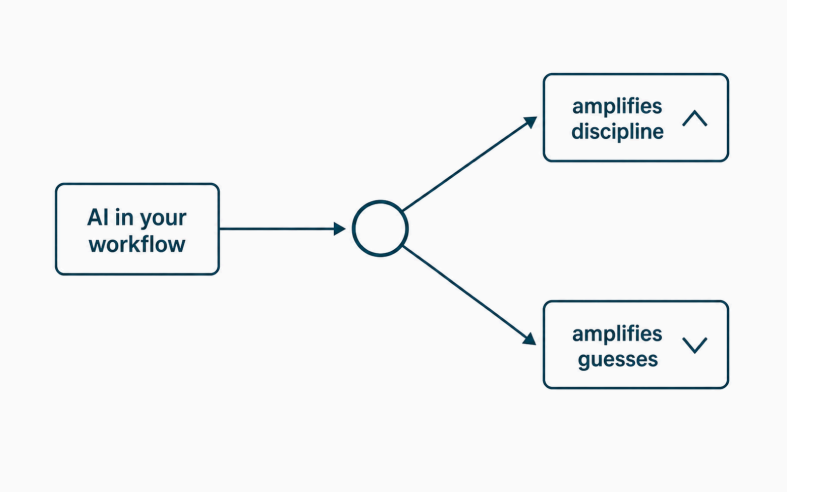

Figure 1: Figure 1.1. AI is a fork, not a multiplier: the direction it runs in depends on what is upstream of it.

What this book is about

This is a book about the upstream layer. The product backlog.

Not the tool you call a backlog. Not the board with the cards. The thinking that is supposed to live inside the cards. The dimensions you fill in before refinement. The decisions you make before a sentence becomes a story. The forks you name out loud instead of hiding them inside an implementation. The slices you cut so a single deliverable can actually be finished. The acceptance criteria you write as tests on purpose. The loop that catches the part of the plan that was wrong, so the next plan is a little less wrong.

Every chapter is one piece of that layer. Each piece is small enough to use on Monday. None of them require buying anything.

If the book has one promise, it is this. *A backlog written for AI to build correctly is the same backlog a team can own confidently.* The two are not separate goals. The discipline that makes AI useful is the same discipline that makes the team's work survive its first contact with reality.

The mirror inside the code

There is a quieter thread that runs underneath the practical work, and it is worth naming early so the rest of the book reads in the right light.

A codebase is not only a machine. It is also a mirror. It reflects the thinking that created it. Rushed work carries the rush forward. Vague work preserves the guess. An avoided product decision shows up later as a generic abstraction that does not quite fit any of its three callers. Clear, owned work carries the care forward in the same way. The code does not choose what it reflects. It reflects what it was given.

This matters because a backlog that thinks is, in the end, how the team makes sure the reflection is one it is proud of. The eight dimensions, the documents, the named fork, the honest acceptance criterion: each one is a small refusal to leave a guess inside the work. The system the AI reads is the team refusing to let the AI average over their product. The plan-versus-actual loop is the team refusing to forget what the work taught them.

The book is about productivity in the same way that a careful contract is about productivity. Both are also about something

else. The book will return to this thread quietly, a chapter at a time. *The Mirror Inside the Code* is the concentrated version.

What this book is, positively

This is a practical field guide. The reason the system works is not the words. The system works because it forces upstream decisions to live upstream, instead of leaking into a sprint that someone later has to explain to a customer. This system comes from patterns observed across real teams: what survived turnover, what scaled, and what broke when AI entered the workflow. The book is the names. You are welcome to use any part of it, ignore any part of it, or take the whole thing and rename every word. The names are not the point. The habits are.

The bridge

The next chapter is the first habit. It is also the simplest demonstration of what AI averages when there is nothing for it to read. The example is a three-word sentence someone types in Slack at 4:47 on a Thursday. *Just add login.* What AI hears when you type that, and what the team will pay for what it hears, is the start.

Read that chapter and you will not unsee what a vague ticket actually costs. The rest of the book is the response.

The Cost of a Vague Request

A short sentence with a long bill

Someone types it in Slack at 4:47 on a Thursday. *Just add login.* Three words. Two of them are filler. The third is doing the work of a hundred decisions that have not been made.

By Monday, the AI product tool has turned the sentence into a story. The story has a title, a description, a couple of acceptance criteria, and a story-point estimate. It looks finished. It is the cleanest backlog item the team has seen in months. Everyone agrees it is ready to refine. By Tuesday, the engineer has it in progress. By the following Tuesday, the engineer is asking the founder why the password reset email goes to spam in Outlook but not Gmail. The founder is asking why there is a password reset email at all, because she meant Google sign-in.

This is not a story about AI being bad at writing tickets. AI is fine at writing tickets that look like tickets. This is a story about what the ticket was *supposed* to settle, and didn't.

> *A story is not a sentence. A story is the smallest contract a team can sign before it builds something.*

What the words don't say

Take the sentence apart. Each blank you fill in becomes work. Each blank you skip becomes rework.

Just add login does not say what method. Password. Magic link. Social sign-in, and if so, which providers in which order. Single sign-on, and if so, which directory. Passwordless. Two-factor, and if so, mandatory or optional, and through which channel. Each branch implies a different schema, a different security review, a different recovery flow, and a different week of work.

It does not say what an account is. A user with a personal account. A user inside an organization with a role. A user invited to an organization who has not accepted. A user who exists in two organizations and switches between them. A user the team imports from a previous system on day one. Whichever of those you pick, you also pick a billing model, a sharing model, and a permission model. None of those are in the sentence.

It does not say what a session is. Twenty-four hours. Thirty days. Rolling refresh. Idle timeout. Sign-out everywhere. The wrong choice here is not visible until the customer in another country logs in on a phone they lost.

It does not say what happens when something is wrong. The email is already taken. The link is six days old. The user forgot which Google account they used. The user is locked out by the rate limiter your security tool quietly applies. Each of those is a story by itself.

It does not say which screen the user sees first. Where the form lives. What signup looks like. Whether signup is even part of the same story. Whether a wrong-password error tells the attacker the email is real or hides it on purpose.

It does not say which laws apply. Where the data lives. Whether the user can delete their account, and what happens to their data if they do. Whether their consent for a marketing email is a separate flag, and whether the answer is in the user table or somewhere your privacy reviewer or data protection officer has not seen.

The sentence is three words. The work is a hundred decisions. The AI is doing what AI does. It is filling in the blanks with the most common version of each answer it has ever seen. So is the engineer. So is the founder. They are filling in different blanks with different defaults. The product is the disagreement.

Why this is not a writing problem

It is tempting to read what I just wrote and conclude that the team needs a more detailed ticket template. It does not. A template with a hundred fields gets skipped, because nobody fills in a hundred fields at 4:47 on a Thursday. The problem is not that the team is using too few words. The problem is that the words they are using are not pointing at the things that have to be settled.

Consider the difference between these two prompts. Same intent. Same person. Same Slack thread.

> *Just add login.*

> *We need email-and-password sign-in for individual users, plus Google sign-in. No SSO this quarter. Account is per-user. Sessions last thirty days, rolling, with a single sign-out everywhere button. Password reset via email link, link expires in one hour, link is single-use. Forgotten emails are out of scope. Region is the Netherlands, data lives in the EU. The login screen replaces the current marketing homepage when the user has not signed in for fourteen days, otherwise the marketing homepage stays. Error messages do not reveal whether an email exists. No 2FA in this story; 2FA is a follow-up.*

The second one is not a template. It is a sentence. It happens to be a sentence that contains the decisions the first sentence pretended did not exist. The engineer reading it cannot build

the wrong thing by accident. The AI reading it cannot guess the wrong thing on purpose. The founder reading it back two months later cannot ask why the email goes to spam in Outlook, because the founder already chose what the email is for.

The cost of unanswered decisions is not what they cost to answer in the ticket. It is what they cost to answer in production, in customer support, during the next release, while three other people are also trying to ship.

What AI is actually doing

When you hand AI a vague sentence and ask for a backlog story, AI is not lying to you. AI is doing something more interesting and more dangerous. It is averaging.

It looks at the sentence. It looks at what most logins do in the training data it has seen. It picks the most common version of each blank. It writes a story that reads as if those choices were yours. It is not a bug in the model. It is the model doing the only thing it can do without your decisions: average.

The output looks confident because the model is confident. The model is confident about the average. It is not confident about you, because you are not in the sentence.

The engineer is also averaging. The engineer reads the story, recognizes most of it because the engineer has built login screens

before, and fills in the rest the way the engineer's last team did it. Two months later, the rest is the support queue.

You do not have a tooling problem. You have an *averaging* problem. Every reader, human or machine, fills in the same gaps with their own average. The product ships as the merge of three or four averages. The bugs in week three are the places where the averages disagreed.

> **Myth.** A clearer prompt to AI will produce a clearer ticket.
>
> **Reality.** A clearer prompt produces a more confident-sounding ticket. Clarity in the ticket comes from clarity in the decisions behind the prompt. The decisions are upstream of the words. Skipping them does not save time. It moves the cost from refinement to support.

The vague-ticket tax

There is a tax on every vague ticket. It is paid in pieces, by different people, at different times, with different names.

The product owner pays it in refinement, when they end up answering the same question for the third time because the answer was never recorded. The engineer pays it mid-build, when they stop to ask the question the ticket did not answer, and lose forty minutes of focus. The QA engineer pays it when they file a defect that is not a defect; it is the absence of a decision. The customer pays it when they hit the case the team never settled.

The founder pays it when the customer churns and the team writes the post-mortem.

The tax never appears on a single line in any system. It is distributed. That is why it survives.

You can see it if you look. Count, for one sprint, how many times someone in your team had to ask "what should it do when," "is X in scope," "did we decide," or "I assumed Y, is that right." Each one is a tax payment. Each one was avoidable upstream of the sprint, and unavoidable downstream of it. The tax is not the questions. The tax is having to ask them in the wrong week.

What the ticket is for

The ticket is not a description of the work. The ticket is the *settled set of decisions* that lets the work be done.

This is the shift. The ticket stops being a paragraph that tries to describe the feature, and starts being a small contract that records the choices the team has already made, the ones it is about to make, and the ones it has explicitly decided to defer. The work, when it begins, is the implementation of that contract. The work, when it ends, is the verification of that contract.

The contract has a known shape. There are specific dimensions a ticket has to resolve before refinement is over. They are not optional. They are also not infinite. There are eight of them, and they are the subject of the next chapter.

For now, hold the picture. Three words went into the backlog, and the team paid a hundred-decision bill at the customer-support

window. The fix is not to write a longer sentence. The fix is to know which decisions the sentence has to contain, so that whoever writes it, human or AI, writes one the team can actually build.

The shortest path to fast delivery is a slower ticket. The ticket is the only place in the workflow where you are still cheap.

The bridge

If a vague ticket is a hundred-decision bill in disguise, the obvious question is: which decisions, exactly. Not every story needs every decision settled. But every story needs the same small set of dimensions answered, or the rest of the work is a guess.

There are eight of them. Once you can see them, you cannot stop seeing them, because they appear in every ticket that ever went wrong on you. The next chapter names them and shows what they look like when they are present and when they are not.

Part II

The Foundations

The next four chapters are the load-bearing layer. The eight decisions every story has to carry. The three documents every ticket binds to. The system the AI reads when the wiring is in place. The mirror inside the code, which is why any of this is worth the discipline.

Most teams that succeed with AI have these four in place. Most teams that struggle have one of them missing. The chapter on whichever piece your team is missing is the one that will move your team the most. Read it first.

The Eight Decisions Every Story Carries

A short list that catches most failures

There is a small joke I tell teams when we start. Most software projects do not fail. They are *failed*, in slow motion, by the absence of eight specific decisions that the team kept meaning to make but never wrote down. Once those eight things are on the ticket, the project does not become easier. It becomes possible.

The eight are not a template you fill in. They are the dimensions a ticket has to resolve before refinement is over. If any of them is missing, the rest of the work is a guess, and the guesses are what build the product the customer eventually complains about.

A small note on vocabulary before we name them. I call them decisions in the title because each one requires a call. I call them dimensions in the system because each one names a side of the work. The words point at the same shape.

A story without these eight is not a story.
It is a wish.

You can write them down on a sticky note. You can put them at the top of your ticket template. You can put them in your team handbook. The format does not matter. The eight matter.

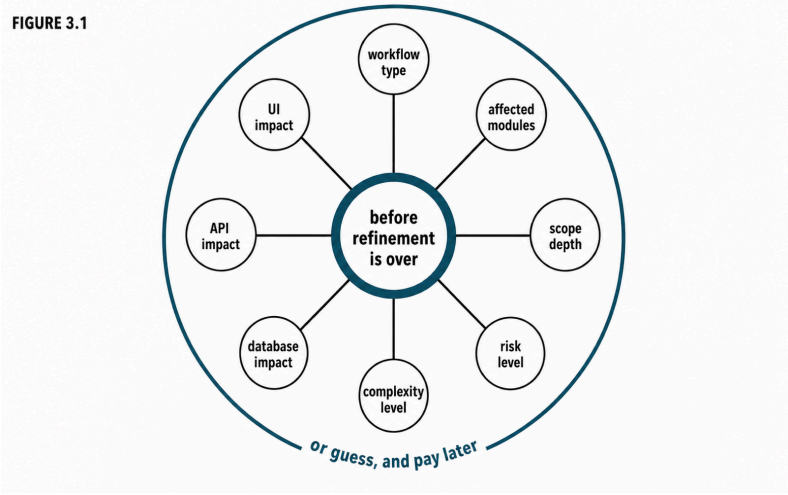

Figure 2: Figure 3.1. The eight dimensions arranged as a wheel: the work that has to be done before the ticket can be refined.

Here they are. Each one gets a definition, a what-bad-looks-like, and a what-good-looks-like.

1. Workflow type

Definition. What kind of work is this? A feature is not a bug fix. A migration is not a refactor. A research spike is not a content

update. Each one has different process requirements, different success criteria, and different stakeholders.

Bad. The ticket says "*Improve checkout.*" Improve how. Is this a bug, a feature, a redesign, a performance tweak, or an experiment. Each one would be done differently.

Good. The ticket says "*Feature: add Apple Pay as a payment method at checkout, alongside existing card and bank-transfer options.*" The word "feature" is doing work. It tells the team this is additive, customer-visible, and requires design review. It tells AI which template to fill. It tells QA which lane.

Cost of skipping. Work that is misclassified gets misrouted. A migration treated as a feature skips the rollback plan. A bug treated as a refactor skips the regression test. The mistake is invisible until the wrong shoe drops.

2. Affected modules

Definition. Which named parts of the product does this touch. Not "the backend." The list. In business language, not folder names.

Bad. "*Touches the API.*" Which API. Which endpoints. Which downstream service. Which dependent feature. Vague paths are vague guarantees.

Good. "*Touches: Checkout (payment intent service), Notifications (order receipt email), Billing (invoice generator). Does not touch: Inventory, Catalog.*" Now the reviewer knows who needs to look at it. Now AI knows where to read context.

Now the regression watch knows what could break that has nothing to do with the new code.

Cost of skipping. Reviews go to the wrong people. Context loading is too narrow or too wide. Tests miss the side effects. The story gets done in one module and breaks two others that nobody told the engineer about.

> *A common version of this looks like a ticket that says "*Update the user profile page.*" The engineer changes a shared component, which is also used in the admin panel, which breaks the admin's user search the next morning. The story's module list said "user profile." It did not say "the shared component used by user profile and three other surfaces." It costs two days. Two days that did not exist in the sprint.*

3. Scope depth

Definition. How far does this change reach. Four levels are enough. *Single file. Single module. Multi-module. System-wide.*

Bad. "*Small change.*" Small according to whom. Small how. A small change to a shared component is a system-wide change wearing a t-shirt.

Good. "*Scope: single module. The change is fully contained within Notifications. No callers change. No contracts change.*" Or: "*Scope: multi-module. Touches Checkout and Billing. Contract between them does not change. Both modules ship in the same release.*" Or: "*Scope: system-wide. Touches the auth layer, which every module depends on. Requires regression sweep.*"

Cost of skipping. Scope is the input to lane selection (next chapter) and to the size of the review. Misjudging scope means misjudging the process. A system-wide change pushed through a fast lane becomes a Friday incident.

4. Risk level

Definition. What can this break that would be hard to put back. Not the same as complexity. Risk asks: if this goes wrong, can we roll back, and at what cost.

Three levels work. *Low* (reversible in minutes, no customer impact). *Medium* (reversible in hours, possible customer impact). *High* (hard to reverse, customer-visible impact, or both).

Bad. No risk noted. Or "*low risk*" with no reasoning.

Good. "*Risk: medium. The new code path is behind a feature flag, so rollout is reversible by toggling the flag. The migration that adds the column is additive only and reversible. Customer impact during failure: form rejects valid input until the flag is off, no data corruption.*"

Cost of skipping. Risk drives review depth, gating, monitoring, and rollback planning. Unstated risk gets the cheapest version of all four. Cheapest version of those things is also called Monday morning.

5. Complexity level

Definition. How much of this is new ground. Different from risk. Risk is about *what breaks*. Complexity is about *what is not known*.

Three levels also work. *Trivial* (the team has done this exact thing before, recently). *Medium* (the team has done close-enough, with some new). *High* (genuinely novel logic, novel integration, or novel constraints).

Bad. Complexity inferred from story points alone. A single number compresses two different things (how much work, and how uncertain it is), which is why estimates feel subjective. *The Memory That Makes Estimates Better* covers the fix; for refinement, name complexity directly.

Good. "*Complexity: medium. The flow is new, but the components are reused. The novelty is the validation rules, which need to be derived from the legal team's spreadsheet.*"

Cost of skipping. Complexity drives spike work, prototypes, and the decision to research before estimating. Mis-estimated complexity is the standard reason for the same team estimating the same kind of work wrong three sprints in a row.

6. Database impact

Definition. What this does to the data. Five levels cover most of it. *None. Additive only. Schema change. Data migration. Query change.*

Bad. "*Adds a field.*" Where. To which table. With what default. Reversible how. Backfilled how. Read by whom. Indexed how.

Good. "*Database impact: additive only. Adds nullable column* `external_ref` *to* `orders`*. No backfill required. No new index for now (low cardinality, scans acceptable at current volume).*

Reversible by dropping the column. No downstream consumers depend on this column yet."

Cost of skipping. Schema and data are where reversibility goes to die. An unrecorded data migration is a data loss event with a friendlier name.

The most expensive ticket I have ever seen was a one-line description and a four-word AC. It triggered a schema rename that broke a reporting pipeline in another country. The cost was four engineers for two weeks, and a customer apology call.

---

7. API impact

Definition. What this does to a contract another team or another system depends on. Four levels. *None. Additive endpoint. Modified endpoint. Breaking change.*

Bad. API impact unmarked. The engineer adds a required field to a response. Three integrations break overnight. None of the three integrators were told.

Good. "*API impact: additive endpoint. Adds* `GET /v1/orders/{id}/timeline`. *Existing endpoints unchanged. No client migration required. Versioning untouched.*"

Or: "*API impact: breaking change. Removes deprecated field* `customer_email` *from* `GET /v1/orders/{id}`. *Migration notice was sent six weeks ago. Sunset date in this release. Three internal consumers updated. One external partner confirmed by email.*"

Cost of skipping. Breaking changes that nobody flagged are how trust between teams ends. Trust is hard to rebuild. The cost is not the change. The cost is being lied to by accident.

8. UI and customer impact

Definition. What the customer sees, and what they have to learn. Five levels are enough. *None. Minor tweak. New component. New screen. Redesign.*

Bad. "*Updates the page.*" Which page. What changes. Does the user notice. Does the user need to be told.

Good. "*UI impact: new component (a payment-method picker). Added inside an existing screen (checkout). Reuses tokens and patterns from the design system. No accessibility regressions in audit. Customer-visible: yes. Customer notification: not required; new option appears at next visit.*"

Cost of skipping. Customer-visible changes that the design system does not own become small visual debt. Small visual debt compounds. A year later, the product looks like five different products held together with tape, and nobody can point at the ticket that did it.

> **Myth.** Design review slows the team down.
>
> **Reality.** Design review slows down the *story*. It does not slow down the product. The product is the sum of stories. Stories that ship without design review are why redesigns happen every two years.

Five more, for when you are ready

There are five more dimensions you will eventually want to track. They are not as universal, and they should not be in the first version of your habit. But they exist, and they catch failures the eight do not.

Compliance sensitivity. Does this touch a regulated domain. Privacy, finance, health, accessibility. Three levels are enough: none, low, high.

Customer-facing impact. Is this internal-only, or visible to the customer. Internal changes can move fast. Customer-visible changes need a different review.

Reversibility. If you ship this and it is wrong, how hard is it to take back. Easy. Difficult. Irreversible. Irreversible changes earn the most process.

Data sensitivity. What kind of data does this touch. None, personal data, financial data, health data. Higher sensitivity earns higher review.

Workflow continuity. Does this story continue an open thread of work, or is it new ground. A continuation can reuse the planning of its predecessor. A new thread cannot.

These are the second album. Get the first eight working first.

The dimension shape, in practice

The eight do not have to live in a heavy template. They can live in a short block at the top of the description.

> Workflow: *feature* Modules: *Checkout (payment intent service), Notifications (receipt)* Scope: *single-module* Risk: *medium (behind a flag, additive migration)* Complexity: *medium (new validation rules, derived from legal spreadsheet)* DB impact: *additive only* API impact: *additive endpoint* UI impact: *new component on existing screen*

That is one minute of typing. It changes everything downstream of it.

When all eight are present, the team reading the ticket knows what they are looking at. The reviewer knows where to look. The AI agent knows what to do. The QA engineer knows what to test. The PO can defend the ticket in a meeting without re-reading it for ten minutes. The story is no longer a wish. It is a contract.

Refinement is the act of making the eight dimensions true. Everything else in refinement is theater.

The eight-dimension checklist

The same eight, written so a PO can scan them in under a minute before pushing a ticket. Use it once. After two sprints, the team stops needing the list because the list is what the team has already become.

- **Workflow.** Have you named what *kind* of work this is (feature, bug fix, migration, refactor, research spike, content update)?
- **Modules.** Have you listed the modules in business language, and named what this does *not* touch?
- **Scope depth.** Single file, single module, multi-module, or system-wide? One of those four words is on the ticket.
- **Risk.** If this goes wrong, can you roll it back? How long does the roll back take? Who feels it while it is wrong?
- **Complexity.** Is this ground the team has covered before, or is it novel? If novel, in which specific piece?
- **Database impact.** None, additive only, schema change, data migration, query change? One.
- **API impact.** None, additive endpoint, modified endpoint, breaking change? One. If breaking, who has been told?
- **UI and customer impact.** None, minor tweak, new component, new screen, redesign? Will the customer notice? Do they need to be told?

A ticket that fails any one of these is a ticket that should not move past refinement. The cost of failing the check on the ticket is minutes. The cost of failing it later is sprints.

A complete worked example

Take a real-sounding request. *Customers should be able to download a CSV of their invoices for any date range.* Run the dimensions against it.

Workflow. *Feature.* ***Modules.*** `billing/` *(invoice generation, already exists),* `accounts/` *(the role* `CustomerOwner` *is allowed to do this; no new role),* `data-export/` *(existing async export framework, already used for the customer-data export shipped last quarter). Does not touch:* `notifications/`, `auth/` *(no scope change beyond reuse of existing* `billing:read`*).* ***Scope depth.*** *Multi-module, contained within the three named above. No system-wide impact.* ***Risk.*** *Medium. Reversible by toggling the feature flag* `customer_invoice_export_csv`*. Customer impact during failure: the panel renders an error, no data corruption, no other billing surfaces affected.* ***Complexity.*** *Medium. The async export framework is reused (low complexity). The novelty is the CSV column set, which depends on the legal team's invoice schema (medium complexity, soft input).* ***Database impact.*** *None. Reads existing* `invoices` *table. No new column, no new table, no migration.* ***API impact.*** *Additive endpoints.* `POST /v1/customers/{id}/invoice-exports` *and* `GET /v1/customers/{id}/invoice-exports/{exportId}`*. Existing endpoints unchanged.* ***UI impact.*** *New component (a date-range picker and an export button) inside an existing screen (account settings, billing tab). Reuses existing date-picker, button, and progress-pill tokens. No new patterns.*

That is one minute of typing. It would have been ten minutes of guessing in the next sprint. The contract is now a contract, not a wish.

Myth. The dimensions are paperwork.

Reality. The dimensions are the smallest contract a team can sign before it builds something. Skip them and you are building from a sketch and a hope. Fill them in and the rest of the work pays them back, sprint after sprint, as work that does not have to be re-explained at a customer-support window.

What this looks like with the AI reading

If the AI is connected to the codebase, to the ticket system, and to the docs (the wiring covered later in *The System the AI Can Read*), the dimensions stop being a list the PO fills in by memory. They become a draft the AI lifts from the source. Modules come from the real module list. Database impact comes from reading the real schema. API impact comes from reading the real contract. Regression watch entries name files the AI saw in the dependency graph.

The PO's job changes shape. The PO no longer drafts the dimensions. The PO reviews the draft, corrects what is wrong, and decides what cannot be decided from code. The list above stays the same. The starting point is no longer blank.

What the eight do not solve

The eight close most of the easy guessing. They do not close all of it. There are still cases where a dimension answer depends on documentation that does not exist. There are still cases where a fork in the road appears, and no dimension can settle it. There are still cases where the work is too large to be a single story, even when every dimension is filled in.

We will get to each of those. The first one is the one most teams skip without noticing. If the ticket's affected module has no documentation, you are not refining the ticket. You are inventing the module on the fly. That is the next chapter.

The bridge

Filling in the eight dimensions only works if the context they refer to actually exists. If "*affected module: Notifications*" points at a module nobody has documented, the line is a label, not a fact. The same is true for stack rules and design system. Without docs, the eight dimensions collapse into eight more places to guess.

The next chapter is about that. *The Documents Beneath the Ticket.* It will sound like the slowest possible advice. It is the fastest possible advice, measured at the right horizon.

The Documents Beneath the Ticket

A four-week detour that saved four months

A team I worked with had a clean little ticket. *Build the billing settings screen.* The eight dimensions were even filled in. Workflow: feature. Scope: single-module. Risk: medium. Complexity: medium. UI: new screen. The PO had done the work the previous chapter asked for. The team was ready to build.

We did not build it. We spent four weeks writing three documents and resolving the boundary decisions those documents exposed. The team thought it was the slowest decision I had ever made. Four months later, when the screen shipped without rework and without scope debate, they thought it was the fastest decision I had ever made. Both reads of the same four weeks were correct, depending on which calendar you looked at.

This chapter is about the calendar most teams do not look at.

Docs-first is the slowest possible advice and the fastest possible advice. It depends on which Friday you measure.

What the ticket assumes that is not in the ticket

A ticket is shorter than the work because it points at things. *Affected module: Billing.* That points at a thing called Billing. *Risk: medium, behind a flag.* That points at a feature-flag system. *UI: new screen.* That points at a design system. *Reuses the existing customer record.* That points at a customer record.

If any of those things is not written down anywhere, the ticket is pointing at a guess. Specifically, three guesses, and they are the same three guesses every time.

There is a module the ticket refers to. Does it have a documented contract. Does the team agree what it is for. Does anyone own it.

There are conventions the work is expected to follow. Does the team agree on testing approach. On error handling. On logging. On where this kind of feature usually lives.

There is a user-facing surface the work has to look like. Does the team have shared design tokens. A shared component library. A shared accessibility floor.

If the answer to any of those is "kind of," the ticket is not refineable. The ticket is asking the team to invent the answers on the fly, on a deadline, while building. That is not refinement. That is improvisation with a Jira number.

Three docs and what they are for

There are three documents every ticket binds to, whether you write them down or not. The team has them or pays a tax in the dark. The work goes much better when they are in the light.

The module map. A short description of each module the product has, what it is for, who owns it, what its public contract is, and what it is *not* for. Business language, not folder names. If the engineer reads "Notifications," the engineer should see the same thing the PO sees, which is the same thing the founder sees, which is the same thing the new hire sees on day two.

The stack rules. A short list of conventions the team has agreed on. Which test framework. Which error envelope. Which queue for background work. Which logging style. Which response shape. Which naming. These do not have to be elaborate. They do have to be written. Conventions that live in one engineer's head are not conventions. They are folklore.

The design system. Tokens, components, patterns, accessibility floor. The shorter the list, the better. Five components used everywhere beat fifty components used somewhere.

These three sound boring. They are. They are also load-bearing in a way nobody appreciates until they are gone.

The team building the billing screen had no module map. They knew what Billing was, in a way. Billing was what the billing engineer worked on. *When the billing engineer left, the team lost the module. The ticket about billing was suddenly a ticket about archaeology. Four weeks to write down what Billing was, who used it, what its boundaries were. The fifth week, the screen began to build itself.*

What happens without docs

A team refining a ticket without docs is not refining. The team is reconstructing context from memory, every time, story by story. It looks like refinement because the meeting is on the calendar and the words sound right. It is not refinement.

You can see this if you listen for it. Listen for the phrases. *I think this lives in. I assume this is. Last time I touched it. We probably want. Let's just.* Each of those phrases is a doc that does not exist, paying its rent in the meeting.

Now multiply that meeting by the number of stories you refine in a sprint. Multiply that by the number of engineers in the meeting. You are losing hours per sprint to reconstructing context that should have been one document and a five-minute read.

In teams I have observed, a significant share of every refinement meeting goes to reconstructing what modules exist and what they do. The exact share matters less than the shape. The reconstruction is

---

invisible to the team because it has always been there. The reconstruction is the tax.

What happens with docs

A team refining a ticket with the three docs in place reads the ticket, reads the relevant module entry, reads the relevant stack rule, reads the relevant design tokens, and is ready to talk about the actual story. Refinement gets short, because the long part is already done.

Stories get refined in minutes, not hours. Engineers stop arguing about which queue to use, because the queue is in the rules. POs stop arguing about scope, because the module map says what belongs in the module. Designers stop redesigning the same modal three times, because the modal is already in the system.

The three docs do not slow the team down. They move the slow part to a place where it is cheaper. The slow part lives in the docs, once. The fast part lives in refinement, every sprint, free.

How to write the three docs without making a project of it

The most common reason teams skip docs-first is that they imagine it as a documentation project. It is not. It is a short, opinionated, one-pass write-up done by the team that already knows the answer. The shape of each doc, and the day-by-day recipe, both live below.

Total cost of writing all three from scratch, for a product that already exists in the team's head: a week or two. Total cost of not having them: every refinement meeting, every onboarding, every disagreement, until the team gives up and writes them anyway.

The team that writes them in week one ships in week six. The team that does not write them ships in month four, with three rewrites in the middle. Both teams paid for the docs. One paid in calm. The other paid in churn.

Myth. Docs slow the team down.

Reality. Docs slow the team down in the week they are written. They speed the team up in every week after. Most teams are still in the first week, repeating it.

A real module-map entry, in full

A module-map entry does not need to be long. It needs to be honest. Here is one for a real-shaped module, written in the form that has held up across products.

> ***Module: Notifications***
>
> ***Purpose.*** *Owns the transactional and operational messages the product sends to customers and to internal users. Email, SMS, in-app, webhook.*
>
> ***Public boundary.*** *Three things, and only three.*

1. `notifications.send(template, recipient, context)` *is the entry point. Other modules call this and never construct a message themselves.*
2. *The template catalog under* `notifications/templates/` *is the canonical source of every message the product can send. Adding a new message means adding a template here first.*
3. *Webhook subscriptions managed via* `POST /v1/webhooks` *are owned by this module. Other modules read webhook events; they do not configure them.*

Owner. *Kim. Backup: the engineering lead until further notice.*

What this module does not do.

- *It does not decide* when *to send a message. Other modules decide. This module sends.*
- *It does not own the user's notification preferences. That is* `accounts/`*.*
- *It does not handle in-product banners or toasts. Those are UI-only and live in the frontend.*
- *It does not own translation strings. The string lives with the template; the translation pipeline is owned by* `i18n/`*.*

Currently open work.

- `BACK-318` *migrating SMS provider, currently behind feature flag* `sms_provider_v2`*. Until that closes, both*

> *providers are wired and either can be active for a given account.*
> - `BACK-322` *re-defining the webhook contract version. Decision packet* `D-29` *open.*

The entry is short. The reader knows what is true today. The reader knows what is in flight. The reader knows who to ask. A new hire on their second morning can be handed this entry and asked to add a new transactional email, and the new hire will not have to ask three people which way is up.

Most teams have two or three modules whose entries write themselves in one sitting. The other ten or fifteen take a Friday between two engineers who have been arguing about them for a year. The Friday closes the argument permanently.

A three-docs-in-a-day recipe

A team that has decided to install the three foundational docs from scratch can do it in a day if they treat it as an opinionated pass, not a documentation project. The shape that works:

- **Morning, ninety minutes.** The team lists every module the product has. Two columns on a whiteboard, real or virtual. Module name. One-line purpose. Each entry under a minute. The list is messy. The list is fine. The list is the spine.
- **Morning, ninety minutes.** Split into pairs. Each pair takes three modules and writes the entry in the shape above. One owner per module is named, with a backup. Disagreements

get parked on a "decide today" list and called within the same day.

- **Afternoon, sixty minutes.** The stack rules. The team's two most senior engineers dictate the conventions they already enforce in code review. One bullet each. Testing, errors, logging, naming, background jobs, response shape, migrations, auth. Someone types. The rule for the list: every bullet has to be enforceable in code review by pointing at it. Vague bullets get rewritten or dropped.
- **Afternoon, sixty minutes.** The design system tokens. If a design system already exists, this is a link and a one-page accessibility floor (which WCAG level, which keyboard rules, which screen-reader expectations). If a design system does not exist, this is the team naming the four to six primary components and their tokens. The team that has been shipping for more than six months already has these; the document is the act of writing them down.

End of day: three documents. Total cost: one team-day, plus the parked decisions to be called within the week. The week after, refinement gets shorter every meeting, because the long part of refinement has moved into a place where the cost is paid once and the value is paid every time.

What docs-first does not mean

It does not mean every ticket waits for every doc. It means *the docs the ticket binds to* have to exist before the ticket is refined. If a story touches Billing and Notifications, the Billing and Notifications module entries have to be in the map. The stack

rules that apply have to exist. The design tokens it will use have to exist. The other docs are not blockers.

It also does not mean docs need to be finished. They need to be present and honest. A module entry that says "*this is the customer email layer; current owner is unknown; current contract is unclear; we are deciding the boundary in story BACK-321*" is a fine entry. It is honest. It tells the next person what they are walking into. That is the job.

It does not mean writing docs that nobody will read. It means writing the smallest thing a person can read in five minutes to feel oriented. If a doc is so long that it gets skimmed, it is not a doc. It is a wall.

When the docs are stale

Stale docs are worse than missing docs. Stale docs lie with confidence. The fix is not to write more docs. The fix is to make doc updates part of the work. When a story changes a module's boundary, the module entry is updated as part of the story. When a story adds a new convention, the stack rules are updated. When a story introduces a new pattern, the design system is updated.

This sounds like more work. It is, by ten minutes per story. The ten minutes is invisible. The cost of *not* doing them is two weeks of next sprint's planning being wrong because the docs lied. Ten minutes is a small price for a sprint that is not lying to itself.

The docs are right when updating them is the cheapest part of the work, and wrong when updating them is the part you forgot.

Honest docs need more than freshness

A doc can be up to date and still wrong. It can be wrong about access. About what data leaks where. About what a screen reader will read out. About whether two teams have quietly named the same concept differently.

Freshness is one lens. Real docs review themselves through several.

Once a quarter, take the module map, the stack rules, and the design system off the shelf and read them through a short list of angles. A *security* angle. *Who can touch what, and where do credentials live.* A *privacy* angle. *Which data leaves the device, which is logged, who reads the logs.* A *permissions* angle. *Which roles can do which actions, and what happens when a role is missing.* An *accessibility* angle. *What does this look like to keyboard-only and screen-reader users.* A *cross-doc consistency* angle. *Do the same words mean the same things across every page in the docs.*

Each pass is a different question against the same docs. Each pass finds something the others miss. A doc that has been read through all of them in a year is not just fresh. It is honest. The two are not the same.

The cost is a half day per quarter. The saving is the bugs you do not ship because the doc that pointed at them was telling the truth.

> **Myth.** A document is correct if it is recent.
>
> **Reality.** A document is correct if it is recent *and* reviewed against the angles a reader will actually use it for. Recency is a necessary condition, not a sufficient one.

A small habit to make docs-first stick

Make the docs the first thing the team looks at in refinement. The PO opens the ticket, then opens the module entry, the relevant stack rule, the relevant design tokens. Reads them out loud. Out loud is not optional. Reading out loud catches the places where the doc is wrong or stale or vague.

If the relevant doc is missing or wrong, the ticket is not refined. It is paused. The next thing the team does is write the doc. The story is back in refinement after the doc is in.

This sounds bureaucratic. It is the opposite. It is the cheapest possible bureaucracy. A doc that takes thirty minutes to write saves four hours of debate this week and four sprints of rework over the year.

What docs-first does for AI

AI agents reading the backlog can only be as accurate as the docs they read. Without docs, the AI averages the world. With docs, the AI reads your product. The same model becomes a different collaborator, because the input changes.

The chapter on *The System the AI Can Read* covers the wiring that makes this real. An AI connected to your codebase and your docs through MCP, or whatever tool the team already uses, reads the three documents above on every draft it produces. The module-map entry becomes the AI's first reference. The stack rules become the AI's review checklist. The design system becomes the AI's component palette. The same model that used to average is now reading from your shelf.

The three documents are the author's mind made legible. Without them, the AI guesses what the team was thinking. With them, the AI reads what the team decided.

This is not theoretical. Try the same prompt against the same AI tool, once with a sentence-long ticket and no module context, and once with the eight dimensions plus three docs the AI can read. The output is not slightly better. It is a different shape. The first is a paragraph. The second is a plan.

The reason AI is not making your team faster is rarely the AI. It is what the AI has to read. Most teams have not given it anything to read except their guesses.

A counter-objection

"This is fine for big stories. It is overkill for our day-to-day work."

Day-to-day work is most of what your team ships. Day-to-day work is where the docs prove themselves. The savings on big stories are visible. The savings on small stories are bigger, because there are more of them. Skip docs-first for small work and you will run a thousand small reinventions a year. Each one is cheap. The total is not.

The bridge

Docs in place. The team can now point at real things when refining tickets. The next question is who, or what, gets to read those docs and use them to draft the work. So far the docs have lived between people. The next chapter wires them into the system the AI reads, so a draft of any ticket starts from the team's truth rather than from a model's average.

The System the AI Can Read

Two runs of the same sentence

One founder ran a small experiment one afternoon. Same AI tool. Same ticket sentence. Two contexts.

The first run was the version her team used every day. The AI sat in a browser tab. It had the ticket text and nothing else. It produced a ticket back. The output looked like a ticket. Workflow, scope, a couple of acceptance criteria, a number that smelled like a story-point estimate. The team would have refined it without thinking.

The second run was a different setup. The AI ran on her laptop, with read access to the codebase and a connection to the company's Jira through Model Context Protocol (MCP), a standard way for an AI tool to connect to outside systems such as Jira, GitHub, or Confluence through controlled, permissioned

access. She typed the same sentence and asked for the same ticket. The AI read code first. Then it wrote.

The output was not slightly better. It was a different shape. The module list was the team's real module names, not folklore. The database-impact line referenced the real schema. The API-impact line referenced the real contract. The regression watch named two files the team had quietly forgotten were on the dependency edge. The acceptance criteria pointed at error codes that already existed in the project.

Same model. Same prompt. Different reader. The first output was an average of every login ticket ever written. The second was a draft against her system.

This chapter is about how to get the second one on purpose, and what it costs when you do not.

> *A ticket the AI wrote from context is not a smarter ticket. It is a ticket about your product, instead of a ticket about software in general.*

Why this gap is now expensive

The opening chapter walked through the research that complicates the simple story about AI and delivery. The DORA framing of AI as an amplifier (it magnifies the strengths of organized teams and the dysfunctions of struggling ones) is the

version that matters here. The amplifier runs on whatever the team feeds it. What the AI is allowed to read is most of what the team feeds it.

There are two reasonable conclusions to draw from that. The first is that AI is not ready for production. The conclusion is wrong. The second is that AI produces output proportional to what it is allowed to read. The conclusion matches the work. A team whose AI reads tickets in a vacuum is asking the model to invent the project. A team whose AI reads the codebase, the ticket system, and the docs is asking it to draft against the project. The two are different jobs.

This chapter is the wiring that changes which job the AI is doing.

What "context-aware" actually means

The first chapter named the failure mode: AI without context averages the world. The chapter on the eight dimensions named what a ticket has to settle. The chapter on documents named what the ticket points at. Each step moved the team upstream of the build. None of them changed where the AI reads from.

This chapter changes that.

A context-aware AI reads your real sources of truth before it writes. The eight dimensions get filled from reality, not from the average of all logins ever built. The module list is the team's real list. The risk note references the actual rollback path. The acceptance criteria reference real error codes in real files. The ticket is grounded.

Call the output a *system-aware ticket*. The phrase is not decoration. It is the difference between a ticket about a product and a ticket about your product.

The three surfaces

There are three surfaces the AI needs to read from to become system-aware. The tools change every six months. The surfaces do not.

The local codebase. The AI runs where the code lives, with read access. It can see which modules actually exist, what their public boundaries are, what patterns the team already uses, which helpers are reusable, which migrations are pending, which tests are weak. The eight dimensions stop being a guess. They become a read.

The ticket system. The AI connects to the place where work lives. The connection is a small piece of plumbing. The most common shape today is an MCP server bridging Jira (or Linear, or GitHub Issues) to the AI from the local machine. The principle is tool-agnostic. The AI reads tickets, reads epics, reads the last decision the team recorded about this module. It drafts the new ticket against that. It pushes the new ticket back into the system only after a person says yes. You stop managing tickets in a browser tab that has none of the context. You manage tickets from the place that has all of it.

The documents. Confluence pages, GitHub READMEs, decision records, design system docs. Same pattern, different source of truth. If the documents the team agreed to write in

the previous chapter live in Confluence, the AI is connected to Confluence. If they live in GitHub, the AI is connected to GitHub. The principle generalizes. Connect the AI to wherever your truth lives. Then the AI is reading your truth.

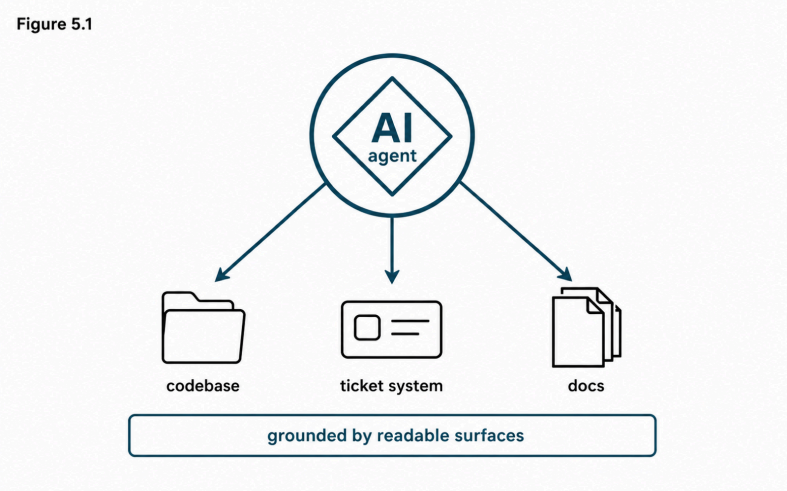

Figure 3: Figure 5.1. The three surfaces the AI must be allowed to read before it drafts: the wiring that turns an averaged ticket into one about your product.

The wiring, concretely

Most AI coding tools today (Claude Code, Cursor, Codex CLI, GitHub Copilot, Gemini CLI, Junie, Windsurf, Continue, Aider) accept the same two inputs. A thin entry file the tool reads on startup, and an MCP configuration that tells the AI which servers it is allowed to talk to. The names will change. The shape will not.

The entry file is small. It tells the AI where the project lives, which docs are canonical, which conventions matter, and which workflows the team uses. The most common names are

`CLAUDE.md`, `AGENTS.md`, `.cursor/rules/*.mdc`, `.github/copilot-instructions.md`, and `.windsurfrules`. The contents are similar across tools because the underlying need is the same. In practice, a focused entry file works better than a sprawling one. The goal is not to preload the whole company into the prompt. It is to point the AI at the places where the truth lives.

The MCP configuration is also small. It is a JSON file the AI reads to discover which servers are available. The team adds Jira here. The team adds GitHub here. The team adds Confluence here. Each server gets scoped credentials. Each server gets explicit permission to read, draft, and (where the team allows it) push.

Together, the entry file and the MCP config are the wiring. Two files. Usually small. For many projects, closer to a page than a manual. The wiring is small on purpose. Long configs lie about themselves and get ignored. Short configs survive.

The workflow, in one paragraph

You describe intent. The AI reads the relevant code, the relevant tickets, the relevant docs. It drafts a system-aware ticket. The draft includes the eight dimensions filled from code, the acceptance criteria referencing real error codes, the regression watch naming real files, the lane already implied by the dimensions. You review the draft. You correct what is wrong. You decide what cannot be decided from code. You approve. The AI pushes the ticket into the ticket system. Total time, for most tickets, is under ten minutes.

This is not the AI doing your job. It is the AI doing the part of the job that did not need a person to do it, so the person can spend that time on the part that did.

What the surfaces buy you

Each surface pays back into a chapter you have already read.

The eight dimensions chapter said the dimensions had to be true. With the local codebase connected, the dimensions get filled from ground truth. Affected modules are the modules that actually exist. Database impact is read from the real schema. API impact is read from the real contract. Scope depth is read from the real call graph. Refinement stops being a meeting where the team reconstructs context from memory. It becomes a review of a draft the AI lifted from the source.

The documents chapter said the ticket points at things. With the docs connected, the AI points at the real things. The module-map entry the AI reads is the same one the engineer reads. The conventions the AI applies are the same ones the reviewer applies. The design tokens the AI references are the same ones the designer maintains. The team and the AI read from the same shelf.

The decision chapter is two chapters ahead, but it earns from this one too. Decision records are something the AI can read. When a fork comes up that the team has already called, the AI proposes the called option. It does not re-invent the argument from training data. It reads what your team decided.

Myth. Connecting the AI to more tools makes it smarter.

Reality. Connecting the AI to more tools makes it grounded. Smartness is the model. Grounding is the wiring. Most teams have the model and not the wiring, then conclude the model is the bottleneck. The model is fine. The wiring is missing.

The honest caveats

Failure modes have to be named, because the failure modes are the ones a team learns expensively if it does not hear them first.

Context-aware is not decision-aware. The AI reading your code does not mean the AI understands what you are trying to do with it. The code shows what is true today. It does not show what should be true tomorrow. Product calls and forks still belong to people. The AI can surface a fork. It cannot call it.

More context is not better context. The instinct is to point the AI at everything. The instinct is wrong. Context flooding (loading more files than the model can reason over) produces high-confidence hallucinations faster than narrow context produces correct answers. In field practice, brittle context windows are a top cause of broken output, and sometimes less retrieval produces better results. The fix is not to give the AI everything. The fix is to give the AI structured retrieval against what it actually needs for the task in front of it.

Access has a blast radius. An AI with read access to your codebase, your tickets, and your docs is a new permissions surface. Apply the angles from the documents chapter. Who is the AI running as. What can it see. What is it logging. What happens when it is wrong. These are the questions a new senior engineer would have to answer on day one. The AI has to answer them too.

Stale context still lies. If the docs are out of date, the AI writes confident wrong tickets faster than a person could. A faster lie is still a lie. The fix is the one the documents chapter named: updates are part of the work, not a project. A system-aware AI on top of stale docs is worse than no AI on top of fresh docs. Freshness is not optional once the AI is reading.

The push step needs a human gate. The AI drafts. It does not create anything in the system of record until a person approves the push. The gate is short. The gate is non-negotiable. Auto-create makes the backlog grow faster than the team can think. The team's job is to think. The AI's job is to draft and to wait.

These are not reasons to skip the wiring. They are reasons to wire carefully. Every one of them has a cheap answer. The cheap answers have to be in place before the wiring is.

The same sentence, again

The book opened with a sentence. *Just add login.* The first chapter showed what AI averages when that sentence is all it has. This chapter shows what AI produces when the same sentence is read against a real system.

Averaged ticket, the version from the first chapter. Workflow: feature. Modules: authentication. Scope: multi-module. Risk: medium. Acceptance: user can sign in with email and password, user receives a confirmation email, errors are handled gracefully. Story points: 8. The output looks fine. It is a sketch of every login ticket ever written.

System-aware ticket, after the AI reads the codebase, the ticket system, and the docs.

> *Workflow: feature. Modules:* `auth/` *(existing OAuth helper present, currently used for the partner integration),* `sessions/` *(existing sessions table, twenty-four hour default, rolling refresh available behind a flag),* `notifications/` *(existing transactional template* `auth.password_reset`*, English-only). Scope: multi-module, contained within the three above. Database impact: additive only. New nullable column* `users.password_hash` *(currently null for OAuth-only users). Reversible by dropping the column. API impact: additive endpoint. New* `POST /v1/sessions` *accepting email and password. Existing* `POST /v1/sessions/oauth` *unchanged. Acceptance, positive: given a user with a password hash,* `POST /v1/sessions` *returns a session token. Acceptance, negative: given a user without a password hash, the endpoint returns error code* `OAUTH_ONLY`*. Acceptance, edge: given an invalid password, returns* `INVALID_CREDENTIALS` *with the team's existing*

> *rate-limit headers (see* `auth/middleware/rate_limit.ts`*). Regression watch:* `auth/session_factory.ts` *(shared between OAuth and the new path),* `notifications/templates/auth.*` *(only English exists; the new password-reset email needs the same English-only constraint or a translation ticket). Open fork: whether to enable rolling-refresh by default for the new path. Caller: PO with security review.*

The second ticket is the first one with the averages replaced by the team's real product. The eight dimensions are filled from files. The acceptance criteria reference error codes the codebase already uses. The regression watch names two specific files that would otherwise have surprised the team in week three. The fork is surfaced for a person to call, not buried in the implementation. The ticket is ten minutes of review away from being pushed.

The model did not get better between the two outputs. The reader did.

Teams that move from averaged tickets to system-aware tickets do not first report a feeling of speed. They report a feeling of fewer surprises. The acceleration is real. The acceleration is invisible because the cost it removes is the cost of being wrong, and the cost of being wrong is the cost the team had stopped seeing.

What this is and is not

It is a connection layer between an AI and the system it is supposed to think about. The AI you already use is the same. The team you already have is the same. What changes is what the AI is allowed to read before it writes.

It is also not a productivity tool. It is an accuracy tool. The productivity is the side effect. A team that runs system-aware tickets for a quarter ships less rework, fewer hot patches, fewer post-merge "I thought we agreed" arguments. The team did not get faster. The team stopped paying for being wrong.

DORA's framing returns here. AI is an amplifier. The team that gives a thoughtful upstream to a fast model amplifies the thoughtfulness. The team that gives a vague upstream amplifies the vagueness. The acceleration runs in both directions. The direction is chosen upstream.

> *The model is not the bottleneck. The bottleneck is what the model is allowed to read.*

A small habit

In refinement, before the team discusses a new ticket, the PO (product owner) asks the AI to draft it from context. Two minutes. The draft appears. The team reads it. The team adjusts

what is wrong, settles what is unsettled, names the fork if there is one. The PO approves. The AI pushes.

The team is doing the same refinement the team always did. The starting point moved. Instead of starting with a blank, the team starts with a draft that is already true about most of the work. The team's attention is now on the part that needs a person. That part is where the team adds value the AI cannot.

The PO writes the workflow, modules, scope, risk, and complexity lines once, on the first ticket. From then on, the AI fills them in by default and the team corrects. The five lines that used to be one minute of typing become ten seconds of reading. Refinement gets shorter. Refinement gets sharper. Both happen because the AI is reading the system instead of the void.

A counter-objection

"This sounds like a security risk we are not ready to take."

It is a security question, not a security risk. The same question every new engineer has to answer about access on day one. Who is the AI. What can it read. What can it write. Where do its actions get logged. What is the human gate.

Most teams answer these questions for the engineer without thinking. The same answers work for the AI, with the same care. The team that is not ready to give the AI access has not decided how access works at all. The decision is worth making for both.

A workable starting shape: the AI gets read access to the codebase, write access only to drafts, push access only behind

an explicit human approval, and an audit log of every action. The shape is small. It is the same shape any new contributor would get.

What the surfaces do not solve

The surfaces ground the ticket. They do not decide what the ticket is for. They do not pick which feature to build next, which customer to serve first, which trade-off the company is willing to make. Those calls belong to people. The AI surfaces the call. The AI never makes it.

The surfaces also do not replace the work the rest of this book is about. A team with system-aware tickets that skips the lane decision still picks the wrong process for the work. A team with system-aware tickets that skips the fork-naming habit still buries product decisions inside implementations. The surfaces are the wiring. The chapters around them are the habits. Both have to be in place.

The bridge

A team that gets here has a backlog the AI reads and a draft the AI writes. The chapters around this one are about what to do with that draft so the work it represents is worth doing. There is one chapter before any of that, though. It is the one most books skip and most teams need.

The next chapter is about why this work is worth the discipline at all. Why the careful ticket, the named fork, the small slice, the honest acceptance criterion are worth more than the time they

cost. The answer is not productivity. The answer is what the code becomes a record of, and what that record reflects back.

The Mirror Inside the Code

What stays after the meeting ends

The most useful thing about software is that it keeps going.

The meeting ends. The sprint closes. The branch merges. The person who wrote the code moves on. The code does not. It answers requests on a Sunday morning. It validates a form a stranger fills in on a phone in another country. It protects a number in a database that belongs to a person the team will never meet. It does this when the team is asleep, when the team has rotated, when the company has been sold, when the original founder no longer remembers writing it.

Code is not only something a person writes. It is something a person leaves behind.

That sentence is the whole reason for this book.

A backlog is a list of things to do. A backlog that thinks is a list of things you will be proud to have left.

The mirror

A codebase is a machine. Most teams agree on that part. A codebase is also a mirror. That part is less obvious, because the mirror is not in the syntax. It is in everything around the syntax.

The codebase reflects the thinking that created it. Rushed work shows up as rushed code. A vague requirement shows up as a guess the engineer had to make under deadline. An avoided product decision shows up as a generic abstraction that does not quite fit any of its three callers. A team that did not agree shows up as two helpers doing the same job differently in two files.

The code is honest about its conditions. It cannot lie. It can only run, and what it runs is what it was given.

When the conditions are good, the code shows that too. Clear thinking shows up as obvious naming. Settled decisions show up as the same pattern in three places without anyone noticing it is a pattern. Honest acceptance criteria show up as tests that fail when the behavior is wrong, instead of when the implementation moved. Care for the next reader shows up as a function that does one thing and a comment that explains the one thing the function cannot show on its own.

Two teams can ship the same feature. One team's feature, six months later, is the foundation the next feature builds on. The other team's feature, six months later, is the thing nobody wants to touch. The features look the same to the customer. The codebases do not look the same to the team. The mirror remembered.

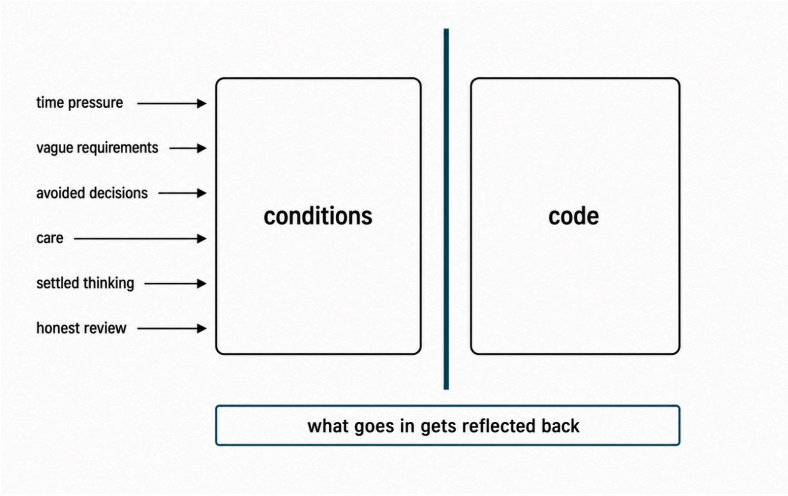

Figure 4: Figure 6.1. What goes into the work shows up in the code: the mirror is not a metaphor, it is a forecasting tool.

AI amplifies the mirror

The 2025 DORA report says it plainly: "AI's primary role is as an amplifier, magnifying an organization's existing strengths and weaknesses." The acceleration is real. The direction it runs in is whichever direction the team is already pointed.

This is why the discipline matters now in a way it did not five years ago. The cost of writing a vague ticket used to be paid by

the engineer who had to interpret it. The cost is now paid by the AI that builds against it, the reviewer who has to catch what the AI got wrong, and the customer who hits what nobody caught. The mirror is faster. The discipline is the same.

A short lineage

The idea that code reflects the people who made it is not new. It is older than the languages most teams use.

Conway, writing in 1968, made the structural version. Systems mirror the organizations that build them. Two teams produce a two-part system. The shape of the org becomes the shape of the product. The observation has aged into a law because no one has managed to falsify it in sixty years.

Brooks, in *The Mythical Man-Month*, wrote about the inner version of it. He described software as a medium made almost entirely of thought. The medium is thought made stuff. Anything you put into the thought ends up in the stuff.

Knuth, on literate programming, gave the same idea a writing-craft frame. Programs are written for humans to read. They are only incidentally for machines to run. The audience on the other end of the code is a person.

The Pragmatic Programmer named the negative version. *Broken windows.* A codebase with one neglected corner invites more neglect. The neglect is contagious because the message of neglect is contagious. The code remembers what the team tolerated, and the team's next decisions follow the code's example.

Four writers across four decades, one shared observation. Code is a record. The record is of the people. The people are reflected back when they read it.

This book is the practical version. The mirror is sharpest at the start of the work, in the backlog, where the conditions get set. Everything downstream is the conditions playing out. If the conditions are clear and owned, the code becomes a record of clear, owned work. If the conditions are rushed and vague, the code becomes a record of that. The code did not get to choose.

What the code remembers

The code remembers the Tuesday the ticket was written in fifteen minutes because the founder wanted to demo on Thursday. It remembers the assumption the engineer made when the PO did not answer the Slack thread. It remembers the fork that was decided by reflex in the implementation instead of by name in refinement. It remembers the acceptance criterion that was vague because writing it precisely would have meant stopping the sprint to think.

It also remembers the opposite. It remembers the half day the team spent writing the three documents that nobody wanted to write. It remembers the fork the team froze for thirty minutes to make a real call. It remembers the acceptance criterion that was specific enough to fail loudly when the behavior drifted. It remembers the slice that was small enough to finish, the review that was thorough because the diff was reviewable, the doc that was updated as part of the work instead of “later.”

This is not poetry. It is the operating reality of every product older than three months.

The reframe

Every chapter in this book is about discipline. The eight decisions. The three documents. The system the AI reads. The lane that fits the work. The fork that gets named. The slice that finishes. The criterion that is testable. The estimate that compounds.

Each one is small. Each one costs minutes. Each one is the kind of thing a team can skip on a Tuesday and not notice the cost until a Thursday three months later. Most teams skip them. Most teams have a reason. The reason is usually a real reason. The reason is usually true. The reason is also usually the wrong horizon.

The reframe in this chapter is this. The discipline is not about productivity. The discipline is about what the code becomes a record of.

The eight decisions are the author refusing to leave a guess inside the work. The three documents are the author leaving a true map for the next person, including future-self. The system the AI reads is the team refusing to let the AI average over their product. The named fork is the team refusing to let the code remember an avoidance. The honest acceptance criterion is the care the next reader of the code is owed. The plan-versus-actual loop is the team refusing to forget what the work taught them.

Read in this light, the chapters of this book are not a productivity system. They are a small set of habits for making sure the

reflection in the mirror, when you finally stop to look at it, is one you are proud of.

> *Code is the form your judgment takes after the meeting ends. The meeting is forgotten. The judgment is what runs on Sunday morning.*

Why this is not sentimental

There is a version of this chapter that would be sentimental. The version where craft is sacred and software is art and the engineer is a craftsman whose hammer is a keyboard. That version is fine. It is also not this book.

The version in this book is more practical and more selfish. A team that builds in conditions of clarity and ownership ships product that holds up. The product holding up is the team's reputation, the team's morale, the team's Friday afternoon, the team's runway, the team's belief in their own work. None of that is metaphysical. All of it depends on the same thing. The conditions upstream of the build.

The mirror is not a sermon. It is a forecasting tool. You can often look at a team's backlog and predict, with embarrassing accuracy, what the codebase will feel like months later. The backlog is the photograph the codebase will be developed from. If the photograph is blurry, the print will be blurry. If the photograph

is sharp, the print can be sharp. Nothing about the camera or the paper changes the fact of the photograph.

The discipline is in service of the team that has to live with the code, the customer who has to use what the code becomes, and the next hire who has to read the code on day two and feel oriented instead of lost.

> **Myth.** Craft and speed are opposites.
>
> **Reality.** Craft, at the backlog layer, *is* speed. The craft is upstream of the build. The build is what most of the team's time is spent on. A clear upstream cuts the build time more than any tool. The teams that ship fastest are the teams whose upstream is the most boring.

What this means for AI

The previous chapter wired the AI into the system. This chapter is about what the AI is now wired into.

If the team has built a backlog that thinks, the AI reads thought. It reads the team's decisions, the team's documents, the team's eight dimensions, the team's acceptance criteria. It produces work that reflects the team. The AI is, in a real sense, a new reader of the team's mirror. What the AI generates is what the mirror was already showing.

If the team has not built a backlog that thinks, the AI reads guesses, and produces work that reflects the guesses. The output is competent and generic. The competent generic output runs for

years. The team that has to maintain it is the one that pays for the generic.

What is left

There is a moment, sometimes years after a sprint, when the engineer who wrote a piece of code is still at the company, and a customer says something offhand about how easy a particular flow is to use. The engineer remembers building the flow. The engineer remembers the Tuesday the fork came up. The engineer remembers the half hour the team spent freezing the ticket to make the right call. The customer does not know any of that. The customer just knows the flow works.

The mirror, in this moment, is not a metaphor. It is the customer's experience pointed back at the engineer's care. The care was paid for upstream, weeks earlier, by people who were tired and would have rather skipped it. The reward is here, now, in a sentence the customer said without knowing what they were rewarding.

That is the reward. Not a metric. Not a launch. The product still working, for someone, in a way that traces back to a decision made well by a person who is no longer in the meeting where it was made.

This is why a backlog is worth thinking about. The list of things in the backlog is the list of things the team will be the author of. The list looks like a queue. It is not a queue. It is a draft of the record.

Every story in the backlog is a small claim about who the team wants to have been when the work is read back.

A small habit

Once a month, the team takes thirty minutes to read a piece of code they shipped six months ago. Not to fix it. Not to refactor it. Just to read it. To notice what the past version of the team left for the present version of the team. To notice what was carried forward honestly, and what was carried forward as a guess.

The habit is uncomfortable for one or two readings. After that, it becomes a thing the team looks forward to. The team starts catching itself, in the present, leaving something the future-team will be glad to inherit.

Most teams do not look in the mirror. The mirror keeps working anyway.

The bridge

If the work the team puts into a story shows up in the code, then the care has to be proportionate to the work. A typo fix on the marketing site does not need the same care as a payment migration. A throwaway internal tool does not earn the same discipline as a customer-facing flow that will run for years. Treating everything the same is its own kind of carelessness.

The next chapter is about that proportion. The work the team gives to a ticket should match what the ticket is. There are three lanes. Picking the wrong one is most of how teams hurt themselves, in either direction. The lane is how the mirror gets the right exposure.

Part III

The Workflow

With the foundations in place, the work has to flow through them. The four chapters in this part are the operating mechanics. The process the work deserves (the three lanes). The decision before the code (forks). The work small enough to finish (slices). The moment done becomes provable (acceptance criteria as tests).

These are the chapters teams quote at each other in standups after a sprint or two of practicing them. The vocabulary is the point. The vocabulary is what makes the habits transferable.

The Process the Work Deserves

A thirty-minute fix that took a week

One founder had a single engineer who could do anything in thirty minutes. The engineer was very fast and slightly bored. The team had one process: every ticket went through full refinement, design review, spec sign-off, and a forty-five minute review meeting on Thursday. The thirty-minute fixes took a week, because the process was the work.

Six months later, the founder ran the opposite experiment. Same engineer, same product, no process. Move fast. Ship. Iterate. The fast lane became the only lane. Six weeks in, a small refactor in the auth module quietly broke single sign-on for one of three customers. The customer churned. The founder learned that the absence of process is also a process. It is the process where the cost arrives late, with a customer attached.

Neither extreme worked. Both teams were running the same lane on every ticket, and the lane fit none of the tickets. The fix is not a third extreme. The fix is three lanes. You pick the lane the work earns. The lane decides the process.

> *The most expensive process mistake is not using too much process or too little. It is using the same amount for everything.*

Why one lane is wrong

The teams that struggle most often have one lane. They picked it once, three years ago, when the team was four people. They never reviewed it. The team grew, the product grew, the customers grew, and the lane stayed the same. The lane is fine for the work it was designed for. It is a nightmare for the work it was not designed for.

A typo fix on the marketing site does not need a spec. A schema migration touching customer balances does need a spec. A team that runs both through the same process is paying twice. Once in dead time for the typo. Once in unfound risk for the migration.

The solution is not more process or less process. It is *the right amount of process for the work in front of you.*

The three lanes

There are three lanes. They are not abstract. They are decisions about which phases the work goes through.

Fast lane. For trivial or low-risk work. Six phases. Classify the request. Load the relevant docs. Implement. Review the implementation. Run verification. Update the docs. Hours, sometimes less.

Graduated lane. For medium work. Most feature work. Most non-trivial bug fixes. Ten phases. Classify. Docs load. Analysis. Sequence planning. Specification. Spec review. Implementation. Implementation review. Verification. Doc update. Days, sometimes a sprint.

Full lane. For high-risk or architectural work. Eleven phases. The same as graduated, plus an explicit user-flow phase between specification and spec review. Migrations, breaking API changes, redesigns, anything where reversibility is hard. Sprints, sometimes a quarter.

The lanes are not abstract grades. They are the actual list of phases the work goes through. Picking a lane is picking a list.

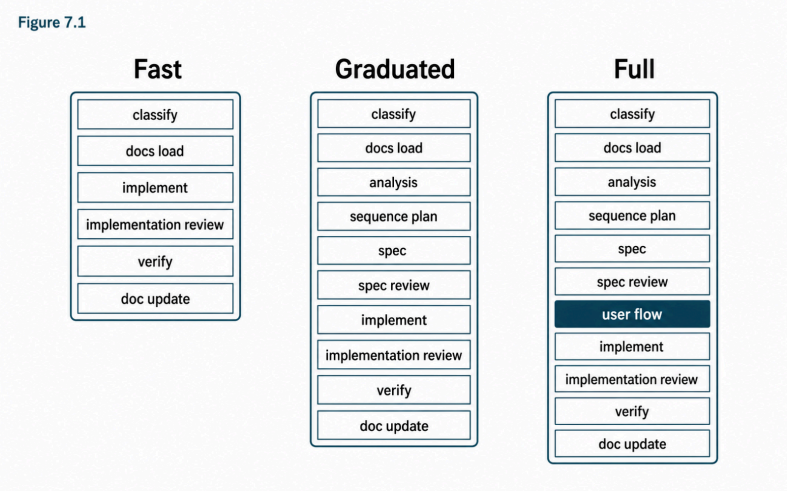

Figure 5: Figure 7.1. The three lanes shown to scale: picking the lane is picking the list.

How to pick a lane

The lane is not a feeling. The lane is a decision rule, and the rule reads off the eight dimensions you already filled in.

A ticket gets the fast lane when *all* of these are true:

- Scope: single-file or single-module
- Risk: low
- Database impact: none or additive only
- API impact: none or additive endpoint
- UI impact: none or minor tweak
- The affected module is documented and healthy
- No fork-in-the-road decision is pending

A ticket gets the full lane when *any* of these are true:

- Database impact: schema change or data migration

- API impact: breaking change
- UI impact: redesign
- Risk: high
- Reversibility: difficult or irreversible
- Compliance sensitivity: high **and** the change is hard to reverse or reaches many customers at once
- Data sensitivity: financial or health

Compliance sensitivity on its own does not force the full lane. A change can touch personal data and still be easy to take back: shipped behind a flag, rolled out to a few customers first, with a privacy review as a gate before wider rollout. What earns the full lane is high sensitivity *combined with* hard reversibility or broad reach. High sensitivity plus a narrow rollout and an easy rollback may remain graduated work with a privacy gate, rather than automatically full-lane work.

Anything else is the graduated lane. Graduated is the default. Most work is graduated, because most work is medium work.

The lane decision table

The rule above is prose. The same rule, as a table the PO can scan in five seconds during refinement:

Dimension	Fast	Graduated	Full
Scope depth	single-file or single-module	multi-module	any
Risk	low	medium	high

Database impact	none or additive only	any other	schema change or data migration
API impact	none or additive	modified endpoint	breaking change
UI impact	none or minor tweak	new component or screen	redesign
Reversibility	easily reversible	difficult	irreversible
Compliance sensitivity	none	low; or high if easily reversible (add a privacy gate)	high **and** the change is hard to reverse or reaches many customers at once
Data sensitivity	none	PII	financial or health
Fork pending	no	no	no (forks must close before any lane runs)

Read the table column by column. A ticket gets the fast lane only when *every* dimension in the fast column is true. A ticket

gets the full lane when *any* dimension in the full column is true. Everything else is graduated.

A worked override

The rule above is the floor. Instinct can move a ticket *up* a lane (toward more process). Instinct cannot move a ticket *down* (toward less). Here is what an honest override looks like on a real ticket.

> *Ticket:* Add a small banner on the dashboard explaining the scheduled maintenance on Saturday.
>
> *By the table: fast lane. Scope is single-file (one component). Risk is low. No DB, no API, no compliance, no PII. The change is a content addition.*
>
> *Override:* Moved to graduated. The dashboard component had two unrelated regressions in the last two sprints. The module is on the watch-list. Until module health recovers, even content changes get the graduated lane (analysis, sequence plan, spec review).
>
> *Signed: PO, with engineering-lead acknowledgement.*

The override is one sentence. The reason is one sentence. The next time the team looks back at the ticket, the reason is in plain view. Three months later, when the dashboard component is healthy again, the team can let similar tickets fall back to the fast lane with the same one-sentence justification, in reverse. Override is not nervousness. Override is named judgment.

Myth. A senior engineer can tell which lane a ticket belongs in by reading it.

Reality. A senior engineer can tell which lane a ticket belongs in by reading *the eight dimensions and the docs*. The reading is fast. The dimensions and docs are not skippable. Without them, the senior engineer is also guessing.

What each lane buys you

Each lane has a purpose. Skip the purpose and you skip the savings.

Fast lane buys speed. It is the absence of process when process would be theater. A typo fix on the marketing site, a copy change in a transactional email, a small bug in a stable module with a clear root cause. These do not need spec review. Spec review on these is a tax on the engineer's attention. It is theater dressed as process.

Graduated lane buys shared understanding. The analysis, the sequence plan, the specification, the spec review. These are not bureaucracy. They are the team making sure the work in front of them is the same work in every head. The cost is a few hours. The value is the absence of "I thought you meant" three weeks later.

Full lane buys reversibility. The user-flow phase, the design review, the spec sign-off, the regression watch. These are the explicit price of work that is hard to take back. The team treats

it as expensive on purpose, because the alternative is treating it as cheap and finding out it was not.

In one product team, introducing lane sizing visibly shed process from work that did not need it and added a small amount of process to work that did. The exact ratios matter less than the shape of the change. Net, the team got faster, and the things that went wrong got smaller.

What skipping the right lane costs

The cost is asymmetric. Each direction of mistake costs differently.

Fast lane applied to graduated work. This is the most common mistake. The team treats a real feature as a small fix. They skip the analysis, the spec, the spec review. The engineer builds the obvious thing. The obvious thing turns out not to be the thing the PO meant. Two weeks of rework. Filed under "polish" or "scope change," but it is the same thing. The rework is the missing phases, paid late.

Graduated lane applied to fast work. Less common, but real. The team treats a typo fix as a feature. The typo fix goes through three meetings. The engineer's morale drops. The team starts skipping process for everything to compensate. Six months later, the team has no process and is back to fast-lane-on-everything. The pendulum did not solve anything. It just changed direction.

Full lane applied to fast work. The bottleneck case. Important changes wait behind small ones because the queue is the same queue. The team starts batching changes to make the queue worth it, which makes each batch larger, which makes each batch more dangerous. The cost is hidden, but it is paid in throughput.

Fast lane applied to full work. This is the dangerous mistake. A migration treated as a quick fix. A breaking change shipped without consumer notice. A redesign merged without design review. The team gets away with it for a while, because the cost arrives late, in support, in churn, in the customer call. The team's confidence builds, the lane mistake gets habitual, and the eventual incident gets larger.

The asymmetry is important. Over-using process is annoying. Under-using process is dangerous. A team should be slightly biased toward the slower lane when in doubt. Slightly. Not always. Bias.

How lane selection looks in practice

In a healthy team, lane selection happens during refinement. The PO writes the eight dimensions. The team looks at them. The lane is read off, not debated. Debate happens about the dimensions, not the lane.

When a ticket arrives mid-sprint, lane selection happens before the engineer touches it. Two minutes, maybe five. Look at the eight dimensions. Apply the rule. Pick the lane.

Lane selection is not a meeting. It is a habit. The habit is what makes the rest of the process light.

> *One team put the lane decision on a sticky note on the ticket. Three colors. Green for fast. Yellow for graduated. Red for full. The PO put the color on at refinement. The engineer trusted the color. The reviewer trusted the color. The whole team's process anxiety dropped within two weeks. They had not added anything. They had named what they were already doing, and named it consistently.*

A note on documentation work

Documentation, research, and project questions get their own short lanes. They look like fast lanes with different phases. Documentation has its own ordering (the chapter on docs-first walked through it). Research and questions skip implementation entirely. They produce a document, not a feature.

A team that mixes feature lanes with documentation lanes ends up doing neither well. Documentation gets treated as a side quest. Research gets treated as the team's coffee break. Both are real work. Both need their own lane logic. Keep them separate.

When the lane changes mid-work

Sometimes a ticket starts in one lane and earns a different one during the work. The engineer pulls a fast-lane ticket and discovers the change is wider than the dimensions said. The right

move is to stop, re-assess the dimensions, and either re-lane the ticket or split it.

This is not failure. It is the lane system working. The fast lane is allowed to fail safely, because failure means the work earns more process. The full lane is not allowed to fail to the fast lane. A full-lane ticket cannot collapse into a fast-lane ticket because the work feels small. If it feels small, the spec was wrong, not the work.

> *The lane is a contract with future-you. The contract is: when this work is done, I will have used the process the work needed, not the process I wished it needed.*

What lanes do for AI

When AI agents are given lane information, they get a built-in scope contract. A fast-lane ticket tells AI to keep the change tight, skip the analysis pass, and stay inside the module. A graduated-lane ticket tells AI to plan first, write a spec, and ask for review before implementation. A full-lane ticket tells AI to refuse to implement until the spec is signed off and the user flow is in place.

Without lanes, AI either over-builds (proposing a spec for a typo) or under-builds (going straight to code for a migration). Both are wrong in different directions. The lane gives AI the same scope contract a senior engineer would carry in their head.

A counter-objection

“This sounds like a lot of process for a small team.”

It is the opposite. A small team has the least capacity to absorb a mistake. A small team running fast-lane-on-everything is one wrong ticket away from a Friday they will not forget. A small team that names the lane on every ticket pays five minutes a story and gets a Friday they will forget. Lane sizing is cheaper for small teams, not more expensive.

The bridge

Even with the right lane, even with the docs in place, even with the eight dimensions filled in, you will hit tickets where a single decision in the middle blocks everything. The decision is not implementation. It is a product or architecture choice in disguise, hiding inside what looked like a story. Trying to push past it gets the wrong answer fast.

The next chapter is about that. The fork in the road. There is a rule for it, and the rule is the difference between sprints that compound and sprints that unravel.

The Decision Before the Code

"I will just make it generic"

There is a sentence that an engineer says in standup, and every time I hear it, I feel a small pang in the part of the brain that has paid for it before. *I will just make it generic, and we can specialize later.*

Two months later, the generic thing has three callers. Each caller needed a different specialization. Nobody can change the generic thing without breaking at least one of the three. The thing that was going to be generic has become permanent and brittle at the same time. The engineer who built it has moved on to a different ticket. The team has inherited a small monument to the moment a fork in the road was treated as an implementation detail.

The fork was not an implementation detail. The fork was a product decision wearing a hoodie.

The most expensive choices in your product do not look like choices. They look like code.

What a fork is

A fork is a decision inside a ticket that has consequences outside the ticket. The ticket is about login. The fork is: do we use the existing authentication library or write our own thin layer. The ticket is about the checkout screen. The fork is: do we put the payment-method picker in a modal or on its own step. The ticket is about a new permission. The fork is: does this become a column on the user table or a separate ownership service.

None of those are implementation. All of them look like implementation. They are the moments where the engineer is one Slack reply away from making a call that the next three sprints will have to live with.

If the engineer makes the call alone, in code, on Tuesday, by reflex, the team owns the consequence. The team will say *we should have talked about it.* The team is right. The fork was a product decision, and product decisions do not belong inside an engineer's flow state.

Why forks hide

Forks hide because they have two costumes. The first costume is *implementation detail.* The second costume is *we already decided this.*

The implementation-detail costume is the most common. The engineer thinks the decision is small. The decision is small in the moment. The decision is large at the horizon. The team finds out at the horizon, not in the moment.

The we-already-decided-this costume is the more dangerous one. The engineer remembers a Slack thread from six weeks ago, in which the team said something like "*we will probably go with X.*" The thread did not decide anything. The thread was a vibe. The engineer treats the vibe as a decision because there is a ticket on the desk and a deadline on the calendar. The decision goes into the code. The vibe becomes architecture.

> **Myth.** Senior engineers spot the forks and handle them.
>
> **Reality.** Senior engineers spot some of the forks. The rest get coded around. The team finds them in the rear-view mirror, with a customer attached.

The five forks you will see

Forks come in shapes. Once you can name the shape, you can spot the fork.

Reuse versus new. There is an existing component that covers most of what you need. Reuse it and extend it, or write a new one tailored to this case. This fork hides as a refactor question. It is not. It is a product question: how generic is this category of feature in our product, and how often will the next ticket need the same shape.

Two architectures. Sync or async. Monolith or service. Library or framework. Server-side rendered or client-side. Each one of these is a five-year choice. Each one of these gets made in a single afternoon when nobody is watching.

Two user-experience paths. Modal or page. Wizard or single screen. Inline edit or separate edit form. Drawer or full takeover. Each shape teaches the user something about your product. The shape is part of the product. The shape is not part of the ticket.

Shared abstraction. This ticket is the third time we have written this pattern. Do we extract the pattern into a shared piece, or keep writing it three more times until the pattern is obvious. The cost of extracting too early is brittleness. The cost of extracting too late is duplication that has diverged in ways the team has stopped tracking. Either decision is defensible. Skipping the decision is not.

Workflow or tool change. We need a new service to do this. We are about to add a new dependency. We are about to introduce a new pattern the team has not used before. These are not "use this library." These are *a new piece of the team's vocabulary.* The team will live with the new vocabulary forever. The ticket should not introduce it by accident.

What to do when a fork appears

There is a rule. The rule is short. The rule is the chapter.

When a fork appears, you name it, you freeze the work, you get the call from the right person, you record the decision, and you resume. The fork does not get decided inside implementation. It

gets decided *above* implementation, by the people whose horizon includes the consequence.

That is the rule. Everything below is the operating manual.

How to name the fork

You write it down in a small, structured record. Call it a *decision packet*. The packet has a stable identifier (D-{N}), so future tickets can reference it. The packet is short on purpose. Eight lines is enough.

> ***D-42.*** *Should the payment-method picker live in a modal or replace the checkout screen with a step?*
>
> ***Category.*** *UX shape with downstream architecture cost.*
>
> ***Options.*** *1. Modal. Fast to ship. Keeps the user in the checkout context. Hides on small screens. 2. Separate step. Slower to ship. Gives more room for future methods. Accessible-first.*
>
> ***Recommendation.*** *Option 2 (separate step). We expect at least two more payment methods in the next two quarters. The modal will hit small-screen issues we cannot easily redesign without breaking the rest of checkout.*
>
> ***Confidence.*** *Medium. The future-method assumption is the soft part of the recommendation.*
>
> ***Caller.*** *PO, with design lead. Engineering lead if architecture cost is in question.*

> ***TTL.*** *Six months. Revisit if the payment-method roadmap changes.*

This takes the engineer eight minutes to write. It saves the team between two days and three months, depending on which version of "later" we are talking about.

The fields are deliberate. Identifier, so the packet is referenceable later. Category, so the team can spot patterns in which kinds of forks recur. Options, so the alternatives are visible and not just the chosen one. Recommendation, so the engineer's instinct is preserved. Confidence, so the caller knows where to push back. Caller, named as a person, so the packet routes to a human and not a void. TTL, so the team revisits the decision before it goes stale.

A team that writes packets in this shape ends a quarter with a small library of references. The library is the team's accumulated judgment, in a place that survives turnover. The next time a similar fork shows up, the team opens the library before opening the argument.

How to freeze the work

The ticket does not move forward until the fork is resolved. This is the part teams resist. Freezing feels like failure. Freezing is not failure. Freezing is the team telling itself the truth about what it has in its hands.

The freeze can be short. Often it is short. The PO sees the fork written down, walks five meters to the design lead's desk, comes

back with the call inside thirty minutes. The fork is resolved. The work resumes.

Sometimes the freeze is longer. Sometimes the right person to make the call is in a meeting until Thursday. The team works on a different ticket. The freeze costs a day or two. The freeze does not cost two months of building the wrong thing.

The freeze is cheap because it is short. The freeze is also cheap because it is *expected*. Teams that name forks as a habit get used to small freezes. The freezes become a normal part of the rhythm. The rhythm stays steady.

In one product team, six weeks of naming and freezing forks surfaced more of them than the team expected. Most resolved within a day; a few within a week; one resolved itself by revealing that the ticket was actually two separate tickets, both smaller than the original. In none of those cases did the team have to undo a decision later. The freeze paid for itself in the first month.

How to record the decision

Once the fork is called, the decision goes into a short record. The same five lines as the fork note, plus the call and the reasoning, plus a date and the caller's name. The record lives with the module it concerns. The next time a similar fork appears, the team checks the record first.

The record is not a contract for life. It has a *time-to-live*. Decisions go stale. The right way to do auth in 2024 is not the right way to do auth in 2026. Each decision record has a date and a soft expiration. When the expiration passes, the record is reviewed. Reviewed does not mean changed. It means looked at, with the question: is this still right.

A team that builds up a small library of fork-decision records is a team that does not have the same argument twice. That is most of what process is for. Stopping the same argument from being free.

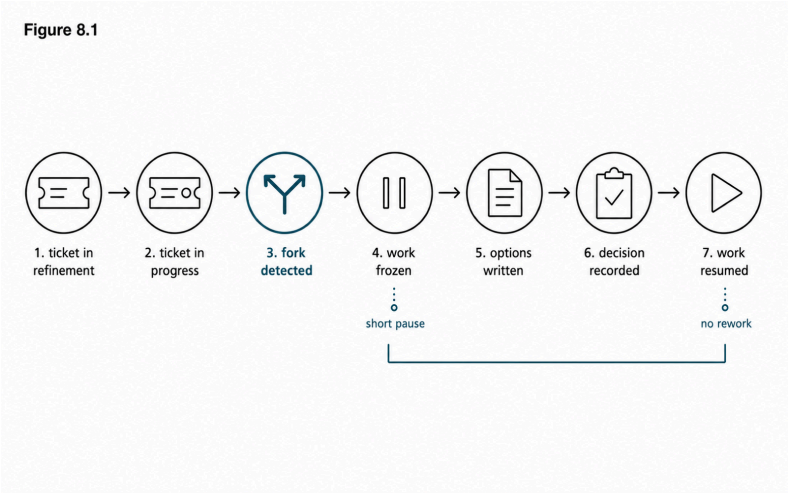

Figure 6: Figure 8.1. A fork named, frozen, called, and recorded: the cost of pausing is small; the cost of deciding silently is what arrives later.

What forks do for AI

AI is good at producing the average answer to a fork. The average answer to a fork is the most common version of each option,

written confidently. The average answer to a fork is also the most expensive way to use AI in your team.

An AI connected to your team's decision packets (the wiring covered in *The System the AI Can Read*) does not produce average answers. It produces *aligned* answers. If the team decided last quarter that we use a separate step rather than a modal for new flows, the AI proposes a separate step. If the team decided we extract shared patterns at the third occurrence, the AI proposes extraction at the third occurrence. The AI is no longer guessing your product. It is reading it.

An avoided fork is an avoidance the code remembers. The packet is the team's small refusal to let that happen. The library of packets is the team's accumulated refusal, in a shape future-team and AI can both read.

This is one of the largest changes you can make to how AI participates in your team. Decision packets turn AI from a generic assistant into a team member with memory. The memory is yours. AI did not invent it. AI remembered it because you wrote it down.

A counter-objection

"This sounds like we are creating a bureaucracy for what used to be conversations."

The conversation is fine. The conversation is not the problem. The problem is that the conversation does not survive into next quarter, and the next quarter has the same conversation again. Decision records are the part of the conversation that lasts. The

conversation can still be in Slack. The decision lives somewhere durable.

You will be surprised how much your team is repeating arguments. Most teams are. The arguments are interesting. The repetition is exhausting. The fix is not to stop arguing. The fix is to stop forgetting the resolution.

The shape of the rule

Name the fork. Freeze the work. Get the call from the right person. Record the decision. Resume.

Five steps. Each one is cheap. Skipping any of them is where the cost begins.

The hardest step for most teams is the freeze. The freeze feels expensive in the moment because the engineer is *right there* about to write the code. The freeze feels cheap in the rear view because the code, written without the freeze, was wrong. Train the team to associate freezes with safety, not with delay. The first three freezes are awkward. The fourth is normal. By the eighth, the team notices forks without having to be told to look.

> *A team that freezes well is a team that ships without surprises. A team that does not freeze ships with surprises and calls them roadmaps.*

When the fork cannot be resolved

There are forks where the team genuinely cannot make the call yet. Not enough information. Not enough customer signal. Not enough certainty about where the product is going. The right move in this case is not to guess. The right move is to *defer the work*, not the decision.

The ticket goes back to the backlog. The fork goes into the open-decisions list. The next ticket worked is one that does not depend on this fork. When new information arrives, the fork comes back, gets called, and the deferred work moves to the front.

This is uncomfortable because the original ticket felt important. Important is not the same as *ready*. A ticket that depends on an unmade decision is not ready. A ready ticket is one that can be built today against settled ground.

The deferred ticket does not punish anyone. It protects everyone, including the engineer who would have spent two months building the wrong thing.

The bridge

When the fork is named, frozen, called, and recorded, the work has a clear path. The next obstacle is size. Most refined work is still too large to finish in one go. A two-week story is a story we have already lost track of by Wednesday. The way out is slicing.

The next chapter is about cutting the work small enough that a human or an AI agent can read a slice, build it, and prove it

within days. Slices are how the trust the rest of this system is building actually compounds.

The Work Small Enough to Finish

The thirteen-pointer that would not die

Every team has one. The thirteen-pointer that started in sprint twelve, moved to sprint thirteen, then to sprint fourteen, and now sits in the team's standup the way a guest sits in a kitchen during a long party: present, unmoving, slightly embarrassing, nobody mentions it.

The team's retro keeps writing the same line. *We underestimated.* Three sprints in a row, the same line, like a band that only knows one song.

The team did not underestimate. The team mistook one large story for one piece of work, when it was actually six pieces of work pretending to be one. The estimate was for a unit that did not exist. The work could not be finished because finishing was never

possible. Finishing required several smaller finishes that nobody had named.

This chapter is about naming them.

> *A story is a sentence about a feature. A slice is the smallest piece of work the team can ship and prove.*

What a slice is

A slice is a unit of work with four properties. If your unit has all four, it is a slice. If it is missing any of them, it is not a slice. It is a story trying to look like a slice.

A bounded scope. You can name the files it touches. Not the layers. The files. The PO and the engineer agree on the boundary. The reviewer can hold the boundary in their head while reading the diff.

An acceptance criterion. The slice has at least one testable statement about what is true when it is done. (The next chapter is about how to write those statements as tests in disguise.)

A documentation target. The slice updates a specific doc. The module entry. A stack rule. A design token. Something. If a slice does not change any documentation, the slice either did not do anything meaningful or did something the team forgot to record.

A regression watch. A short list of things in the rest of the product that could plausibly break because of this change, and

how the team will know if they did. Two or three items. Not paranoid. Specific.

A unit of work with all four can be finished. A unit of work missing any of them cannot. It can be *worked on*, indefinitely, by smart people who are trying their best, and never actually finish, because nobody knows what finishing looks like.

The same thirteen-pointer, cut

Take the thirteen-pointer from the standup. The one called "*Customer can manage their notification preferences.*"

Here it is as one story. Two weeks of work. Touches the user model, the API, the auth check on the preferences endpoint, the preferences screen in the UI, the email service, and the analytics events that fire when preferences change. There is no acceptance criterion that can be tested all at once, because the story spans four layers of the product.

Now the same work cut into six slices.

Slice 1: data model and migration. Add a `notification_preferences` table or JSON column to the user. Migration is additive and reversible. No reads from this column yet. Acceptance: schema deployed in staging, migration up and down both run. Doc target: the data model entry for users. Regression watch: nothing yet, because nothing reads the field.

Slice 2: read endpoint. A new endpoint `GET /v1/me/notification-preferences`. Returns the user's preferences (or defaults if none set). No auth changes yet (uses existing session).

Acceptance: contract test passes, returns the default when no row exists. Doc target: API doc and the integrations doc. Regression watch: no existing endpoint behavior changed.

Slice 3: write endpoint with permissions. A new endpoint `PUT /v1/me/notification-preferences`. Requires the user to be the owner of the preferences. Validates the input. Returns the updated preferences. Acceptance: positive and negative permission tests, validation tests. Doc target: API doc, permissions matrix. Regression watch: any other endpoint sharing the same permission helper.

Slice 4: the screen, read-only. A new preferences screen in the settings area. Renders the current preferences. No editing yet. Uses existing tokens and components. Acceptance: screen renders for a logged-in user, accessibility audit clean. Doc target: the settings module entry. Regression watch: the settings nav, which the new entry is added to.

Slice 5: the screen, editable. Editing of preferences with the new endpoint. Validation. Success and error states. Acceptance: edit submits, error path renders, accessibility audit clean. Doc target: same as slice 4, updated. Regression watch: shared form components.

Slice 6: behavior wired in. The email service reads the preferences before sending. Analytics events fire on change. Acceptance: an email that should be suppressed is suppressed, an event is recorded on change. Doc target: the notifications module, the analytics events catalog. Regression watch: existing email volume should not drop for users who have not changed preferences.

Six slices. Each is small. Each can be finished. Each can be reviewed in under thirty minutes. Each merges without leaving the product broken. Slice three could ship to production with the screen not yet built, and the product would still work because nothing calls the endpoints yet. Slice four could ship without slice three by reading defaults. The slices have an order, but each one is alive on its own.

The team that runs this slicing finishes the work in about two working weeks, not three drifting sprints. The team that runs this slicing also *believes* it finished, because each slice has a definition of done that does not depend on the others.

The retro line for the thirteen-pointer disappears. It is replaced by six small retros that each say "yes, that went fine." The team's morale moves up by an amount that nobody can attribute to anything in particular.

The slicing rules

There are rules for cutting slices. They are not abstract. They are the rules a senior engineer applies in their head when they look at a story and ask: where do I cut.

Each slice should be one to three days of work. Less than one day is a fragment; merge it with its neighbor. More than three days is a story; cut it again.

Each slice should be reviewable in under thirty minutes. A diff that takes an hour to review will be reviewed in five

minutes, badly. A diff that takes thirty minutes will be reviewed in thirty minutes, properly. The reviewer's attention is a hard ceiling.

Each slice should be reversible. A rollback path exists. If the slice is a migration, the migration is reversible. If the slice is a feature behind a flag, the flag can be turned off. If the slice is a UI change, the previous UI is one toggle away.

Each slice should merge without breaking the app. A slice that depends on another slice being merged in the same release is not a slice. It is a hostage. Cut differently.

Each slice should be valuable on its own *to the team*, even if not to the customer. A slice that adds a schema is valuable to the team because it unblocks the next slice. A slice that adds a read endpoint is valuable because it makes the contract testable. Customer value can arrive in the last slice. Team value arrives in every slice.

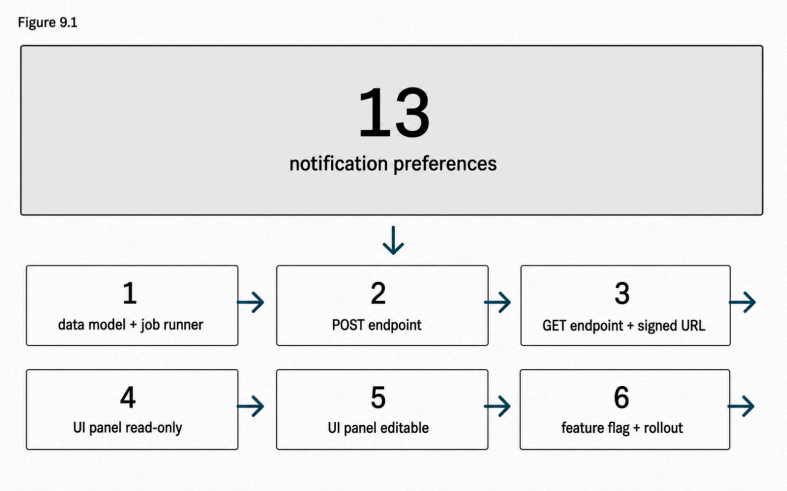

Figure 7: Figure 9.1. A thirteen-pointer cut into six pieces that can finish; slicing is the part of the work that decides whether the work can finish.

The slice checklist

Before a slice enters a sprint, walk it through four questions. If the answer to any of them is no, the slice is not ready. Re-cut or re-name until all four are yes.

- **Bounded scope?** Can you name the files this slice will touch (not the layers, the files)?
- **At least one testable acceptance criterion?** Written as precondition, action, assertion. Not a paragraph.
- **A specific documentation target?** A module entry, a stack rule, a token, an integration doc. *Something* changes.
- **A two- or three-item regression watch?** Specific. Not paranoid. Each item has a way to confirm or rule it out.

Four questions. Under a minute. The slice that passes the four is a slice the team can finish, the AI can build, and the reviewer can review. The slice that fails any of the four is a story trying to look like a slice. Re-cut.

One more sliced example

The thirteen-pointer above was a feature. Slicing also applies to the work that does not look like a feature. Consider a different shape: a migration.

> ***Original story.*** Move customer files from the current self-hosted storage to managed object storage. *Estimate: ten points. The kind of story that has a six-month pattern of "we will do it next quarter."*
>
> ***Sliced.***
>
> 1. ***Dual-write to both stores, behind a feature flag.*** *No reads from the new store. Acceptance: every new upload writes to both; reading still uses the old store; rollback is the flag.*
> 2. ***Backfill old files into the new store, idempotent, resumable.*** *Acceptance: a script can run to completion against a staging copy; running it twice produces no duplicates.*
> 3. ***Dual-read, prefer new store.*** *Reads try the new store first, fall back to the old. Acceptance: a test that*

confirms fallback works when the new store is missing the file.

4. ***Read from new store only, behind a per-customer flag.*** *Rolled out to one customer first, then ten, then all. Acceptance: monitoring shows no failed reads attributable to the cut-over.*
5. ***Stop writing to the old store.*** *Acceptance: the old store sees no writes for seven consecutive days; can be archived.*
6. ***Archive the old store.*** *Acceptance: the old store is read-only; one final integrity check passes; the team holds the archive for the agreed retention window.*

Six slices. Each is reversible. Each is observable. None of them require the team to hold their breath for a long-running migration that either succeeds or destroys data. The team that ships these six in two sprints is the team that, three years later, can do another storage migration without it being an event.

The anti-patterns

Some shapes look like slices but are not. They are worth naming, because they cost the same as the thirteen-pointer.

Vertical slices that depend on a backend that is not ready. Building a UI screen that calls an endpoint that does not exist yet. The slice is not done until the endpoint is. The team has invented two slices and one dependency, and the dependency is unmanaged. Either build the endpoint first, mock the endpoint

and own the mock as a separate slice, or stop pretending the UI slice is done when it is half done.

Horizontal slices that do not deliver anything observable. A slice that says "extract the validation logic into a helper" with no behavior change and no test added. This is not a slice. It is a refactor masquerading as a unit of work. Refactor without behavior change is fine, but it needs its own test that proves the behavior is unchanged. Without the test, the slice is the engineer trusting themselves. The team is trusting the engineer. Future-team will not.

Refactor slices that change a lot of code with no test added. Refactor is the most dangerous slice shape because it has no acceptance criterion in business terms. The acceptance is "code looks better" or "performance is faster." Both are real, both can be tested, but the test has to exist. A refactor slice without a test is not a slice. It is faith.

Cleanup slices that should have been part of the previous slice. A slice called "*tidy up the notifications module*" two weeks after the feature shipped. The tidy-up should have been in slice six. The team is paying for not finishing slice six on time. Tidy-up after the fact is the most expensive tidy-up, because the engineer has lost context and the bugs the tidy-up introduces are paid in a different week.

Myth. Slicing is for big stories.

Reality. Slicing is for *all* stories. Most small stories are also slices when you look at them carefully. The discipline is consistent. The size is variable.

What slicing does for AI

A slice is the unit of work AI can finish.

Stories AI cannot finish: too big, too cross-cutting, too many implicit decisions. AI either generates a sketch that requires a human to extend it, or generates working code that drifts from the team's conventions because it had to guess too much.

Slices AI can finish: bounded scope (so context is loadable), one acceptance (so success is testable), one doc target (so the trail is left), one regression watch (so failure modes are noticed). AI reads the slice. AI implements the slice. AI runs the tests. AI updates the doc. AI flags the regression watch items as touched or untouched. AI is no longer drafting. AI is *finishing*.

The team's job in this picture is not less. The team designs the slice. The team reviews the slice. The team accepts the slice. The team supplies the judgment. AI supplies the speed. The split is honest, and the split makes both sides faster.

> *A slice is the smallest unit of trust the team can extend to AI without regretting it later.*

A counter-objection

"This sounds like splitting things up just to feel productive."

It is not. The productivity gain is a side effect. The point is *finishing*. A team that finishes things builds a different relationship with its own backlog than a team that does not. The team starts trusting its own estimates, because the units of work are small enough to estimate honestly. The team starts trusting its own velocity, because the units of work actually close. The team's morale stops being a mystery, because morale follows finishing.

A team that ships six slices in a sprint feels different from a team that ships one half-finished thirteen-pointer in a sprint. Both did the same number of hours of work. Only one of them is on a roll.

A small habit

When a story arrives at refinement, the question is not *can we estimate this*. The question is *can we cut this into slices*. If the answer is yes, the story is sliced. If the answer is no, the story is paused. The reasons for "no" are usually one of three:

1. A fork is not yet resolved. Go back to the previous chapter.
2. The eight dimensions are not all filled in. Go back to *The Eight Decisions Every Story Carries*.
3. The team genuinely does not know enough to slice yet. The next ticket is a research spike, not the story.

Three reasons. None of them require the team to power through. Power-through is the most expensive way to be productive.

The bridge

Slices are units of work the team can finish. But "finish" needs a definition. A slice is finished when its acceptance criterion is proven. A criterion is proven when a test passes. So acceptance criteria are not paragraphs about how the feature works. Acceptance criteria are tests in disguise, written in business language so the PO can read them, and shaped so the engineer can run them.

The next chapter is about writing them. It is the chapter that turns "*done*" from an opinion into a fact.

The Moment Done Becomes Provable

The story that was done, twice

A story moves to *done* on a Friday afternoon. The engineer pushes the code, the demo works, the PO clicks the link, the screen does what the screen is supposed to do. The team logs off. The story is finished.

Monday morning, the story is open again. The PO has spent the weekend thinking about it and realized the engineer built it slightly differently from what she meant. The engineer reads the ticket and is honest: yes, the ticket said *user can update their email*, and the user can now update their email, and the work the engineer did is exactly what the ticket asked for. The PO reads the ticket and is also honest: no, the ticket said *user can update their email*, and the engineer did not build *the thing she*

meant by that sentence, because the sentence carried meanings the engineer could not have known.

Both people are correct. Both people are tired. The story is reopened. It will be reopened twice more before it stays closed.

This is not a communication problem. This is an acceptance-criteria problem dressed as a communication problem.

> *"Done" is an opinion when the acceptance criteria are paragraphs. "Done" is a fact when the acceptance criteria are tests.*

Why prose acceptance criteria fail

The default shape of an acceptance criterion in most teams is a sentence or two. It reads like a description. "*User can update their email.*" "*User receives a confirmation.*" "*The system handles errors gracefully.*"

These sentences are not wrong. They are also not testable. They each contain hidden assumptions that two people can read differently and both feel correct.

User can update their email. Can they update to any address. Can they update to an address that is already in use. What happens to their old address. Do they have to confirm. Is the confirmation a click on a link or a code in an app. Does the link expire. What if they already requested a change yesterday. Does the change happen immediately or after confirmation.

The PO has answers to all of these in her head. The engineer has different answers in his head. The sentence does not contain either set. The disagreement is not visible until the code runs against the difference, and the difference is the bug.

> **Myth.** Acceptance criteria are descriptions of the feature.
>
> **Reality.** Acceptance criteria are *statements that must be true* about the feature. Statements that must be true can be tested. Descriptions cannot.

The shape that works

There is a small structural shift that fixes most of this. Acceptance criteria are not paragraphs. They are a list. Each item in the list is a single testable statement with three parts.

A precondition. What is true before the action.

An action. What happens.

An assertion. What must be true after.

The shape can be Given-When-Then if your team likes that vocabulary. It can be plain English with the same structure. It does not matter. What matters is that each item is one statement, with one precondition, one action, and one assertion, and it can be turned into a test by a human or a machine without re-asking the PO.

Each statement is one of three types: a *positive case* (the happy path), a *negative case* (the failure mode), or an *edge case* (the

corner that breaks the assumptions). Most stories need at least one of each. Many stories need two or three of each.

The same story, rewritten

Here is the story that opened and closed three times. Rewritten as testable acceptance criteria.

> ***Positive cases.***
>
> *Given a logged-in user with a confirmed email, when they submit a valid new email address, then a confirmation email is sent to the new address, the old address is preserved until confirmation, and the user sees a message that confirmation is required.*
>
> *Given a user with a pending email change, when they click the confirmation link within twenty-four hours of the request, then their email is updated to the new address and a notification is sent to the old address.*
>
> ***Negative cases.***
>
> *Given a logged-in user, when they submit an email address that is already in use by another account, then the form returns error code* `DUPE_EMAIL` *and the form does not submit.*
>
> *Given a logged-in user, when they submit an invalid email address, then the form returns error code* `INVALID_EMAIL` *with the offending field highlighted.*

> *Given a confirmation link, when it is older than twenty-four hours, then it is rejected with error code* `EXPIRED_TOKEN`*.*
>
> ***Edge cases.***
>
> *Given a user with an existing pending email change, when they request a new email change, then the previous request is invalidated and the previous confirmation link no longer works.*
>
> *Given a confirmation link, when the user is logged out, then they are prompted to log in before confirmation completes.*
>
> *Given a user, when they update to an email address that matches their current address, then the system treats it as a no-op and shows a friendly message.*

Each line is testable. Each line is a fact that has to be true when the story is done. The engineer cannot accidentally build the wrong story, because the wrong story would fail at least one of these checks. The PO can still move the goalposts later, but now everyone can see that the goalposts moved.

The number of acceptance criteria for this story went from two paragraphs to eight specific statements. It looks like more work. It is less work. The eight statements take ten minutes to write. The disagreement they prevent took the team three days last time.

In one product team, adopting the three-part acceptance-criteria shape dropped reopens sharply within a sprint. The remaining reopens were not disagreements about what was meant. They were genuine new requirements that the team had learned during the work. Those are honest reopens. The dishonest ones disappeared.

Acceptance criteria and tests

Once acceptance criteria are written this way, they are tests that have not been compiled yet.

Each positive case maps to a happy-path test. Each negative case maps to an error-path test. Each edge case maps to a corner-case test. The work of writing tests becomes the work of translating language to syntax, not the work of figuring out what to test. The figuring-out was done in refinement, by the people who knew.

This has three knock-on effects.

Tests get written, because they are pre-designed. The most common reason tests do not get written is that the engineer does not know what to test. With pre-designed tests in the acceptance criteria, the question is answered. The engineer writes the tests because the path of least resistance now includes them.

QA shifts from finding bugs to finding what the acceptance criteria missed. QA stops being a safety net for the obvious cases (the acceptance criteria caught them). QA

starts being a safety net for the unobvious cases (the ones the team forgot in refinement). This is a better use of QA's brain.

The PO and the engineer have a shared definition of done that cannot drift. Done is "all the acceptance-criteria tests pass, and they are run in CI on every change." That is not an opinion. It is a build artifact.

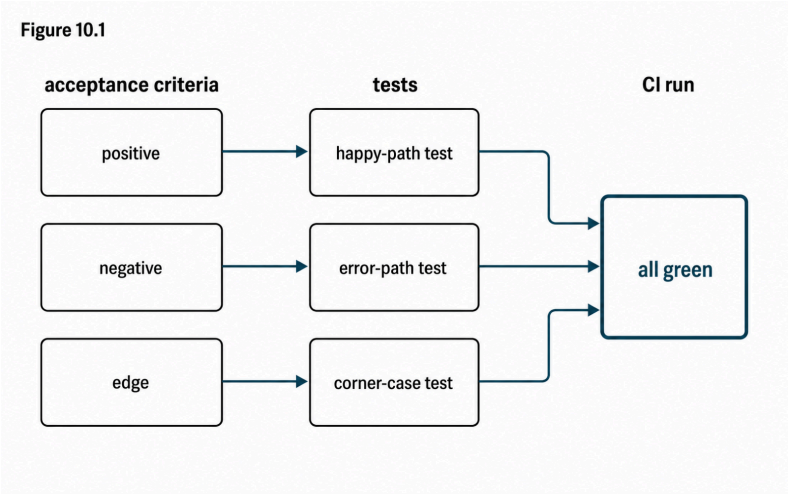

Figure 8: Figure 10.1. Acceptance criteria turn into tests, and tests turn "done" from an opinion into a fact: the translation is the typing.

One translated, end to end

Take one criterion from the rewritten story above and walk it through to a test. Keep it concrete.

> ***Criterion.*** *Given a logged-in user, when they submit an email address that is already in use by another account,*

> *then the request returns HTTP 409 with error code `DUPE_EMAIL` and the form does not submit.*

The criterion already has the three parts: precondition (logged-in user, email exists), action (submit a duplicate), assertion (`DUPE_EMAIL`, no submission). The translation to a test is mechanical.

```
test('submitting a duplicate email returns DUPE_EMAIL and
does not submit', async () => {
  const existing = await createUser({ email:
'taken@example.com' });
  const session = await loginAs(await createUser({ email:
'me@example.com' }));

  const response = await session
    .post('/v1/me/email-change')
    .send({ email: 'taken@example.com' });

  expect(response.status).toBe(409);
  expect(response.body.code).toBe('DUPE_EMAIL');
  expect(await
getEmailChangeFor(session.userId)).toBeUndefined();
});
```

The test name is the criterion. The body is the criterion's three parts in code. A reviewer who reads the test name knows what is being claimed without reading the body. A PO who reads the test name knows the contract is being honored. The AI writing the test has nothing to invent.

A prompting pattern for negative and edge cases

A pattern worth keeping: when asking the AI to write tests, give it the positive cases, the negative cases, and the edge cases as three separate lists. Ask it to produce one test per case, named after the case. Tell it to fail loudly on assumptions: if a value cannot be inferred from the criterion, the AI should leave a `// TODO: confirm` comment instead of guessing.

The prompt itself can be three short bullets:

- Here are the positive cases: [list]. One test each, named after the case, strongest possible assertion.
- Here are the negative cases: [list]. One test each, asserting the specific error code and any required headers.
- Here are the edge cases: [list]. One test each, naming the corner condition in the test name.

The AI produces a first pass of the tests quickly. The reviewer reads the names in a couple of minutes. The work that used to take an afternoon takes ten minutes. The acceptance was the work. The translation was the typing.

When AI writes the tests

Once acceptance criteria are testable statements, AI can write the tests directly. Give the AI the criterion and the project's test conventions, and it produces a test that runs. Most of the test is mechanical. The thinking was done upstream.

This is one of the largest gains AI offers a team that has done the upstream work. Tests stop being expensive to write, because the cost of writing them is no longer the cost of thinking

through what to test. The thinking is in the AC. The writing is a translation.

A team that has not written acceptance criteria this way cannot get this gain from AI. The AI either generates generic tests that do not match the story, or asks the engineer enough clarifying questions that the engineer might as well have written the tests themselves. The acceptance criteria are what unlock the AI's usefulness here. Without them, the AI is back to averaging.

Tests are not a cost. Tests are the form your decisions take after refinement. Honest criteria are the care the code carries forward.

How to write acceptance criteria that survive

There are habits that make this shape stick.

Write at least one negative case for every positive case. The negative case is where the bugs live. A team that lists three positive cases and zero negative cases has not refined the story. It has dreamed the story.

Write at least one edge case for any story that has time, sequence, concurrency, or state involved. Time-based things expire. Sequenced things race. Concurrent things

deadlock. Stateful things drift. Edge cases are where these costs show up. Skipping them is skipping the bill.

Use stable error codes, not user-visible strings. A test that checks for the string "*That email is already in use*" breaks when the copywriter rewrites the message. A test that checks for error code `DUPE_EMAIL` survives. The PO can pick the user-visible string later, in a different ticket, without breaking anything.

Refer to data by its meaning, not by its database shape. "*The user's old email is preserved*" is a meaningful statement. "*The* `users.email_pending` *column is null*" is an implementation detail that might change in slice three of the next refactor.

Each criterion is independently verifiable. Avoid criteria that read "and then also." Two ideas in one criterion become two tests, written badly. Split them.

What acceptance criteria do for AI

When AI agents are given acceptance criteria of this shape, several things happen.

The agent stops asking clarifying questions about the obvious cases. The obvious cases are listed. The agent moves directly to implementation.

The agent generates tests as part of the implementation, not as a follow-up. The tests are derived from the criteria. The tests pass because the implementation was written to satisfy them.

The agent can refuse to mark the slice as done until the criteria pass. This sounds aggressive. It is the same standard a senior

engineer would apply. The agent is just applying it without flinching.

The agent's output is reviewable by the PO, not just the engineer. The PO can read the criteria and read the tests and see that the tests match the criteria. The PO does not have to read the code. The PO becomes part of the verification step in a way that does not require the PO to learn the codebase.

> **Myth.** Detailed acceptance criteria slow refinement down.
>
> **Reality.** Detailed acceptance criteria make refinement *finish*. Vague acceptance criteria make refinement reopen six times under different names: scope creep, "more context needed," QA defects, customer escalations. The slow version is the vague one. The vague version just hides the slowness under different labels.

A counter-objection

"This sounds like writing tests in the ticket. Tests are the engineer's job."

The tests are still the engineer's job. The *design* of the tests is not. The design of the tests is the team's job, in refinement, with the PO present. What goes into refinement is *the testable behavior the feature has to demonstrate*. What comes out of refinement is a list of statements that can be translated into tests. The engineer translates. The team designed.

This split is what makes the team fast. The PO is no longer a gatekeeper of vague meaning. The engineer is no longer a sleuth trying to infer intent. Both are working from the same artifact, in different roles. The artifact does not care which person reads it.

A small habit

In refinement, after the eight dimensions are filled in and the lane is picked, the PO and the engineer write the acceptance criteria together. Out loud. The PO names the case in business language. The engineer translates it into the three-part shape. If the engineer cannot translate it, the case is unclear. The PO clarifies. If the PO cannot answer the engineer's clarifying question, there is a fork (go back to the chapter on the decision before the code). They handle the fork, then continue.

This takes ten to twenty minutes per story. It feels like a lot. It is the cheapest twenty minutes the team will spend on this story. The alternative twenty minutes will be paid in week three, in support, in customer escalation, in the retro that says "*we underestimated*" for the fourth time about a different story.

The bridge

Eight dimensions filled in. Docs in place. Lane picked. Forks frozen and called. Slices cut. Acceptance criteria written as tests in disguise. The story is now buildable, finishable, provable. The team has, on paper, everything it needs to refine well.

The next chapter shows what it actually looks like when all of these parts run together on a single story, in one short session,

with the AI drafting from context and the team owning the calls. It is the chapter where the system stops being a list of habits and becomes a conversation that settles the work.

Part IV

The System

The final five chapters are what the parts assemble into when they run together. The conversation that settles the work in a single short session. The review that protects the product when AI did the typing. The memory that makes estimates better one sprint at a time. The owner behind every story whose call the work carries. The backlog that thinks.

By the last chapter, the system has a name. By then, the team has been running it long enough that the name is the least interesting thing about it.

The Conversation That Settles the Work

A Wednesday afternoon, about fourteen minutes

This chapter is shorter than the ones before it. It is not a re-teach. It is a record.

Real sessions are messier than this one. The hesitations, the dead ends, the half-formed thoughts that turned into nothing have been removed from the transcript below so the shape of the work is visible. The fourteen-minute figure is illustrative, not a promise; the point is the order of operations, not the clock.

By this point in the book, every part of a good ticket has a name. The eight dimensions. The three documents. The system-aware draft. The lane the work earns. The fork that gets named. The slice that can finish. The acceptance criterion that is a test in disguise. The work that remains is to show what it looks like

when all of those parts are in place at once, and a single story moves from a sentence to a pushed ticket inside one short session.

The transcript below is a composite. The shape is real. The exact words are paraphrased so the example can stand on its own. The PO is Lena, who has been running her team's backlog the way the previous chapters describe for about two quarters. The AI is the same AI her engineering team uses, connected to the codebase, to Jira through MCP, and to the team's docs. The session is about a request that came in over the weekend.

> *A good refinement session does not feel like building a ticket. It feels like reviewing one.*

The request

A customer-success teammate posted in Slack on Sunday.

> "Two enterprise customers have asked if they can export their own data on demand. Today they email us, we run a script, we send a zip. They want a button. Is this small?"

The ticket has not been written. The work has been described. Lena opens the AI in her terminal on Wednesday afternoon.

Turn one: the AI proposes

> ***Lena:*** *Customers should be able to export their own data on demand. Today this is a manual script. Draft a system-aware ticket.*

> ***AI:*** *Reading repo (modules:* `accounts`*,* `data-export`*,* `notifications`*,* `auth`*), open tickets (none matching), and decision records (*`docs/decisions/2025-07-data-export-cli.md` *is relevant).*
>
> *Proposed draft:*
>
> > *Workflow: feature. Modules:* `data-export/` *(existing CLI script* `bin/export-customer.ts` *is the canonical source of the dataset shape),* `accounts/` *(existing* `Customer` *model,* `CustomerOwner` *role),* `auth/` *(existing scope* `customer:read`*, no scope yet for* `customer:export`*),* `notifications/` *(existing transactional template* `account.welcome`*, no export-ready template). Scope: multi-module. Database impact: additive only (one new table* `data_export_requests` *to track state and the resulting file location). API impact: additive endpoints.* `POST /v1/customers/{id}/exports` *to request.* `GET /v1/customers/{id}/exports/{exportId}` *to check status and download. UI impact: new component (an "Export your data" panel on*

> > *the account settings screen). Uses existing button and progress tokens. Risk: medium. The script today produces the dataset in three to nine minutes for current enterprise sizes. The customer-facing version needs to be async. Reversibility: easily reversible (feature behind flag* `customer_self_export`*). Compliance sensitivity: high (the dataset includes personal data and raises GDPR access and portability considerations, depending on what the export contains). Data sensitivity: contains PII. Customer-facing impact: customer-visible.*
>
> *Open forks (1): F1: Should the export run synchronously up to a size threshold and asynchronously above it, or always asynchronously? The CLI today is synchronous and works for current enterprise sizes, but customer-facing latency budgets are tighter than ops budgets.*

The draft took eleven seconds.

Turn two: Lena corrects

Lena reads the draft twice. She catches one thing.

> ***Lena:*** *Modules look right except* `notifications`*. We are not sending an email at all for the first version. The*

> *download appears in the panel. Drop* `notifications` *and the template line.*

> ***AI:*** *Updated. Removing* `notifications/` *from affected modules and the related regression watch entry. Email notification is now noted as deferred. Want me to file a follow-up ticket placeholder for "Notify customer when export is ready"?*

> ***Lena:*** *Yes. Title only. Same epic.*

> ***AI:*** *Placeholder drafted:* `CUST-EXPORT-2: Notify customer when self-service export is ready`*. Status: candidate. Will push on approval.*

She moves on.

Turn three: the fork

Lena reads F1 again. The fork is real. It is also the kind of fork that, two quarters ago, would have been decided inside the implementation. She freezes.

> ***Lena:*** *Freeze on F1. Open a decision packet. Options: A) always async, B) sync under 30s and async above, C) sync under 60s and async above. Recommend?*

> ***AI:*** *Recommendation: A, always async. Reasoning: The current synchronous CLI takes 3 to 9 minutes for present enterprise sizes. Sync in the UI would block the user's browser tab and consume a server worker for the duration. Always-async makes the contract uniform regardless of dataset size, which is the dimension most likely to change. Confidence: medium-high. The soft part of the recommendation is the assumption that dataset size grows. Drafting decision packet now:*
>
> > ***D-47.*** *Sync vs async for customer self-service export. Category: workflow or tool change. Options: A) always async (recommended), B) sync under 30s threshold + async fallback, C) sync under 60s threshold + async fallback. Recommendation: A. Confidence: medium-high. Caller: PO with engineering lead. TTL: 6 months (revisit if enterprise dataset sizes shrink dramatically).*

The packet appears in `decisions/pending/D-47.json`. Work cannot move forward until it is resolved.

> ***Lena:*** *Walking over to the engineering lead now. Two minutes.*

She comes back four minutes later. The lead agreed with A. They named two extra small consequences (retry semantics, observability event for export-failed) and noted them.

> ***Lena:*** *Resolved: A. Add the two consequences they raised.*

> ***AI:*** *Recording resolution to* `decisions/resolved/D-47.json`*. Resolution: A, always async. Adding to the manifest: retry policy (exponential backoff, three attempts) and observability event* `export.failed` *with reason code. Plus a regression watch entry on the existing* `data-export` *CLI to confirm it is unaffected.*

The fork closed in under five minutes. The decision is now in a place the next ticket touching exports will read first.

Turn four: the slices

The story is now refined enough that it should not be a story.

> ***Lena:*** *Slice it. Each slice one to three days, reversible, mergeable independently.*

> ***AI:*** *Proposed slices (5):*
>
> > ***Slice 1.*** *Add table* `data_export_requests` *and a job runner that wraps the existing CLI script. Internal-only invocation. No UI, no API. Acceptance: schema migration runs up and down; the job runs against a fixture customer end-to-end and writes the file to staging storage. Doc target:* `docs/modules/`

`data-export/business.md`. *Regression watch: none (no external surface yet).*

Slice 2. *API endpoint* `POST /v1/customers/{id}/exports`. *Calls into slice 1's job runner. Requires the existing* `customer:read` *scope plus the new* `customer:export` *scope (which will be added in slice 2 itself, granted to* `CustomerOwner` *by default). Acceptance, positive: a* `CustomerOwner` *can create a request and receives a* `202` *with an* `exportId`. *Acceptance, negative: a user without* `customer:export` *gets* `403 SCOPE_MISSING`. *Acceptance, edge: a duplicate request within five minutes returns the existing in-flight* `exportId` *instead of creating a new one. Doc target: API doc, scopes matrix. Regression watch:* `auth/middleware/scope.ts`.

Slice 3. *API endpoint* `GET /v1/customers/{id}/exports/{exportId}`. *Returns status (*`pending`, `ready`, `failed`*) and, when ready, a short-lived signed URL. Acceptance: status transitions are testable end-to-end; signed URL expires in fifteen minutes. Doc target: API doc. Regression watch: signed-URL helper (used elsewhere for invoice PDFs).*

Slice 4. *UI panel "Export your data" on the account settings screen. Reads-only against the new endpoints. Renders states pending, ready (with download), failed*

> *(with retry CTA). Acceptance, positive: the panel renders for a `CustomerOwner` and updates on poll. Acceptance, negative: a non-owner sees the panel disabled with an explanation. Acceptance, edge: the panel handles a `failed` state without errors and offers retry. Doc target: design system additions (status pill variant used here), settings module entry. Regression watch: shared `Panel` component (used in three other screens; visual diff only).*
>
> ***Slice 5.*** *Wire the feature flag `customer_self_export` and enable it for the two requesting enterprise customers. Acceptance: with flag off, the panel does not render and the endpoints return `404 FEATURE_NOT_ENABLED`. Acceptance, positive: with flag on for a customer, the full flow works end-to-end. Doc target: feature-flag inventory, runbook entry for enabling and disabling. Regression watch: existing feature-flag wrapper.*

Notes:

- *Each slice merges to main without breaking the app.*
- *Slices 1 to 3 add no customer-visible behavior.*
- *Slice 4 is customer-visible but invisible without slice 5.*
- *Slice 5 is the cut-over.*

Lena reads the slices. She moves slice 3 above slice 2 in her head, then puts it back. The dependency is real either way.

She reads the acceptance criteria on each slice, more slowly on slice 2. The phrase "returns the existing in-flight `exportId`" catches her. She tightens it in her head, types a small note in the margin to revisit the wording, then checks one number against the team's reality.

> ***Lena:*** *On the duplicate window in slice 2, the five-minute value. Check it against the existing retry policy on the partner-integration side. If they retry inside that window, we should not be issuing them new exports.*

> ***AI:*** *Checking. The partner-integration retry policy in `docs/decisions/2025-04-partner-retries.md` retries on a backoff of 30s, 2m, 8m. Five minutes sits between the second and third retry, which means a partner retry inside the window will hit the same in-flight `exportId` (correct), and a retry outside the window will get a fresh request (also correct). No conflict. Flagged in the slice for the implementer.*

> ***Lena:*** *Looks right. Lane?*

> ***AI:*** *Graduated, on the rule. Compliance sensitivity is high, but high compliance only forces the full lane when the change is also hard to reverse or reaches many*

> *customers at once. This one is easily reversible, behind the* `customer_self_export` *flag, rolling out to two customers first, so it stays graduated, with a privacy-review gate before the flag flips. The API is additive. Module health on* `data-export` *and* `auth` *is green, with no relevant defect-pattern recurrences last quarter.*

> ***Lena:*** *Graduated. We will do a privacy review on slice 5 before flipping the flag.*

> ***AI:*** *Noted on the manifest. Adding a verification gate "privacy review sign-off" as a precondition for slice 5 closing.*

Turn five: the push

> ***Lena:*** *Push.*

> ***AI:*** *Ready to push to Jira project* `CUST-EXPORT`*. Parent epic:* `CUST-EXPORT-1` *(new). Five child stories from the slices above. One follow-up candidate ticket* `CUST-EXPORT-2` *(notification). One decision record* `D-47` *linked. One regression-watch sub-issue on the existing CLI. Confirm?*

> ***Lena:*** *Confirm.*

> ***AI:*** *Pushed. Epic* `CUST-EXPORT-1` *created with five linked stories (*`CUST-EXPORT-3` *through* `CUST-EXPORT-7`*), one candidate (*`CUST-EXPORT-2`*), one regression watch issue (*`CUST-EXPORT-8`*), and the decision record* `D-47` *linked from the epic description. Plan recorded under* `specs/cust-export-1.yaml`*. The plan-vs-actual log will pick up automatically when work begins.*

Lena closes the terminal. The session lasted fourteen minutes.

The same request, six months earlier, would have produced a single oversized story with no fork named, no slices, and no decision record. The team would have built it for three to five sprints. The version above is finishable inside two sprints, by people who already know what they are agreeing to.

The split, made explicit

Read the transcript again with one question in mind. Who did what.

The AI proposed. Modules, dimensions, slices, acceptance criteria, the recommendation on the fork, the lane suggestion, the regression watch, the doc targets, the push to Jira. Almost everything the AI produced was a draft drawn from system context (the codebase, the open tickets, the decision records).

The human decided. Whether the notification module belonged. Whether the fork warranted a freeze. Whether to take the recommended option. The lane. Whether to elevate. Whether to push. The new follow-up ticket. The two consequences the engineering lead added.

The AI never made a product call. It surfaced the call. It proposed an option. It waited.

This is what "AI builds, you own" looks like at the backlog layer. The AI produced the draft; the human owned the calls. The split is honest because the AI's role is to produce a defensible draft, and the person's role is to defend it.

A team that runs this split well does not have to ask whether the AI is in charge. The AI is never in charge. The AI is the fastest first reader the team has ever had.

> **Myth.** A good AI workflow is one where the human steps in less.
>
> **Reality.** A good AI workflow is one where the human steps in *at the right moments*. The human steps in for product decisions, forks, lane calls, push approvals, and corrections of context the AI could not have. The human steps out of the parts that did not need a person to do them.

What this session relied on

The transcript reads like a clean session because everything upstream of it was in place. It is worth naming what those

upstream pieces were, because without them the same session would have stalled inside the first turn.

The team had docs. The module list referenced real modules because the module map existed. The decision record on data export from the previous summer existed because the team writes decision records when forks are called. The conventions the AI applied existed because the conventions are written in the project's instruction docs, not in any one engineer's head.

The team had the wiring. The AI had read access to the codebase. The AI had a connection to Jira through MCP. The AI had access to the docs. The wiring took a Friday afternoon to put in place two quarters ago and has not changed much since.

The team had habits. The PO knows the eight dimensions by name. The PO knows when a fork has to be frozen. The engineering lead knows that walking over to make a call inside five minutes is cheaper than building the wrong thing for three weeks. The habits are not impressive. They are the unglamorous part of the work that makes the rest of it look easy.

If any one of those three were missing (docs, wiring, habits), the session would have either stalled or produced a clean-looking ticket that hid the real work the team would pay for later.

A counter-objection

"This only works because everything was already there."

That is the point. The earlier chapters were the work that gets you to this session. The earlier chapters are the price you pay

so refinement gets short. A team that runs this session well is a team that has done the homework. There is no shortcut to the session. The session is the reward.

There is also a quieter version of the same objection: "We will never have everything in place." Most teams will not have everything in place all at once. Most teams add the pieces one at a time, over a quarter or two, and notice the sessions getting shorter as they go. The session above is the destination. The walk to it is what the earlier chapters describe.

A small habit

The team that wants this kind of session has a standing recurrence. Once a week, one refinement session is run in front of the team, with the AI live, so everyone can see what the session feels like. The point is not training. The point is normalization. After three or four of these, the rest of the team starts running their own refinement sessions the same way, without having to be asked. The shape becomes obvious because it has been seen.

The habit is fifteen minutes a week. The return is every refinement the team does for the rest of the year.

The bridge

The session above produces a refined ticket. The slices begin. The AI helps build them. The tests pass. The work merges. At some point in the next week, the work is supposedly done. The next problem starts here.

How does a PO who does not read code know that what the AI built is actually right. How does the team calibrate trust in a per-area, per-sprint way, instead of either rubber-stamping every green check or reviewing every line. The next chapter is the human skill of verifying AI output. It is the chapter that decides whether the speed in this session compounds, or evaporates the moment something quietly breaks.

The Review That Protects the Product

The thing the dashboard does not show

One founder had a green dashboard for nine sprints in a row. Every story closed. Every test passed in CI. The team's velocity chart was the cleanest she had ever seen. The AI was doing most of the building. She was watching the right numbers.

She was not watching the right number.

In the same nine sprints, the team's customer-success queue grew by roughly a third, in her team's own count. The bugs were small. None of them were big enough to be called incidents. Each one was a *small disagreement* between what a customer expected and what the code did. None of them showed up in the CI run, because none of them were what the team had tested. The acceptance criteria the team wrote were honest. The acceptance criteria the team did not write were where the bugs lived.

Nothing in her dashboard was wrong. The dashboard was measuring what had been agreed. The bugs were what had not.

This chapter is about the difference.

A green check means the test passed. It does not mean the right test ran.

Why this gap is now larger

A randomized controlled trial by the research group METR illustrates the trap. Sixteen experienced open-source developers completed two hundred and forty-six tasks in repositories they knew well. With early-2025 AI tools allowed, they took about 19 percent longer, even as they believed, afterward, that the tools had sped them up. The study is a snapshot of one setting, not a universal law, and its authors have cautioned against overgeneralizing it. But it points straight at the thing this chapter is about. AI removes typing, not verification. If the team's review habits do not scale with the build speed, the time saved in writing is repaid, with interest, in catching what was written.

The opening chapter's framing returns here in a different vocabulary. AI did not change what *review* is for. It changed the volume that arrives at review, and it changed who plausibly does it. The team that did not change its review habits is now drowning in green checks that do not mean what green checks used to mean.

What review is for, when AI builds

The acceptance chapter said *done* becomes a fact when acceptance criteria are tests. That chapter answers the question "what must be true?" This chapter answers the next one. *How does the team confirm it is true, when the team did not write the implementation?*

A note on scope before that. Review asks whether the change matches the contract and whether the contract was strong enough. QA asks what the contract missed. The two are complementary, not interchangeable. A team that asks one to do the other's job ends up doing both badly.

The answer has three layers. The team needs to know what to check, what to trust, and how to update both based on what actually happens.

What to check. The acceptance criteria. Read them as the contract. Confirm each one has a test. Confirm each test runs in CI on this change. Confirm the test fails when you break the behavior on purpose. A test that does not fail is a test that does not exist.

What to trust. The conventions the team has already agreed on. The patterns the codebase already uses. The behavior of areas the team has already verified in previous sprints. Trust is not given to the AI. Trust is given to areas of the codebase where the team has already accumulated evidence.

How to update. Every review either confirms trust or erodes it. The team records both. A clean review in Notifications raises the

trust on Notifications by a notch. A defect found in Billing lowers the trust on Billing by a notch and raises the next review's depth.

This is the loop. Review is not a single gate. It is a feedback system that gets tuned over time.

What the PO can verify without reading code

Most POs cannot read code at the speed they would need to in order to review every change. This is not a failure. It is a fact about the split. The PO can verify two things that matter more than the code.

The tests, read as English. A well-named test reads like a sentence the PO wrote in refinement. *Given a user with a password hash, posting to /v1/sessions returns a session token.* The PO does not need to read the test to know what it claims. The PO needs to read the *name* of the test and match it against the acceptance criterion. If the test name does not match a criterion, either the test is testing the wrong thing or the criterion was never agreed to. Both are review findings.

The regression watch, walked through. The regression watch was the team's bet about what could plausibly break. The reviewer walks the list. For each item, the team confirms it was either touched and re-tested, or untouched. An item that was touched and not re-tested is the most common cause of the bug the customer files three weeks later.

These two checks take the PO between three and ten minutes per slice. They do not require reading code. They protect the product

more reliably than reading code does, because they match the layer the bugs actually live at.

> **Myth.** A PO who cannot read code cannot meaningfully review AI output.
>
> **Reality.** A PO who reads tests by name and walks the regression watch reviews more of what matters than a PO who reads the code line by line. The line-by-line read is the wrong scale. The wrong scale is what missed the bug.

What the engineer should verify

The engineer reviewing AI output has a different job. The work is small. It is also unfamiliar to most reviewers because the engineer did not write the change. There are five things to look for. The list is short on purpose.

Did the change do the smallest thing. AI agents have a tendency to extend the diff when they should have closed it. A slice that touches three files when the criteria allow one is a slice that quietly grew. Pull it back to the smallest version that satisfies the criteria.

Did the change touch only what the manifest said. The regression watch named what could plausibly be affected. The diff should overlap with that list. A file edited outside the regression watch is either an honest expansion (record it) or a sign the slice was wrong from the start (re-slice it).

Did the tests fail on the broken case. On high-risk or low-trust slices, take the time to comment out the implementation, run the tests, and confirm they fail. If they pass, the tests are not testing the behavior. They are testing some adjacent thing. On routine, high-trust slices, this step is overkill; spot-check it occasionally rather than on every change.

Did the docs update. Every slice had a doc target. The doc target was not optional. A slice that closed without updating its doc is a slice that has already started to lie about itself.

Did the change record what it learned. A note in the plan-versus-actual log. Three lines. What took longer than expected. What was novel. What the team would do differently next time. The note is the seed of the chapter after this one.

These five checks take the engineer between five and fifteen minutes per slice. They produce a review the team can defend.

Reading the tests, not the code

A specific habit is worth naming because it is what most teams skip.

When AI writes a test, read the test. Do not skim it. Read the precondition, the action, and the assertion. Three lines. Confirm the assertion is the *strongest* statement of what should be true, not the *weakest* one that could pass.

AI agents tend toward weak assertions. A test that asserts the response was not null is weaker than a test that asserts the response was a session token belonging to the requesting user

with the right scopes. The first test passes when almost anything happens. The second passes when the right thing happens. Both are green. Only one is review.

The same applies to negative cases. A test that asserts an error was raised is weaker than a test that asserts the error code was `OAUTH_ONLY` and the rate-limit header was present. Strong assertions are the difference between a green dashboard that protects you and a green dashboard that flatters you.

A weak assertion is a quiet promise. It will be broken later, by something nobody saw, in a sprint nobody connects to this one.

The regression watch, as a review map

The regression watch is not a paranoia list. It is the team's review map. Every item is one question the reviewer is supposed to answer.

For each item, the reviewer asks two things. Was this file touched in the diff. If yes, do the tests cover the new behavior. If no, does the diff change any file that *calls* this file. The second question catches the silent regression: the diff did not edit the file, but it changed the contract the file depends on. The file did not change. The reason for the file to keep working did.

A regression watch with three or four items, walked properly, often protects the product better than a fifty-item checklist that

nobody reads. The shorter list survives the reviewer's attention. The longer list does not.

The teams that adopt this two-question habit do not first report finding more bugs in review. They report a quieter customer-support queue. The bugs they catch in review are the ones the customer would have filed two weeks later. The customer does not file the ticket the team caught.

Trust, calibrated per area

Trust in AI output is not one value. It is a value per area of the codebase. A team can reasonably trust the AI in Notifications after six clean sprints and still review every line the AI writes in Billing because Billing has had defects.

This is not pessimism. It is calibration. Most teams that "trust the AI" do so flatly: green check, merge. Most teams that "do not trust the AI" do so flatly: every change reviewed deeply regardless of where it lives. Both extremes are wrong, in opposite directions.

The shape that works is a small per-module ledger. For each module, the team records three things over time. *How often does AI-built work in this module pass review without findings. How often does it ship without a defect in the following two sprints. Which defect patterns recur here.* The ledger fits on a single page for most products.

The smallest starting version is three columns: module, last AI-related defect, current review depth. A team that has nothing today can start with that and grow into the rest as the data accumulates.

The ledger does two things. It tells the reviewer how deep to go on the current change (deeper in low-trust modules, lighter in high-trust ones). It also tells the team where the next investment in tests, docs, or stack rules will pay back the most. Trust is not given. Trust is observed. Where it is high, the team moves fast. Where it is low, the team moves carefully until the data changes.

Trust is also revocable. A module that has been clean for six sprints can lose its high-trust standing after a single defect that traces back to a missed convention. The ledger update is not punishment. It is honesty. The team is not pretending to be more confident than the data says.

This ledger ties directly to the chapter after this one. The plan-versus-actual loop is the data feed for the trust ledger. Every story that closes adds one data point. Six months in, the ledger is more accurate than any senior engineer's instinct, because the instinct rests on the same data, with worse recall.

The human gate before shipping

The team has the wiring from the System the AI Can Read chapter. The AI can draft. The AI can push tickets. The AI can run tests. The AI can submit pull requests. The team is now in a position where, technically, the AI could merge.

The team's job is to make sure it does not.

A human gate before shipping is non-negotiable. The gate can be short. The gate can be one person, named on the ticket, who has read the tests, walked the regression watch, and clicked the merge. The gate cannot be skipped. Auto-merge on green is the failure mode most teams will reach within a year of running AI agents in their pipeline, because auto-merge feels like progress until the day it does not. After that day, the team retroactively adds the gate, and adds it more strictly than it would have if it had added it first.

The gate is not about distrusting the AI. The gate is about ownership. The person who clicks the merge is the person who owns the consequence. Software that ships without a person on the merge is software that does not have an owner at the moment it goes live. That arrangement breaks the day it has to.

Where teams quietly stop reviewing

There are three predictable points where teams stop reviewing AI output without noticing. They are worth naming because the pattern is consistent.

The successful streak. Six green sprints in a row. The reviewer starts skimming. The seventh sprint slips a defect through because the reviewer's eyes were on the dashboard instead of the diff. The fix is to set a tripwire in the trust ledger: if the streak gets long enough that the reviewer is bored, run an explicit "adversarial review" on the next change. Try to break the tests. Try to find the regression that was not on the watch. The exercise restores the reviewer's attention.

The trivial-looking change. A typo in a copy string. A small refactor. A renamed variable. The change "obviously cannot break anything." The change broke it because the variable was referenced by a string lookup in a place the reviewer did not check. The fix is not to escalate every small change. The fix is to apply the same regression-watch walk even when the watch is "nothing." Confirm "nothing" is actually nothing.

The end of the sprint. Friday afternoon. Two stories left. The reviewer wants to go home. The reviewer skims. The defect ships on Monday. The fix is structural, not personal. Do not schedule reviews of high-risk changes on Friday afternoons. The structure costs nothing. The structure protects the team from its own circadian honesty.

A quiet thing about the mirror

A careless review is the moment the code remembers a defect nobody chose. The defect did not have to be there. Nobody decided to put it there. It got carried forward because the review that would have caught it was rushed, or skipped, or applied to the wrong layer. The code preserves it the same way it preserves any other condition the work was done under.

A thorough review is the opposite. The code preserves the care the same way. Two reviews of the same diff can produce the same green check and a different reflection in the codebase. The reflection is what the team will live with.

A counter-objection

"This sounds like the AI is making us slower, not faster."

It is making the build faster and the verification more important. Both happen at the same time. A team that absorbs the build acceleration without absorbing the verification work gets the worst of both worlds. The faster build. The slower confidence.

The teams that come out ahead are the ones that move some of the review investment upstream, into the acceptance criteria and the regression watch (so the review is *short* because the contract was *clear*), and some of it downstream into the plan-versus-actual loop (so the next sprint's review is *calibrated* by what last sprint actually shipped). Both ends matter. The middle is where the team would otherwise drown.

A small habit

After every sprint, the team spends ten minutes reading the trust ledger. They name out loud which modules moved up, which moved down, and why. They pick one module that moved down (or stayed low) and propose one small change to its conventions, docs, or tests that would move it up next sprint. They write the change as a small ticket.

Ten minutes a sprint. Twenty minutes a month. The most useful retrospective the team has, because the data is on the table, and the action is small enough to do.

Myth. A team that runs review well does not need a trust ledger.

Reality. A team that runs review well writes the ledger by accident, in standups, in Slack threads, in the words "we had a thing in Billing again." The ledger is what makes the words actionable. Without the ledger, the team has the data and not the discipline. With it, the team can act on what it already knows.

The bridge

The work has been refined, built, and reviewed. The team has every reason to believe the right thing happened. The next problem is what the team forgets the moment the sprint closes.

Estimates are wrong in the same direction quarter after quarter, because the team has no memory of the previous misses. Modules disappoint in the same patterns, because the patterns are in people's frustration, not in any artifact. Reviews catch the same kinds of things, because the things they caught last time are not written down anywhere. The next chapter is about turning this memory from a feeling into a system. It is the loop that makes everything in this book compound.

The Memory That Makes Estimates Better

A team that estimated the same thing wrong three times

One team had a quiet pattern. They built integration features. They were good at them. Every sprint, they took on one integration story, the engineer estimated it at three days, and the story took five.

Sprint one. *We underestimated, the partner's API was weirder than expected.* The team nodded. The story shipped. The team moved on.

Sprint two. Different partner, same kind of work. Estimated at three days. Took five. *We underestimated, this partner's auth was different.* Team nodded. Story shipped. Team moved on.

Sprint three. Different partner, same kind of work. Estimated at three days. Took five. The retro said *we underestimated.* The retro

had said that, in those exact words, twice already this quarter. Nobody noticed.

The team did not have an estimation problem. The team had a *memory* problem. Each retro said something true. Each retro stopped saying it the moment the retro ended. The pattern was visible only if someone wrote down the estimate and the actual, side by side, three times in a row. Nobody did. The pattern lived in the team's frustration, not in the team's notes.

The fix is the chapter.

> *Estimation is not a skill. Estimation is a system. Most teams do not have the system, then they wonder why the skill never improves.*

What the loop is

The plan-versus-actual loop is a small habit with three parts.

At refinement, you record the estimate. Effort, duration, lane, expected risk, expected complexity. Not just a number of story points. The structured version: how long, what lane, what could go wrong, what the team thinks is novel.

At end of work, you record the actual. How long did it really take, what lane did it really need, what actually went wrong, what was actually novel. Three to five minutes per story.

Once every few sprints, you read the records. Side by side. You look for patterns. Which modules are honest in their estimates. Which are not. Which work types blow up. Which engineers are conservative. Which are optimistic. Which retro lines keep repeating.

That is the whole loop. The discipline is in doing it consistently, not in the format.

Why story points are not enough

Most teams that have any plan-versus-actual habit have it as story points. Story points are a single number that compresses two different things: how *much* work there is, and how *uncertain* it is. A five-point story might be five points because there is a lot of code to write, or because nobody knows what the code is. The team estimates both as five and writes off the difference as "subjective."

The subjective difference is most of the cost.

A pattern I have seen work: split story points into two numbers. Size (how much code) and uncertainty (how much we do not know). Same total scale, two axes. Within two sprints the team can tell each other "*this is a four-two*" or "*this is a two-five*" and have a different conversation about the same total six. The four-two ships on time. The two-five does not, and the team plans for it.

You do not have to use the same shape. You do have to decompose the number into something you can learn from. A single number

is a slot machine. You pull the lever every sprint and tell yourself you are getting better at it.

> **Myth.** Estimates get better when engineers get more experienced.
>
> **Reality.** Estimates get better when teams measure them. Without measurement, more experience produces more confident wrong estimates. With measurement, even a junior team improves quickly.

What to record

You do not need a tool. You can use one. The shape that works is the shape that gets filled in. Long forms do not get filled in. Two or three lines do.

For each story or slice, two small entries.

Plan, written in refinement.

> *Effort: 3 days Lane: graduated Expected risk: medium (rate limits on partner API) Expected complexity: medium (we have done this kind of work; novel piece is the response shape) Module: Integrations Type: external integration*

Actual, written when the work merges.

> *Effort: 5 days Lane actually run: graduated, plus one fork freeze (decision pause about auth shape) What actually went wrong: partner sandbox had a different response shape than production; lost a day to that What was actually novel: nothing surprising Module: same Type: same Reopen count after merge: 0*

Two short blocks. Five minutes of typing total per story. The data is now in a place the team can read.

Most teams have all of this data in someone's head, in a few Slack threads, and in a retro deck nobody reads. The data exists. The problem is not that the team does not have the information. The problem is that the information has no shape, so the team cannot use it.

What to look for

Once you have a few sprints of records, you read them differently from how you read a retro. You are not looking for blame. You are looking for patterns.

Module patterns. Estimates land for some modules. Estimates miss for others. The modules where estimates miss are usually the modules with stale docs or unstable contracts. The fix is not better estimating. The fix is fixing the module.

Work-type patterns. Migrations always run longer than estimated. External integrations always run longer than estimated. UI work with new components always runs longer. Bug fixes in stable modules tend to land. You can read this as "we are bad at estimating migrations," or as "migrations need a one-point-five multiplier when we estimate them." The second framing actually changes behavior.

Engineer patterns. Some engineers estimate conservatively. Some estimate optimistically. This is not a problem to fix. This is a calibration to know. When a conservative engineer says four days, plan for four. When an optimistic engineer says four days, plan for six. This is not unfair. This is the team using the data it has.

Lane patterns. Stories estimated for the fast lane that ended up needing the graduated lane. Stories estimated for graduated that ended up needing full. Each one is a refinement learning. The dimensions covered in *The Eight Decisions Every Story Carries* missed something, or the lane rule from *The Process the Work Deserves* either had a gap or was misapplied. The retro line "*we underestimated*" was actually "*we mis-laned.*"

Fork patterns. Stories that had a fork-freeze in them. How long the freeze cost. Which forks recur. A team that sees the same fork twice should be writing a decision packet (as described in *The Decision Before the Code*) that prevents the third.

Figure 13.1

	planned effort	actual effort	planned lane	actual lane	novel at plan	novel at actual
partner X webhook	3d	5d	fast	graduated	no	yes
partner Y webhook	3d	5d	fast	graduated	no	yes
partner Z webhook	3d	5d	fast	graduated	no	yes
billing migration	5d	6d	graduated	graduated	no	no
settings copy	2d	2d	fast	fast	no	no
report export	2d	2d	fast	fast	no	no

Figure 9: Figure 13.1. Plan against actual, read sideways: patterns no retro caught become obvious once the rows are stacked.

A worked plan-versus-actual, three misestimates in a row

The point of the loop is invisible at one row. It becomes obvious at three. Here are three rows from a single team's log, six weeks apart, on the same kind of work.

Story	Module	Type	Plan effort	Plan lane	Actual effort	Actual lane	What went wrong (plan)	What went wrong (actual)
Partner X webhook	Integrations	External integration	3 days	Graduated	5 days	Graduated + 1 freeze	Rate limits	Sandbox response shape

								differed from production
Partner Y	Integration webhook	External integration	3 days	Graduated	5 days	Graduated	Auth shape	Partner's auth was an undocumented variant
Partner Z	Integration webhook	External integration	3 days	Graduated	5 days + 1 freeze	Graduated	None flagged	Sandbox missing fields the docs claimed were present

Three rows. The plan said three days every time. The actual said five days every time. The "what went wrong" column says something different every time, which is why the team never noticed the pattern in retros: each story had a fresh-sounding excuse.

The plan-versus-actual table flattens the excuses. The pattern, for this one team, is: *partner integrations with sandbox-vs-production divergence run consistently longer than planned, by roughly the same factor each time.* The fix is not better estimating. The fix is one of two things. Either the team applies a multiplier to this work type at refinement, and the estimates land. Or the team adds a pre-implementation step ("read the partner's sandbox response shape against production") to the integration

lane, and the divergence is caught earlier, and the actual comes back down toward three days.

Both fixes work. The team picks one and tries it for three sprints. The next three rows of the table either confirm the fix or reveal the next misestimate, which the team now has the discipline to notice.

This is the loop. The data does the work. The team's job is to write the rows.

Where the loop feeds the trust ledger

The plan-versus-actual log is not just an estimation tool. It is the data feed for the trust ledger from the *Review That Protects the Product* chapter. A module that has clean plan-versus-actual records (estimates land, work closes without surprise) earns the team's calibrated trust over time. A module with repeated misses raises its risk floor on the next ticket, which raises the next ticket's lane.

The connection is small and important. Without the log, the trust ledger is opinion. With the log, the trust ledger is observation. Two months in, the trust profile of the team's product is more honest than any senior engineer's instinct, because the instinct is built from the same data, with worse recall.

The record is the team refusing to forget what the work taught them.

The compounding part

The first sprint of records produces little. You have six rows. Anything could be a fluke. You write the records anyway, because the third sprint depends on the first.

The third sprint produces one usable pattern. *Stories with external integrations land at a roughly consistent multiple of their estimate.* You apply the multiplier next time. The next external integration story lands closer to its estimate. The team notices.

The sixth sprint produces three or four patterns. *Stories in the Notifications module land. Stories in the Billing module run twenty percent over. Bug fixes in stable modules land. Refactors flagged as "small" land at twice their estimate.* The team starts adjusting at refinement, before the work begins, instead of explaining at retro, after the work ends.

The twelfth sprint produces something stranger. The team's estimates land more often than not. Sprint planning takes thirty minutes instead of three hours. Stakeholders start trusting the team's commitments. The team starts trusting itself. The retro line "*we underestimated*" stops appearing. It is replaced with smaller, more specific lines. *We did not see the data sensitivity in story 142. The eight dimensions missed it.* That is a different kind of retro. That is a retro the team can act on.

This is the compounding part. It is invisible at sprint one. It is the team's superpower at sprint twelve. The teams I have seen run this loop consistently tend to follow the same shape of curve. The first half of the year is boring. The second half is when the company notices the team has changed.

A team that records plan and actual for six sprints in a row is usually having a different conversation by the seventh. The team will not realize the conversation has changed until somebody else points out that the deadlines stopped slipping.

The hard part

The hard part is not the loop. The hard part is the discipline.

Recording the *plan* feels useful. The team is thinking about the work, the team writes the estimate down, the team feels prepared. Recording the *actual* feels pointless. The work is done. The energy has moved on. Writing down what actually happened takes five minutes that nobody is enthusiastic about. The team skips it once. The team skips it twice. The team has no data three sprints later. The team has the same retro line.

The loop only works if the actual gets recorded. The actual is the half that loses. There is no immediate reward for writing it. The reward is in the next sprint's planning, three weeks away. The reward is in the sprint twelve sprints from now, when the team realizes the deadlines started landing.

The way to make the actual stick: make it part of the story's *definition of done*. A story is not done until the actual is recorded. The workflow can check for it: the ticket cannot close, the merge

cannot complete, or the PO cannot accept the story until the actual is recorded. The PO can refuse to close the story without it. The engineer can refuse to merge without it. Make the loop close before the story closes.

This sounds strict. It is. It is also the only mechanism I have seen that survives the team's natural tendency to skip the recording.

What the loop does for AI

AI gets two things from the loop.

First, AI gets *calibration data*. When AI is asked to estimate a story, it can read the team's plan-versus-actual history and adjust. AI is no longer averaging an industry. AI is averaging your team. The estimates AI produces start being useful, because they are tuned to the specific patterns of your work, not generic patterns from training data.

Second, AI gets *better feedback*. Every story the AI participated in has a recorded actual. The actual tells AI which slices took longer than expected, which acceptance criteria missed, which forks the AI should have spotted earlier. Over time, AI does not just get better at writing code. AI gets better at *predicting where the team will get stuck*, because it has the data.

This is the closest you can get to AI that learns your team without anyone calling it that. The learning is in the data. The data is in the loop. The learning is in the data, the data is in the loop, and the loop is what makes the team remember.

A counter-objection

"This is just metrics. Metrics get gamed."

These records are not for management. They are for the team. They are the team's notes about its own work. If the team is gaming them, the team is gaming itself, which is interesting but not the kind of gaming that hurts anyone except the team.

If a manager reads the records and uses them to grade engineers, the records will become defensive and useless within a sprint. The team will write down what makes them look good. The records will lie. The team will lose the loop. The team will lose the compounding. The manager will have gained one quarter of misleading dashboards and lost a year of compounding improvement. That is a bad trade.

The records work because they are honest. They are honest because no one outside the team uses them as performance evaluation. Protect this. It is the difference between a habit that pays off and a habit that gets quietly abandoned.

A small habit

At the end of every story, the engineer writes the actual block. Two minutes. They paste it into the ticket or into the team's shared log, wherever the team agreed.

At the start of every sprint planning, the PO and the engineers spend ten minutes reading the last sprint's plans and actuals. They are not looking for blame. They are looking for patterns.

They name the patterns out loud. They adjust the current sprint's estimates accordingly.

Ten minutes plus two minutes per story. The cheapest improvement loop a team can run. The most consistently skipped improvement loop a team can run. Pick one.

The bridge

When the eight dimensions are filled, the docs are present, the lanes are picked, the forks are paused, the slices are cut, the acceptance criteria are tests in disguise, and the plan-versus-actual loop is running, something starts to happen to the backlog and to the AI that is hard to describe to someone who has not seen it.

There is still one piece the loop alone cannot supply. Every record in the log has a name on it. Every fork called by the team has a caller. Every story that closes had a person who owned the decisions that shaped it. The next chapter is about that. The owner behind the story is the human counterpart to everything this book has set up, and the reason any of it survives a year of work.

The Owner Behind the Story

A name on a ticket

Open any story in your backlog. Find the field that says who owns it.

Most teams have something there. A PO, sometimes a team lead, sometimes "engineering" as a group, sometimes empty. The field is filled in because the ticket template asks for it. The field is rarely treated as a fact about the work.

A story that does not have a name on it does not have an owner. A story that has a group name on it (engineering, ops, design) has an owner whose decisions are no one's call. A story that has a person's name on it has a single human who can be asked, in plain language, *what did you decide?*

That last shape is the one this chapter is about.

A ticket without a named owner is a wish the team filed under "later." Wishes do not survive contact with refinement, AI, or production.

What ownership of a story actually means

In a team where the AI does most of the building, the language of ownership shifts. The engineer's hands are not the only hands touching the code. The PO is not the only person writing the description. A draft of any artifact (ticket, slice, acceptance criterion, decision packet) can come from the AI in seconds. What is left to own, when the typing is automated?

The decisions. The consequences. The product.

The eight dimensions on a ticket are decisions a person made. The AI may have drafted them from context. The person confirmed them. The lane the work runs in is a decision. The AI may have read it off the dimensions. The person accepted the lane. The fork that gets named, the option that gets chosen, the slice that gets approved, the ticket that gets pushed: each one is a decision. Each one has a person whose call it was.

The split is honest. The AI supplies the speed. The person supplies the judgment and carries the consequence. The two roles do not blur. They cannot, because when something goes wrong

six weeks from now, the right question is not "what did the AI do?" The right question is "who decided?"

A story without that answer is a story whose decisions will eventually be attributed to no one. The team will call it "scope creep" or "a misunderstanding" or "process." The team will be too kind to itself. The story did have a moment where it could have been called differently. The team did not name the person whose moment that was.

Why ownership cannot move to the model

There is a temptation, when the AI starts producing draft tickets and draft code and draft tests, to treat the AI as a member of the team. Some companies have done it as a policy. They name the AI in standups. They give it a Slack handle. They attribute work to it in retrospectives.

The naming is fine. The attribution is the problem.

Ownership of a decision cannot move to a model. The model has no continuity. It will not remember the decision next Tuesday. It will not be in the room when the consequence shows up. It cannot be asked what it was thinking, because what it was thinking is not retrievable in any way that is useful. The decision the model made is, in practice, the decision the person who approved the model's output made. Pretending otherwise does not change who pays for the consequence.

This matters because of where AI is in 2026, not because of any moral position. AI is fast, fluent, and useful. It is also unreliable on its own across long, multi-step work: the 2026 APEX-Agents

benchmark of professional banking, consulting, and legal tasks reported a top first-attempt score of 24.0%. That does not mean every AI output is wrong. It means ownership cannot move to the model, because the model is not accountable for the consequence and will not be in the room when the consequence arrives. An organization that treats AI as the owner of its work is an organization that has decided, accidentally, to ship the first attempt.

A thinking backlog distributes ownership without diffusing it. Every story has a person. Every decision packet has a caller. Every merge has a human gate. Every plan-versus-actual record has a name attached to the original estimate. The names are not for blame. They are for the next conversation. When something is unclear, the team knows who to ask. When something is wrong, the team knows where the call was made and can revisit it. When something is right, the team knows whose judgment to lean on next time.

Myth. Distributed teams plus AI means ownership has to be collective.

Reality. Distributed teams plus AI means ownership has to be *more* explicit, not less. The number of decisions per sprint has grown. The number of people who can be in the room shrank. The only way to keep the work coherent is to name the person whose call each decision was, in the moment it was made.

The named caller

A practice that scales surprisingly well: every decision packet carries a *named caller*. Not a role. Not a team. A person.

The earlier chapter on forks introduced the decision packet shape. Fork in one line, options as a list, recommendation, confidence, caller. The caller field is the load-bearing one. It is the person whose call this is. They are the one who can resolve the packet by writing the answer.

When the caller is filled in correctly, three things happen. The packet resolves faster, because the team knows who to walk over to. The decision survives longer, because the team can point at the caller when the decision comes up again next quarter. The team gets quietly better at routing the calls, because the same person seeing the same kind of fork five times in a row stops being a coincidence and starts being a small specialty.

When the caller is filled in poorly (a team name, an empty field, "TBD"), the packet sits. The work it blocks sits with it. The team learns to route around the packet, which usually means making the decision somewhere else, without recording it. The decision now lives in someone's head. The next person to encounter the fork makes it differently, because they read a different head.

Caller is a small field. Caller is most of how the system works.

The named gate

The other place ownership has to live is the merge gate. The chapter on review made the case that auto-merge is a failure

mode that teams reach inside a year of running AI agents. The merge gate is the antidote.

The gate has a person. The person has read the tests, walked the regression watch, and clicked the merge. The person is named on the pull request. The person can be asked, three months from now, *what did you check?*

A team that runs merges this way does not have to ask whether the AI's output was reviewed. The PR has a name on it. The name is the review.

A team that runs merges without this gate has a more delicate problem. When a defect ships, the team has no person to ask. The team has a process. The process did not catch the defect. The team will hold a meeting about the process. The meeting will produce a change to the process. The change will not be assigned to a person, because the original process did not have a person either. The cycle will repeat.

The way out is small. Names on merges. Names on packets. Names on the plan-versus-actual records. Names on the trust-ledger updates. Each name a real person who will be in the room when the consequence shows up.

The backlog as a reflection of the people

The previous mirror chapter argued that code reflects the conditions that made it. The same is true of a backlog. A backlog reflects the people who maintain it.

A backlog with named owners, named callers, named gates, and a clean plan-versus-actual log reflects a team that has decided the work matters enough to put names on. A backlog with empty fields, group ownership, and unresolved decisions reflects a team that has decided, without saying so, that nobody is going to be on the hook.

Both backlogs do the same thing on Monday morning. Both produce a list of stories the team will work on. The team that maintains the first backlog will, six months from now, be running a different kind of company than the team that maintains the second. The difference will not be a tooling difference. It will be the accumulated weight of months of named accountability versus months of diffuse accountability. The diffuse version is the more expensive one. It just hides the bill in a different place.

A living backlog is a living reflection of the people who keep it. The reflection is not flattering or unflattering. It is honest.

What this does for AI

A team that has named owners gets a quiet benefit from the system-aware AI. The AI's drafts are addressed to a person, not to a void. The decision packets the AI surfaces have a recommended caller. The slices the AI proposes have an assigned PO. The merge gates the AI generates have a default reviewer.

This sounds small. It changes the energy in refinement. A draft the AI produces with "Caller: PO with security review" routes itself. A draft the AI produces with "Caller: someone" routes nowhere. The team that has named the owners on its modules and its decisions teaches the AI to address its drafts to real people, because the AI is reading from the same map the team is.

The AI's job is not to decide who owns the work. The AI's job is to surface the work to the owner the team already named. If the team did not name the owner, the AI will either guess (badly) or address the draft to the void (more badly). The team's structure becomes the AI's routing table. A weak table produces weak routing.

Where ownership goes wrong

There are three patterns where ownership breaks in AI-heavy teams. They are worth naming because the patterns are repeatable and avoidable.

The "AI did it" pattern. A defect lands in production. The team traces the change to a slice the AI implemented. The retrospective concludes that the AI made a mistake. The action item is to "tune the prompt" or "add a step to the agent workflow." Nobody is named. The next defect lands two sprints later, in a different module, with a different prompt. The retro repeats. The fix is to name the person who approved the slice, the person who clicked the merge, the person whose call the regression watch was. The defect is then traceable to a human decision. The decision can be examined and improved. Without

a name, there is nothing to improve, because the cause is the model, and the model is not in the room.

The "shared backlog" pattern. A backlog the team owns collectively. Tickets without assignees. Refinement without a designated lead. The team feels democratic. The work moves slower than it should because, at every decision point, nobody is the one to make the call. The PO ends up making most of the calls anyway, by default, and resents it. The fix is not less democracy. The fix is named callers on the things that need calling and a clear PO on every story. The democracy moves to *which* stories to do, not *who* owns each one.

The "AI as junior engineer" pattern. The team starts talking about the AI the way it would talk about a junior engineer. *The AI got it wrong. The AI needs more training. We should pair more with the AI.* The framing is friendly. It is also wrong, because a junior engineer has continuity, growth, and a stake in their work. The AI has none of those. The framing causes the team to delegate ownership to the AI in ways it would not delegate to a real junior. The fix is to keep the language clean. The AI can participate in the work. It cannot own the consequence. The decisions are still the team's.

A counter-objection

"Naming people on everything sounds like a blame culture."

It is the opposite. A blame culture is one where, when something goes wrong, the team searches for someone to attribute the failure to. A named-owner culture is one where, before anything

goes wrong, the team has already attributed every decision to the person whose call it was. The names are written *in advance*. When something goes wrong, the question is not "whose fault?" The question is "what was the call, and what would we change about how it was made?" The conversation is operational, not interpersonal.

A team that names owners well does not point fingers more. It points fingers less, because the answer to "whose fault?" is "nobody's, the call was made on the information available at the time, here is what we learned." A team that does not name owners spends more time arguing about cause than it does about cure.

A small habit

Once a week, the PO reads through the open decision packets and confirms each one has a caller. Not a role. Not a team. A person. Where the caller is missing, the PO names one, or escalates to the engineering lead. Five minutes.

Once a month, the PO reads through the closed packets from the previous month and notes which callers recur. A pattern of "the same person keeps being the caller on database forks" is not a problem. It is a discovery. The team has a database owner. The team should make that explicit and route accordingly.

These are not governance habits. They are routing habits. They protect the team from the most expensive failure mode in an AI-heavy backlog: a decision that nobody makes because nobody was named.

The bridge

The previous chapters built the system. The eight dimensions, the documents, the system the AI reads, the lanes, the forks, the slices, the acceptance criteria, the review, the trust ledger, the named owners. Each one is small. Each one is concrete. Each one has been described in enough detail that the team can start applying it on Monday.

The next chapter is what happens when the team has done all of it for long enough that none of it feels remarkable anymore. The work is being done. The standups are short. The customer-support queue is calm. The mirror is reflecting something the team would not have predicted twelve months ago. That is where the book ends, on a quiet sprint, with the team looking at something they built without quite realizing how they got there.

The Backlog That Thinks

A quiet sprint

It is the second-to-last day of the sprint. The team is in standup. Nothing strange is happening. The four stories that were planned are at the stage they should be on this day. The engineer working on the integration is in the third of six slices. The PO is reading the acceptance criteria for the story coming up next week. The new hire has a question, and the question has a documented answer, and she found the answer before standup so she could ask a better one.

A founder walks past the standup. The standup ends in twelve minutes. The founder looks at her phone, then at the team, then at her phone again. *Is everything okay?* she asks the lead engineer. *Yes, why? It's just quiet.* The engineer thinks about it. *Yes,* he says. *It has been quiet for a while.*

The team is shipping. The customers are not filing tickets about half-built features. The estimates are landing. The retros are short. Nobody has called it good, because nobody has a frame for what is happening. The team is doing the work. The work is being done. The unremarkable nature of the situation is itself the remarkable thing.

This is what a backlog that thinks feels like. It does not feel like a breakthrough. It feels like the absence of a problem the team had stopped noticing.

The reward for doing this work is invisible. It is the sprint where nothing strange happens. It is the customer call you do not have to make. It is the Friday afternoon that ends on time.

What is in place

Take the team apart and look at what they have built without quite naming it.

The eight dimensions are filled in on every story. Workflow, modules, scope, risk, complexity, database impact, API impact, UI impact. They live in a short block at the top of every ticket. Nobody talks about the dimensions. They are just there. New tickets do not get refined without them. The PO writes them as part of writing the ticket.

The three foundational documents exist. The module map. The stack rules. The design system. None of them are long. All of them are honest. They get updated as part of the work that changes them, not as a separate project. Refinement reads them. Implementation reads them. Reviews read them. AI agents read them. The team's product knowledge has a shape that survives turnover.

The wiring is in place. The AI runs with read access to the codebase, a connection to the ticket system, and access to the docs. The PO drafts a system-aware ticket in two minutes instead of ten. The eight dimensions arrive pre-filled, against real modules and real schemas. The team reviews, corrects, approves. The AI pushes only on the human gate.

The mirror is on the team's mind, even if nobody says it that way. The team writes the dimensions, freezes the forks, and updates the docs because they understand that the code will reflect every shortcut and every piece of care that goes into it. The discipline is not about productivity. The discipline is about what the code becomes a record of.

Lanes are picked at refinement, not at the moment work begins. Fast lane stories ship in hours. Graduated lane stories ship in a few days. Full lane stories ship across sprints with the right ceremony. Nobody argues about which lane. The eight dimensions point at the lane and the team reads it off. The argument moved upstream, where it is cheap.

Forks are named. The team has built a small library of decision packets over the months. The library is not impressive. It is invisible. It is also why the same fork does not get debated

twice. New tickets that hit a fork the library already covered get resolved in two minutes. New forks get named, frozen briefly, called, recorded.

Slices are the unit of work. The team does not ship two-week stories anymore. The team ships one-to-three-day slices. Each slice has its own acceptance criteria, its own doc target, its own regression watch. The sprint board has more boxes than it used to. Each box moves more often than it used to. Velocity, by any honest measure, is up. The team did not become faster. The team stopped trying to finish things that could not finish.

Acceptance criteria are tests in disguise. Three-part shape. Positive, negative, edge. Each criterion has a test that proves it. *Done* is no longer an opinion. *Done* is the green check next to the test. The PO and the engineer agree on done because the criteria are the contract. The reopen rate has dropped, whether the team tracks it formally or simply feels the absence of reopened work.

Refinement looks different. Most stories go from intent to a pushed ticket in one short session: the AI drafts from context, the team corrects, a fork is surfaced if there is one, slices are cut, the ticket is pushed on approval. What used to be a meeting is now a review.

Review looks different too. Tests get read by name, not by line. The regression watch is walked, not skimmed. The trust ledger calibrates how deep the next review goes, per module. A human gate sits on every merge, with a name on it.

The plan-versus-actual loop is running. Each story has a plan block and an actual block. The team reads them at sprint

planning. The patterns the team finds are specific to their work and their modules, and the team estimates accordingly. Estimates land. The next set of patterns becomes visible. The compounding has started.

Ownership has names. Every decision packet has a caller. Every merge has a human gate. Every plan-versus-actual record has the name of the person whose estimate it was. The names are not for blame. They are for the next conversation, three months from now, when the team needs to know who to ask.

Each piece is small. The pieces together are a different thing.

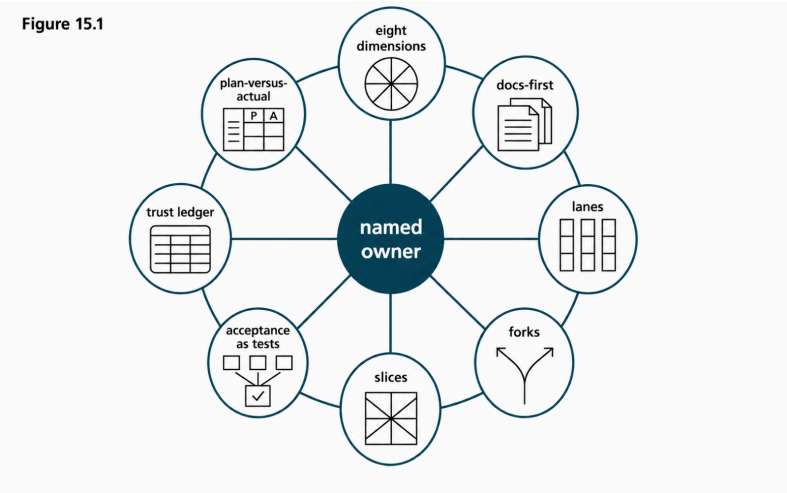

Figure 10: Figure 15.1. The system, assembled: eight small pieces around a single human at the centre, each piece strengthening the others.

What changes about AI

When all of this is in place, AI in your team becomes a different kind of collaborator. The change is hard to describe without sounding promotional. It is also undeniable when you see it.

The AI stops asking for clarification on the obvious cases, because the obvious cases are listed in the acceptance criteria. The AI proposes the right shape for new work, because it has read the team's decision packets. The AI writes tests as part of implementing slices, because the tests are derivable from the criteria. The AI updates the documentation, because the doc target is part of the slice. The AI flags work it cannot complete, because the lane rule is clear and the AI can apply it.

The team stops asking the AI to make product decisions. The product decisions are settled. The team stops asking the AI to invent missing context. The context is documented. The team starts asking the AI to do what it is good at: implement well-specified work, fast, with tests, against settled ground.

The AI supplies speed. The team supplies judgment. Each side does the part it is good at. Neither side does the other's job badly.

The team's AI has stopped averaging. It is reading the team's product, the team's decisions, the team's eight dimensions. The averages it used to fill the blanks with are no longer the path of least resistance, because the blanks are filled.

Myth. Better AI will fix your delivery problems.

Reality. Better AI amplifies whatever you give it. Vague tickets in, more vague work out, faster. A backlog that thinks in, more good work out, faster. The amplifier is not the answer. What you feed the amplifier is.

The compounding nature

This system pays in months and quarters, not in sprints.

Month one feels slow. The team is writing docs, learning to name forks, getting used to writing acceptance criteria as testable statements. The output looks lower than usual. The team's leadership has to be patient. This is the cost of building the system, paid up front.

Month three feels normal. The docs exist. The forks are named in habit. The acceptance criteria are no longer a stretch. The team's output is back to where it was, with less churn.

Month six feels different. The team is moving on settled ground. The retros are short because there is less to complain about. The customer-support tickets about ambiguous features have thinned out. New hires onboard faster because the docs are real. The team's collective memory is no longer in two engineers' heads.

By around the twelfth month, the company is shipping work that does not break, in lanes that match the work, with estimates that land. The team's reputation has changed inside the company. The team's reputation has changed with customers. Nobody can

quite point at the moment it happened. It happened in the work, story by story, sprint by sprint.

The team that gets here did not get a budget for it. The team that gets here built it by doing the chapters of this book, one at a time, on real stories. The investment is small. The discipline is consistent. The reward compounds.

Teams that run this kind of system consistently for a year are usually the most surprised by their own delivery six months later. They cannot pinpoint when it started feeling different. The change is gradual and total.

What this is not

This is not a methodology, a framework with a name, or something you buy. Most of what is in this book is what good senior engineers and good product owners do anyway, on the days they have time. The contribution of this book is naming the parts and making them habits.

It is not a substitute for talent. A team with bad engineers running this system will still ship bad code, slowly. A team with good engineers running this system will ship good code, faster. The system amplifies what you already have. It does not replace it.

It is not a guarantee. Things still go wrong. Customers still surprise you. Markets shift. Strategies change. The system does

not prevent surprise. It prevents *self-inflicted* surprise. That is most of the surprise teams report. The rest of the surprise is what makes the work interesting.

The backlog that thinks is not a thing you buy. It is a habit you keep. The reward is invisible until you have kept it for a while. Then it is the only way to work.

A note on humility

If you finish this book and decide your team is already doing most of it, you are correct. Most teams that are shipping well are doing most of this. They are doing it without names, often without realizing it. The book did not invent the work. The book named the work and made it transferable.

If you finish this book and decide your team is doing none of it, you are also probably correct. Some teams have built around the absence of this system for so long that the absence has become the culture. The book is not for that team's culture. The book is for the engineer or the founder inside that team who wants to start. Pick one chapter. The fastest one to start with is the eight decisions. Add a row to the ticket template. Watch what happens for a sprint.

If you finish this book and decide your team is doing some of it, you are most likely correct. Most teams are doing three or four

of the pieces well and skipping three or four. The pieces being skipped are usually the same ones across teams: docs-first, forks, plan-versus-actual. Those three are the ones that hurt the most to start and pay the most when they compound.

The mirror, paid off

There is a quiet thread the book has carried from the first chapter. A codebase reflects the conditions that made it. Rushed work carries the rush forward. Vague work preserves the guess. Clear and owned work carries the care forward in the same way.

The team in the quiet sprint above did not know it was paying off the mirror. The team was just shipping. From the outside, the work looked unremarkable. From inside the code, the work was something else. The code remembered the team's habits and reflected them back as a product that holds up.

This is the reward the book has been pointing at the whole time. It is not a metric. It is the moment, sometimes weeks later, when a customer says something offhand about how easy a flow is to use, and the engineer who built it remembers the Tuesday the team froze a fork for thirty minutes to make a real call. The customer does not know about the freeze. The customer just knows the flow works. The team's care is in the customer's experience, traceable back to a decision made well by someone who is no longer in the meeting where it was made.

That is what a backlog that thinks produces. Not faster output. Better reflection.

The split, said once

The book has avoided saying this in many places on purpose. It is worth saying clearly once.

The work in this book runs on a split. The AI does the typing. The team does the thinking. The AI reads context and drafts work that is already true about most of the things that could be checked from code. The team decides what cannot be decided from code: which feature matters, which fork to call, which lane to run, which trade-off the company is willing to live with. Each side does the part it is good at. Neither side does the other's job badly.

A team that runs this split well does not have to ask whether AI is in charge. AI is never in charge. AI is the fastest first reader the team has ever had. The team is the author.

The tagline of the series is *AI Builds, You Own*. This book is about the first place ownership has to start: the backlog.

The reward

The reward will be quiet. It will sneak up on you. One day, in standup, somebody will ask if everything is okay, because it is unusually quiet, and you will realize the quiet has been there for a while. The customer-support queue is shorter. The sprints end on time. The team has stopped explaining confusing behavior that should never have reached the customer. The estimates are landing. The retros are short.

The mirror is on the wall. It has been on the wall the whole time. The team is finally happy with what it sees.

That is the backlog that thinks.

Case Studies

These case studies are composite and illustrative. The teams and tickets are based on real engagements and patterns observed across multiple companies. Specific names, numbers, and details have been altered, combined, or simplified to protect the teams involved and to make the lessons portable. Where numbers appear, they are honest within rounding and worth what their context says they are worth: not benchmarks for your team, but anchors for the shape of the change.

The studies use the language and habits the rest of the book introduces. Read them after the chapters they reference, not before.

Case Study A: From averaged tickets to system-aware tickets

The team

A B2B SaaS company, around forty engineers, three product owners. The product is a marketing automation platform used by mid-market customers. The codebase is a TypeScript monorepo with about 480k lines, eleven services, a React frontend, and a Postgres primary plus three read replicas. The team has been shipping for six years. The team uses Jira for tickets, Confluence for docs, and a mix of Cursor and Claude Code for AI-assisted development.

Where they started

In the year before the change, the team had quietly adopted AI tools without changing any other part of their workflow. Each engineer used the AI in their editor. The PO wrote tickets the same way she always had: a paragraph of description, two or three acceptance criteria, a story-point estimate decided in refinement.

The tickets the AI wrote were indistinguishable from the tickets she wrote. The tickets the engineers built from those tickets were indistinguishable from the work they had been shipping the year before. The output volume was higher. The reopens were also higher.

The team measured three things over the previous two quarters and was honest about them.

- Reopened stories per sprint: averaged around four. The previous-year baseline was around two.

- Customer-support tickets traceable to "this is not what we expected" within thirty days of release: up roughly a quarter.
- Time spent in sprint planning: about the same as the previous year, but felt longer, because more decisions arrived ambiguously.

The team's leadership had a sentence they repeated to themselves: *the AI is making us faster, but the quality is slipping.* Both halves were true. The connection between them was where the team had not done the work.

What they changed

The change was wiring, docs, and one habit. It took a week of one engineer's time and ten minutes of every team member's time during refinement for the following sprint.

Wiring. They added an MCP configuration to their AI tooling. The AI was given read access to the codebase, read-and-draft access to Jira via an MCP server, and read access to Confluence. The push-to-Jira step was gated behind explicit human approval. Total setup time: a Friday afternoon for one engineer.

Docs. They wrote two of the three foundational documents the docs-first chapter describes. The module map (which they had partial drafts of) was completed in three days. The stack rules (which lived in two engineers' heads) were dictated in two hours. The design system already existed.

One habit. The PO ran every new ticket through the AI in refinement first. The AI produced a system-aware draft. The PO reviewed, corrected, approved, and pushed. Refinement time per ticket dropped from about ten minutes to about three.

What changed

The team kept measuring the same three things over the following quarter. The numbers below are illustrative of the shape of the change, not promises about your team.

- Reopened stories per sprint dropped from around four per sprint to around one.
- Customer-support tickets traceable to "this is not what we expected" within thirty days of release dropped to slightly below the pre-AI baseline. The team had not been this calm on that metric in two years.
- Sprint planning time dropped by about a third. Refinement of individual stories shortened more dramatically. The team's discussion in refinement was no longer about "what did we mean by this?" It was about the specific dimensions or forks the AI had flagged for human resolution.

There were three less-obvious changes that mattered more than the numbers.

The PO stopped being the only person who could refine a ticket. Two of the three engineers started running system-aware drafts on their own tickets before refinement. The team's refinement meeting became a review of drafts, not a generation of them.

The team started catching forks early. The AI's drafts surfaced an "Open fork" section on about one in five tickets. Most of those forks had previously been buried in implementation and re-discovered in QA or in production. The cost of catching them in refinement was a five-minute walk to the engineering lead's desk. The cost of catching them in production had been days.

The customer-success queue got quieter. This was the change leadership noticed first. The team had not done anything visible to customers. The customers, without knowing why, started filing fewer "this is not what we expected" tickets. The change was the silent version of the work the team had done upstream.

What it cost

The team did not get a budget for the change. They did not buy any new tools. They did not hire any new people. The change cost one engineer about a week of setup, plus extra attention in refinement during the first month while the team adjusted to reviewing AI drafts instead of writing tickets from scratch. Once the habit settled, per-ticket refinement time dropped from about ten minutes to about three.

The change also cost the team a few weeks of patience. The first two sprints felt slower in a way that was hard to defend, because the team was adjusting to a new shape of work. By the third sprint, the new shape was faster on every measurable axis, and the team had stopped asking whether the change was working.

What it required to stick

The change stuck because the team made the system-aware draft the default starting point, not an optional add-on. The PO did not have a choice about whether to run the AI in refinement. The AI was the first step. The PO was the second.

The change also stuck because the engineering lead made the docs part of the work. Whenever a story changed a module's

boundary, the module entry was updated as part of the slice. Whenever a story added a new convention, the stack rules were updated. The docs did not become a project. They became part of every story, costing about ten minutes each and saving hours of re-explanation later.

What it did not change

System-aware tickets did not change the team's velocity in a meaningful way. The team did not ship more stories per sprint. The team shipped roughly the same number of stories with significantly fewer reopens and fewer customer-visible defects. The acceleration was in the cost the team had been paying for being wrong, not in the speed the team was right at.

System-aware tickets also did not change the team's product strategy. The AI did not pick better features. The PO and the company's leadership still owned that work. What the AI changed was the precision of the contract between a feature decision and the code that implemented it.

Case Study B: What chapter 9 looks like in a real team

This study is the same story as the thirteen-pointer in *The Work Small Enough to Finish*, with the people in it. The slicing chapter teaches the move; this case shows the slicing session, the retrospective conversation it produced, and the small policy changes the team kept afterward.

The story everyone had stopped looking at

One team had a story in their backlog called *Customer can manage their notification preferences.* The story had been estimated at thirteen points. It had moved from sprint twelve to sprint thirteen to sprint fourteen. By sprint fifteen, the team had stopped putting it in active rotation. It sat in the backlog the way a guest sits in a kitchen at the end of a long party: present, unmoving, no one quite ready to address it.

The team's retrospective notes contained the same line in three consecutive retros. *We underestimated.* The team had stopped writing it down by retro four, because the line had begun to feel embarrassing.

The story was not too hard. The story was not under-resourced. The story was not the wrong priority. The story was *one large piece of work pretending to be one unit.* It was six pieces of work bundled into a single ticket. The team had estimated the bundle. The bundle could not finish, because finishing was never possible. Finishing required six smaller finishes that nobody had named.

The slicing session

The team set aside ninety minutes with their PO, two engineers, and a connected AI. They opened the ticket. They asked the AI to propose slices according to the team's slicing rules: each slice one to three days, reversible, mergeable independently, with a doc target and a regression watch.

The AI proposed six slices. The team accepted five and re-cut one. The final list was the version that appears in the slicing

chapter of this book (data model, read endpoint, write endpoint, screen read-only, screen editable, behavior wired in).

The session took seventy minutes. By the end of it, the thirteen-pointer was no longer in the backlog. Six slices were, each between two and three story points, each with its own acceptance criteria and regression watch.

What happened next

The slices entered the team's next sprint as the highest-priority items. The team ran them in dependency order. Each slice merged independently. The team's pull-request count for the sprint was visibly higher than usual. The team's velocity, measured in story points closed, was higher than the team had seen in two quarters.

The slices took about two working weeks end to end. The original story had already crossed multiple sprint boundaries without finishing.

But the time was not the headline.

What the team did not expect

The thirteen-pointer was the team's only example of "the story that would not die." When it finally shipped, the team's retrospective skipped its usual format and turned into a conversation about why the team had been stuck for three sprints on something that, sliced properly, took about two working weeks.

The conversation produced three observations the team wrote down and treated as policy from that point on.

Most of the team's "underestimates" were mis-bundles. The team went back through six months of retro notes and found that almost every "we underestimated" story was a bundle that should have been slices. The team started slicing every story above five points by default, and most of the bundle-shaped stories disappeared.

The team's morale around long stories had been quietly bad. Nobody had said anything, but two of the engineers admitted that pulling a thirteen-pointer off the board had felt like committing to something they were not sure they could finish. They had been pulling smaller, less-impactful work first as a result. The shape of the team's velocity over the previous quarter, viewed in this light, was easier to explain.

The team's standups had been protecting the same status. *Still working on it. Should be done by Friday.* For three sprints in a row, the same two engineers had been reporting the same status on the same story. The team had stopped asking, because asking felt unkind. The slicing made the standup useful again. The slices closed. The status was different every day.

What the team learned about AI in this process

The AI did not save the thirteen-pointer. The team saved the thirteen-pointer by deciding it was not one piece of work. The AI was useful because the team finally let it draft slices against the actual codebase, instead of asking it to "make a plan" against a sentence.

The slicing took the AI about twelve seconds. The team's review of the slices took an hour. The hour was the work. The twelve seconds was the speed. The AI proposed; the team decided.

What did not change

The team did not become faster at all kinds of work after this. The team became faster at *finishing*. Their velocity, six months later, was meaningfully higher than the previous baseline. Most of the gain came from no longer having work that lingered.

The team also did not stop bundling everything. The pattern reappeared on three or four stories over the next quarter, usually when the original requester was outside the team and described the work as "one feature." The team learned to slice anyway, even when the requester thought it was overkill. The requester always changed their mind two weeks later when the first slice shipped and they could see something working.

What it took to stick

This change stuck because the team treated slicing as a rule, not a preference. Every story above a certain size went into the slicing session, no exceptions. The team also added a small habit to refinement: every story, regardless of size, got asked the question "can this be sliced?" If the answer was *no, it is one slice already*, the team marked it as such and moved on. The question was free. The question protected the team from re-discovering the bundle pattern at the cost of three sprints again.

What both studies have in common

Both teams started in the same place. Both teams had AI in their workflow. Both teams had docs that were partly written. Both teams had retros that kept saying the same things. Both teams shipped work the customer eventually objected to.

Both teams ended in a different place. Not because they bought something. Not because they hired someone. Not because they switched to a new framework. Because they put a small set of habits in place, in the right order, and let the system the rest of this book describes compound.

The work in both studies is small. The work in both studies took weeks, not quarters, to install. The work in both studies kept paying back for the rest of the year, and the year after.

This is the part of the book that is supposed to feel anticlimactic. The result is not a transformation. The result is a team that ships what it agreed to, in the time it estimated, with the AI supplying the speed and the team supplying the judgment. That is the destination. The chapters were the route.

> *The reward is unglamorous: a quieter customer-support queue, a sprint that ends on time, a Friday afternoon when nothing strange happens. If that sounds boring, the rest of this book was not for you. If it sounds*

rare, you have already understood why it is worth the work.

Sources and Notes

The studies, frameworks, and ideas referenced in the body of this book are listed here. Where the body quotes or paraphrases a named author, the source is given so the reader can verify the wording and consult the original. Where the body cites a number, the source is named alongside the kind of study it came from, so the reader can place the number in its context.

On AI and developer productivity

Paradis, E., et al. *How Much Does AI Impact Development Speed? An Enterprise-Based Randomized Controlled Trial.* arXiv:2410.12944 (2024). https://arxiv.org/abs/2410.12944. A randomized controlled trial with ninety-six Google engineers. Reported a roughly 21 percent reduction in time on a complex, enterprise-grade task with AI assistance, with a wide confidence interval. The authors caution against generalizing the result across tools, settings, or task types.

Becker, J., Rush, N., Barnes, E., Rein, D. (METR). *Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity.* arXiv:2507.09089 (July 2025). https://arxiv.org/abs/2507.09089 https://metr.org. A randomized trial with sixteen experienced open-source developers across two hundred and forty-six tasks in repositories they knew well. Reported that developers took about 19 percent longer with early-2025 AI tools, even though they believed afterward that the tools had sped them up. The authors note the study is a snapshot of one setting and warn against overgeneralizing.

The two trials taken together are the source of the book's argument that AI is not a fixed multiplier on delivery and that the surrounding workflow decides whether it speeds a team up or quietly slows them down.

On agent reliability across professional work

Mercor. *APEX-Agents: AI Productivity Index for Agents.* arXiv:2601.14242 (January 2026). https://arxiv.org/abs/2601.14242. A benchmark of long, multi-step professional tasks drawn from investment banking, consulting, and law. The top Pass@1 score reported was 24.0%. APEX is not a coding benchmark; it is included in this book because the gap between APEX and coding-specific benchmarks is exactly the point. Long, ambiguous, multi-step work is where agents struggle, and vague backlog work belongs to that category.

SWE-bench Verified leaderboard, accessed May 21, 2026. https://www.swebench.com/. The leaderboard shows top systems resolving a substantially higher share of coding tasks than early agent systems did in 2024. The contrast with APEX is informative: agents do well where the constraints are hard and verifiable (a test passes or it does not) and worst where the constraints are soft and multi-step. Readers should note that OpenAI has publicly called SWE-bench Verified contaminated as of early 2026, because models can reproduce the original patches from task identifiers; the APEX-SWE paper (arXiv:2601.08806) is the source for the contamination claim, and the practical effect is that the SWE-bench-vs-APEX gap is wider in reality than the headline numbers suggest, which only sharpens this book's argument about where agents struggle.

On AI as an amplifier

DORA / Google Cloud. *State of AI-Assisted Software Development* (2025). https://dora.dev/research/2025/dora-report/. The source of the "amplifier" framing used throughout the book. DORA's own wording: "AI's primary role is as an amplifier, magnifying an organization's existing strengths and weaknesses." The specific operational figures cited in some popular summaries of the 2025 report do not come from DORA itself; readers should consult the DORA publication directly for what it does and does not measure.

On code as a reflection of the people who made it

Conway, M. *How Do Committees Invent?* Datamation, April 1968. The original source of what is now called Conway's Law: systems

mirror the organizations that build them. The paraphrase in the body of *The Mirror Inside the Code* is faithful to the original observation.

Brooks, F. P., Jr. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, 1975 (anniversary edition 1995). The lineage section paraphrases Brooks's argument that software is a medium made almost entirely of thought; readers wanting the original framing should consult the printed edition directly.

Knuth, D. *Literate Programming.* The Computer Journal, vol. 27, no. 2 (1984). The paraphrase in the lineage section reflects Knuth's broader argument that programs are written for humans to read.

Hunt, A., & Thomas, D. *The Pragmatic Programmer.* Addison-Wesley, 1999. The "broken windows" idea referenced in the lineage section is one of the book's most-cited metaphors. Readers wanting the original framing should consult the chapter on software entropy.

A note on field observations and composites

Examples in the body of this book that are not attributed to a named source are field observations. Some are drawn from one team's experience; many are composites that combine patterns observed across multiple teams, so that a lesson can be shown without exposing any single team's private work. Where a percentage or ratio appears in such an example, it is included for the shape of the change it describes, not as a benchmark. Other teams will see different numbers in different directions for similar reasons.

The case studies in the back matter are explicitly framed as composites. The numbers in them are illustrative of the shape of the change, not promises about your team.

A note on tools and standards

The book references Model Context Protocol (MCP) and a number of specific AI tools by name (Claude Code, Cursor, GitHub Copilot, Codex CLI, Gemini CLI, Junie, Windsurf, Continue, Aider). These are referenced for illustration only. The discipline the book describes does not depend on any specific tool. The principle is to connect the AI to the team's real sources of truth (codebase, ticket system, docs) through whatever mechanism the team already uses or chooses to adopt.

Standards and frameworks referenced in passing (GDPR's right of access, OAuth, WCAG, OWASP) are referenced for illustration of the kinds of constraints a real ticket has to settle.

Teams operating under any of them should consult qualified professionals for their specific obligations.

Glossary

The terms below appear throughout the book. Most of them are not unique to this book. Where the book uses a term in a specific or non-obvious way, the definition reflects that use.

Acceptance criteria as tests. A criterion written as a precondition, an action, and an assertion, in a form that can be turned directly into a test. Each criterion is one of three types: a positive case, a negative case, or an edge case. Most stories need at least one of each.

Affected modules. The list of named modules a story touches, written in business language, not folder paths. The list drives reviewer routing, regression scope, lane selection, and the AI's context retrieval.

Averaging. What an AI does when given a vague ticket without context: it fills in each unsettled decision with the most common version of that decision from its training data. The output looks confident because the average is confident. The output is rarely yours.

Backlog that thinks. A backlog where every story carries the eight dimensions, points at real docs, knows its lane, names its forks, slices into pieces small enough to finish, defines done as testable acceptance criteria, and records plan against actual after the work closes. The phrase that gives the book its title.

Caller. The named person on a decision packet whose call it is to resolve the fork. Not a role. Not a team. A specific human who can be walked to, asked, and held to the call.

Composite case study. A case study built from patterns across multiple teams, altered to protect privacy and keep the lesson portable. The shape of the change is real. The exact numbers are illustrative of that shape, not benchmarks for any one team.

Context-aware AI. An AI agent connected to the team's real sources of truth (codebase, ticket system, docs) so that it reads the project's reality before drafting any work. The opposite of an AI sitting in a browser tab with only the prompt to work from.

Decision packet. A short, structured record of a fork that needs resolving. Includes the fork in one sentence, the options, a recommendation, a confidence level, a named caller, and (after resolution) the decision and reasoning. Often filed as `D-{N}` with a stable identifier so future tickets can reference it.

Docs-first. The rule that the docs a ticket binds to (module map, stack rules, design system) must exist before the ticket is refined. If they do not, the work pauses and the docs get written first.

DPO. Data protection officer. The person on the team responsible for privacy and data-protection compliance, usually

involved when a story touches personal data, consent flows, or data exports.

Eight dimensions. The eight decisions every refined story must resolve. Workflow type, affected modules, scope depth, risk level, complexity level, database impact, API impact, UI and customer impact (often shortened in tickets to "UI impact"). The chapter on the eight decisions uses *decisions* in the title and *dimensions* in the body. The two words refer to the same shape of work.

Fast lane. The shortest of the three execution lanes. Used for trivial or low-risk work. Six phases. Hours, sometimes less.

Field observation. A claim based on observed team patterns, not a published benchmark. Field observations appear in this book where the shape of a change matters and a precise external statistic does not exist or would not generalize.

Fork. A decision inside a ticket that has consequences outside the ticket. A product or architecture choice that would have been irreversible if it had been decided silently inside an implementation. Forks get named, frozen, called, and recorded.

Freeze. Pausing work on a ticket until a named fork is resolved by its caller. The freeze is short by design. The freeze is the team's most reliable defense against forks decided by reflex in implementation.

Full lane. The longest of the three execution lanes. Used for high-risk or architectural work. Eleven phases. Migrations, breaking API changes, redesigns, anything where reversibility is hard.

Graduated lane. The middle of the three execution lanes. Used for medium work, which is most work. Ten phases. Days to a sprint.

Human gate. The explicit human approval point before an AI-drafted ticket is pushed, an AI-generated pull request is merged, or a customer-visible change ships. The gate is the place where the AI's draft becomes the team's commitment.

Lane. The list of phases a story moves through. There are three lanes: fast, graduated, full. The lane is decided from the eight dimensions, not from a feeling.

Named gate. The specific person responsible for a human gate, named on the artifact. A gate without a name is a gate that nobody owns; the named gate is what makes the approval traceable.

Plan-versus-actual loop. A small habit with three parts. At refinement, the team records the plan (effort, lane, expected risk, expected complexity). At end of work, the team records the actual. Every few sprints, the team reads the records side by side and looks for patterns. The discipline is in doing it consistently, not in the format.

PO. Product owner. The person responsible for a story being well-formed before it enters refinement and for the story's outcome after it ships. Glossed here because the book uses the term often.

Privacy gate. A human review step required before the customer-visible rollout of work that touches personal data, compliance-sensitive flows, or access and export behavior. Often

attached to a graduated- or full-lane story as a precondition for the rollout slice.

Refinement. The act of making the eight dimensions true for a story, picking a lane, naming any forks, slicing if the story is large, and writing acceptance criteria as testable statements. In a system-aware workflow, refinement is mostly the review and correction of a draft the AI produced from context.

Regression watch. A short list, written as part of refinement, of two or three places in the rest of the product that could plausibly break because of this change, and how the team will know if they did. Used as the reviewer's map during review.

Review depth. How deeply a change is reviewed, set by module trust, risk, lane, and recent defect history. Review depth is not a fixed standard; it rises and falls with what the trust ledger has been recording.

Slice. A unit of work with four properties: a bounded scope (named files), at least one testable acceptance criterion, a documentation target, and a regression watch. A slice can be finished. A story missing any of these properties cannot.

System-aware draft. A draft produced by AI after reading the team's codebase, tickets, and docs. It becomes a system-aware ticket after human review and approval at a human gate.

System-aware ticket. A ticket drafted by an AI that has read the team's real codebase, ticket system, and docs first. The dimensions are filled from ground truth. The acceptance criteria reference real error codes. The regression watch names real files.

Distinct from an "averaged ticket," which was drafted from a prompt and no context.

The code remembers. The book's compact phrase for the observation that a codebase reflects the conditions that made it. Rushed work carries the rush forward. Clear and owned work carries the care forward. Used sparingly in the book on purpose. Once the team understands it, the phrase does not need repeating.

Trust ledger. A small per-module record of recent AI-related defects, review findings, recurring patterns, and current review depth, used to calibrate how deeply the next AI-built change is reviewed. Trust is observed, not granted. Trust is revocable when the data changes.

Workflow type. The first of the eight dimensions. The kind of work a story is: feature, bug fix, migration, refactor, content update, research spike, documentation update, root-cause analysis, security pentest, and so on. Different types route through different lanes and different review depths.

About the Author

Haider Lasani has spent more than sixteen years in software delivery: shipping product, advising teams, and watching what separates the companies that trust their own delivery from the ones that do not. The pattern across the failures kept being upstream of the code. The pattern across the successes kept being a small set of habits that nobody had named. This book is the names.

He lives in the Netherlands. He writes for the people responsible for what ships next, and for the teams who have to live with what shipped last. He believes that the part of the job that does not generate a meeting invite is the part that decides whether the company exists in two years.

He reads more than he writes. He writes more than he speaks.

This is the first of five books in the series *AI Builds, You Own* the other four are introduced in the front matter and cover the layers downstream of the backlog.

The system in this book is not mine. It belonged to every good engineer and every good product owner I have worked with. My contribution was to write it down.