Left of the Loop

As implementation becomes abundant, shared understanding becomes the scarce resource

Simon Schrottner

REVIEW DRAFT — July 2026 · not for distribution

Contents

A note on how this book was made

This book was written with Claude. The ideas, the stories, the framing: those are mine. But I'm not a natural writer. I struggle to structure an argument, and I miss my own contradictions. AI closed that gap. It helped me find holes I couldn't see because I was too close to the material. It made writing this possible at all.

This book argues that AI can quietly erode the friction that produces shared understanding when we let it replace the conversations that build it. I'm aware of the irony in using AI to write that argument.

Claude helped me think more clearly about arguments I already held. The moments that changed the book came when other people challenged those arguments from outside my own framing. I tried to build that friction back in the only way I had: publishing the argument as a blog series before finishing the book, and letting readers push back in public. Matthias asking "why do I need the team" did more for this book than any private editing pass could have. That pushback was the same mechanism this book argues for, just run in public instead of in a room.

Matthias Meisinger, Johann Blauensteiner, Bernhard Dieber, Carla Koeberl, and Giulia di Pietro read the first version of this book. It was the rough one, before any of this held together, and they told me what wasn't working. Thank you for getting through it.

I still don't know if that was enough. Judge for yourself. That's what the rest of this book is for.

Introduction

This book is not quite what it looks like.

It looks like a book about AI: about agents, specs, workflows, and the changing shape of software development. Those things are in here. But they're not what the book is about.

What the book is about started with a conference talk, a university friend, and 18 PlayStation Move controllers.

We were building a demo for KubeCon. The talk was called "18 Bluetooth Controllers Walk Into a Bar," about observability and runtime configuration for JoustMania, an open-source party game where players jostle motion controllers until someone falls over. Complex execution: multiple Bluetooth adapters, battery-powered devices, sensors firing at 100Hz. When a player complains their controller "felt different," how does anyone debug it at 2am at a convention?

A good and genuinely interesting problem and two people who knew the domain and cared about the project.

As we had different schedules, we started hacking on our own. We both made progress and moved fast.

And we never quite specced it together.

The knowledge gap opened quietly. Decisions got made that the other person didn't know about. Assumptions turned out not to be shared. Work that should have built on itself didn't quite fit together. Not because either of us was wrong, but because we hadn't stopped to build the shared picture before we started building the thing.

The fix wasn't better tools, a different framework, or a smarter approach to Bluetooth telemetry. What we'd skipped was simpler and harder than any of that.

The book is about the conversation we didn't have, not the Bluetooth controllers.

What happened between me and my university friend, two capable people with good intentions and different schedules who moved in parallel instead of together, is happening to software teams everywhere right now, at a scale and speed that makes the gap much harder to notice and much more expensive to close.

AI gave every engineer a tool that makes starting to hack immediately feel productive. The agent is ready. The prompt is right there. Why stop to spec? Why slow down for the conversation? The output is coming, it looks right, the ticket will close.

And somewhere in the gap between all that individual motion, the understanding stops being shared.

There's a structural reason AI has this problem.

Unless embedded in a validation loop, AI systems generate plausible output without grounded accountability. Modern agentic workflows can include tests, linters, static analysis, and human review, but those are additions to the system, not properties of the model itself. Strip them away and the model generates confidently regardless of whether the output is right. No peer review. No colleague who reads it and says "that's not how it works."

The same thing happens to engineers who go heads down with AI tools and stop talking to each other. The output keeps coming, the confidence stays high, and the errors accumulate quietly, until someone external catches them, or until nobody does.

AI hallucination and team misalignment aren't the same mechanism, but they rhyme: both produce confident output that nobody is checking against shared intent.

The industry's response to AI has been, mostly, to go faster.

Add the tool. Ship more. Reduce the headcount that feels redundant when the agent can implement. Optimize the individual. Measure the output.

That's a familiar move. Long before AI, teams arrived at this conclusion themselves, not because someone told them process was bad but because the process they had was tedious and wasn't bringing value. The planning meetings ran long and produced nothing. The reviews caught nothing. The deliberate conversations before anyone wrote a line of code felt like overhead with no return. So teams cut them. Rationally. And for a while it felt like freedom.

We learned, sometimes painfully, that this was wrong. Shared understanding was the foundation, and the conversation that felt like it was getting in the way was often the work.

AI is offering the same deal at a larger scale. Move faster. Skip the room. The tool is right there.

Extreme Programming understood what was at stake in the nineties.[1] Mob programming, pair programming, collective ownership: an entire practice built on the conviction that software development is a social activity. The understanding built through collaboration was the point.

The industry moved on. Faster frameworks. Individual metrics. Optimized handoffs. The team became a coordination cost to minimize.

The industry is rediscovering, under pressure, what XP already knew.

It begins with Sisyphus, and with me as the problem, pushing harder against a process that was already broken and shipping faster with AI without ever changing the terms of the curse.

It ends with the Phoenix, the choice to rise differently, not back to the old way but to a rediscovery of what was always true in a new form.

In between, it follows one thread, shared understanding, through its whole life. Where it belongs, what it actually is, and who builds it. What it's worth once implementation gets cheap. How a team creates it together, challenges it until it holds, and keeps it from dying when the people who held it move on. How you tell it's really there, what structure and culture keep it alive, and what it takes to scale it across an organization without thinning it to

[1] Kent Beck, *Extreme Programming Explained: Embrace Change* (Addison-Wesley, 1999).

nothing. Each chapter takes up one question about understanding, and together they follow it from where it starts to what you do with it.

The thesis: as implementation becomes abundant, shared understanding becomes the scarce resource, and constructive friction is the mechanism that protects and tests it. The organizations that keep that friction will outperform the ones that optimize it away.

This is a practitioner's book, not an academic argument. It doesn't claim to be the first to notice that teams matter or that shared understanding is valuable. Those ideas have lineage: Peter Naur's theory-building view,[2] XP, domain-driven design,[3] collective code ownership, and an academic literature of their own.[4] What's new is the specific problem: every current tool and framework for AI-assisted development solves alignment between one person and one agent. None of them address what happens when the spec needs to emerge from a room of people who don't yet share the same understanding. The spec: what to build and why, held in common. Not a document.[5] This book is the argument for that layer, the human and social conditions that make any spec-first approach work.

The conversation that feels slower than prompting alone is doing something the prompt cannot do.

The argument in this book didn't emerge in isolation. While writing it, I kept finding people who'd arrived at the same conclusions on their own: a CTO who'd built the same principles into a workflow before reading a word of this, ThoughtWorks researchers mapping the same territory from another angle, Matt Pocock building executable skills for specification. The pressure toward this model is real enough that people keep hitting it without coordinating.

I'll add one more signal: I started writing this book alone, with no one to

[2] Peter Naur, "Programming as Theory Building" (1985; reprinted in *Computing: A Human Activity*, ACM Press/Addison-Wesley, 1992). Naur argued that a program is the theory its builders hold, that the theory cannot be fully captured in documentation, and that a program whose builders have dispersed is effectively dead.

[3] Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley, 2003).

[4] Martin Glinz and Samuel Fricker, "On Shared Understanding in Software Engineering: An Essay," *Computer Science — Research and Development* 30, nos. 3–4 (2015): 363–376.

[5] Throughout this book, *spec* means this shared understanding. Where the formal sense is meant — a contract, like the OpenFeature specification — the full word is used.

tell me whether the idea was right. An argument for teams, begun without one, which is why the absence shapes its early chapters. The readers came later, once there was a draft to show.

If you feel like something is wrong but can't point to it, if the speed feels good and the output looks right but something is missing, this book was written for you.

It's written for engineers, but also for the people in the room with them. Product owners trying to understand why AI adoption hasn't made delivery more predictable. Team leads watching their teams drift apart. Stakeholders wondering why the spec keeps getting misunderstood. If you care about how software gets built together, not just how fast it gets built, you're in the right place.

You're not behind. You're seeing what the speed costs.

We started hacking. We forgot to talk to each other. We forgot to build the picture together.

This is the conversation we should have had first.

The Curse of Sisyphus

Sisyphus was condemned by the gods to roll a boulder up a hill for eternity, only to watch it roll back down each time he neared the top. Not punishment for failure, but punishment for believing he could outsmart the system.

If implementation is the cheap part now, where does the hard part go?

I was the problem.

That took me a while to see. I was shipping fast, quality was good, tests were there. Every review request I opened was a genuine improvement to the codebase. And I had multiple open at any given time, because I was filling the review wait with more implementation. While someone in Canada was sleeping, I was already three tasks deep: a new review request open, another under review, a third being planned.

The team was distributed, six hours between Europe and Canada, but that wasn't the issue. Distributed teams work. The issue was volume. AI let me produce faster than the team could absorb. Review requests piled up. I started stacking them to manage the backlog, which created more noise, not less. The system was designed to produce exactly that outcome, not a slow team, but a process that couldn't keep pace with the output.

I felt the burden. No one said anything directly, but you feel it in the silence. Review requests sitting. Comments sparse. The sense of generating more than the team can hold.

So I did what felt productive. I opened another review request.

The review bottleneck and the spec gap were the same problem wearing different clothes.

We had something close to the right process. A product manager and the more senior engineers would talk through what needed building. Decisions got made and intent got written into Confluence. Then it became tickets, and the team picked and went.

That sounds reasonable, but in practice, two things were broken simultaneously.

The scoping documents were written from a product manager's perspective: where we want to go, what the final state should look like, the vision six months out. Useful for a roadmap. Not useful when an engineer is trying to decide how a function should behave today. And when the docs did go technical, they'd sometimes arrive with a solution already decided by the seniors and the product manager, handed down to whoever picked the ticket. Or they'd leave the technical decisions entirely to the engineer, mid-implementation, without the context the seniors had built up in their informal conversations.

We had also stopped doing planning sessions. I'm not sure exactly when. It wasn't a decision. It just faded. The backlog existed, the docs existed, the tickets existed. Planning felt redundant when everything was already written down somewhere.

The problem was that the writing-down was never the point. The point was the shared understanding that used to get built in the room. When the room went away, the understanding went with it. The documents stayed. But documents can't answer questions mid-implementation. Documents can't notice when a technical decision made six weeks ago no longer fits what the system actually looks like.

So the spec was always incomplete in a specific way. Not wrong at the top; the vision was usually fine. Missing in the middle, where the engineers who hadn't been in the discussion between seniors and the product manager needed to make real decisions, and had nothing to go on except a ticket and a Confluence page written for a different audience.

And then I came along with an AI power tool and started shipping against that gap at three times the previous pace.

The review bottleneck wasn't really about reviews. It was about shared understanding. Nobody could review fast because nobody had been in the room when the thinking happened, and half the thinking had never been written down at all.

Working on JoustMania, the same Bluetooth party game behind the KubeCon talk, we wanted to add a feature that remembered how players had played before. Dynamic behavior that adapted over time. Interesting idea. We implemented it with AI assistance: storage, pipelines, game recording, replay. Each piece followed logically from the previous one. The agent was never wrong about any individual step.

The review request sat open for weeks. We never merged it.

The feature was pointless. JoustMania is a local party game. Controllers get passed around, sessions end, nobody registers. There's no way to know if the same person is holding the same controller from one game to the next. Player history doesn't attach to anything. We'd built a sophisticated solution to a problem that didn't exist.

Nobody asked that question before the first line of code. We asked it, implicitly, by never merging the review request.

The boulder made it all the way up. Then it just stayed there.

Addy Osmani, writing about loop engineering in June 2026, quoted Boris Cherny, head of Claude Code at Anthropic: "I don't prompt Claude anymore. I have loops running that prompt Claude and figuring out what to do. My job is to write loops." Osmani's observation: "Two people can build the exact same loop and get completely opposite results. One uses it to move faster on work they understand deeply. The other uses it to avoid understanding the work at all. The loop doesn't know the difference. You do."[6]

The loop amplifies whatever understanding the team brings to it. If the team didn't build shared understanding before the loop ran, the loop executes the misalignment faster. The boulder doesn't just go up faster; it goes up faster in the wrong direction.

Eliyahu Goldratt's Theory of Constraints[7] makes a simple claim: optimizing any part of a system that isn't the bottleneck doesn't improve throughput. It just moves work faster toward the constraint. If shared understanding is

[6] Addy Osmani, "Loop Engineering" (addyosmani.com/blog/loop-engineering/), quoting Boris Cherny of Anthropic.

[7] Eliyahu M. Goldratt, *The Goal* (North River Press, 1984); the Theory of Constraints.

the bottleneck, if the team's ability to agree on what to build is what limits delivery, then making implementation faster doesn't help. It produces more output against a constraint that hasn't moved.

The boulder goes up faster. The constraint was never the pushing.

Some teams will read this and disagree. They removed the planning sessions, went AI-first, doubled their output, and things are working fine. They might be right. For now. What "for now" means varies: sometimes it's the first time a senior engineer leaves and the system becomes inexplicable to the people who remain. Sometimes it's the first time a feature request surfaces a decision nobody remembered making. Sometimes it's the moment the codebase gets complex enough that the informal knowledge can't hold it together anymore. Speed without shared understanding creates debt that compounds invisibly.

Most teams I see have done the same thing I did. They've added AI to the implementation end of the process and called it transformation.

It's just acceleration, and acceleration without a better process gets to the wrong place faster.

The instinct is understandable. AI tools drop into an existing workflow cleanly. The engineer opens an editor, starts prompting, ships more code. The feedback loop is immediate and satisfying. The problems it creates are delayed and diffuse: a review queue that grows quietly, a spec that gets thinner as the pace increases, a team that starts to feel like an audience for one person's output.

Changing the process is a different kind of work. It requires the team to agree that something is broken before the pain is obvious. It requires investment in planning before anyone has written a line of code. It requires trusting that the time spent in a room together, before the agent runs, is the most valuable engineering time a team has, and that's a hard sell when the backlog is full and the AI tool is right there.

The agent loop is real. AI can take a well-specified ticket, implement it, run a review pass, flag what doesn't fit, and cycle back, without a human in the middle. That's not science fiction. It's available now, and it works when the input is good enough.

That loop has a left side and a right side. The right side is where the agent

runs: implementation, review, iteration. The left side is where the thinking happens: what to build, why, and what done looks like. This book is an argument for the left side. For the humans who need to be there while the work is still on the page, building the shared understanding that determines whether the loop produces the right thing.

The input is the problem.

OpenAI described this precisely after shipping roughly one million lines of production code with Codex agents: "from the agent's point of view, anything it cannot access in-context effectively does not exist."[8] Knowledge in Slack threads, in documents, in someone's head: invisible to the system. The agent builds from what it can see. What it can't see might as well not exist.

Garbage in, garbage out, just faster. If the prompt reflects a thin spec that a product manager wrote on a six-month horizon while everyone else was working on tickets, the output will be confidently, efficiently wrong.

The shift has to happen before a line is written. Collaboration moves left: into planning, into spec, into the moment where the team builds shared understanding of what they're actually trying to build.

If the team can be in the same room, or on the same call, that's the strongest version: synchronous, whole team, shared understanding built in real time. Everyone leaves with a spec and everyone in the room owns it.

Not every team has that option. When the timezone gap is six hours, synchronous starts to mean someone sacrificing their morning or their evening on a regular basis. For teams in that situation, the planning needs a different form factor. But the outcome has to be the same. A shared decision, made by everyone who needs to act on it, recorded in a way the agent can work from.

Think of it like a review request. Not on code, on the spec itself. The spec opens for review. Anyone can comment. Decisions get made in the thread, not in a side conversation between the product manager and two seniors. And there's a merge condition: a moment where the team agrees this is decided, this is what we build from, the agent can run.

That last part is what our Confluence documents were missing. They existed. They sat there. But there was no merge gate. No moment where the team

[8] OpenAI, "Harness Engineering: Leveraging Codex in an Agent-First World" (openai.com/index/harness-engineering/).

said this is decided. No record of why the decision was made, only what it was.

That matters more than it sounds. Six months later, when the system has changed and the decision no longer fits, the team needs to know whether to revisit it or discard it. The reasoning is what tells them. A document with no comment history and no approval record gives them the what with none of the why. A spec treated like a review request gives them both.

The agent working from a spec like that is in a fundamentally different position. It has context, not just requirements. It can flag when a new ticket contradicts a previous decision. It can surface the reasoning when something looks wrong.

The spec becomes a living record of intent, not a snapshot of what one person thought the product should be.

I'm not writing this from the outside. I was the power user creating the bottleneck. I was shipping against a spec I hadn't fully questioned, in a team that hadn't fully owned it, across a timezone gap that made everything slower and more isolated.

The process wasn't broken in an obvious way. It was broken in the quiet way that distributed async teams break: everyone doing their part, nobody in the same moment, understanding siloed in the people who wrote the documents rather than shared across the people doing the work.

What I needed wasn't a better AI tool. I needed the team to converge on intent before I started. In a room if possible. Async if not. Together, deliberately, with a record of what we decided and why.

The process was the problem. But the process isn't the only thing AI changes. It also changes what's left for the people inside it.

The End of a Craft?

"Craftsmanship names an enduring, basic human impulse, the desire to do a job well for its own sake." — Richard Sennett, *The Craftsman* (2008)[9]

If the agent does the building, what's left that only a person can do?

Everyone is asking this out loud now and most of the answers are either panic or a sales pitch.

If the agent handles implementation, what's left for the engineer? What still requires a person who has spent years getting good at this?

The honest answer is the work that can't be specced: the judgment before the spec exists and after the agent has run.[10] The agent is extraordinary at building what it's told to build. It has no opinion about whether that was the right thing to build, or whether the system it's being added to is becoming something coherent or something nobody can hold.

That judgment has a name once it's shared. Direction. And it's the part of the craft the agent doesn't touch.

Direction is easy to mistake for vision, which sounds like a slogan on a wall. It's more concrete than that. It's the running answer to "where does this need to go," held closely enough that the next decision can be checked against it. Whether today's choice still makes sense six months from now. Whether the system is becoming one thing or three.

[9] Richard Sennett, *The Craftsman* (Yale University Press, 2008).

[10] This is "Discernment" in the 4D Framework for AI Fluency — Delegation, Description, Discernment, Diligence — developed by Rick Dakan and Joseph Feller with Anthropic (aifluencyframework.org). The framework names the individual competencies for working with AI; this book is about the shared understanding a team builds before those competencies have anything reliable to act on.

The agent has none of it, and not because it isn't capable enough yet. It optimizes the request in front of it, which is what it's for. Asked whether this is a good implementation of what it was told, it will often answer better than a person could. It has nothing of its own to say, though, about whether the system should be heading there at all, because that question is about the whole history of the system and where it's meant to end up, not the task on the screen. That picture lives in people who have watched the system become what it is.

It's fair to push back that the agent isn't blind to that history anymore. Memory persists across sessions, and context files like AGENTS.md[11] let a team write the picture down and hand it over. That's real, but it moves the work rather than removing it. Someone still has to write that picture, keep it true, and hold it in common. A memory kept in one person's local setup is the old privatization in new clothes: the drift between two copies is now written down, and still invisible to anyone but them. The agent can store the direction. It can't decide it, notice when it's gone stale, or tell whether the copy it was handed is the one the team is steering by.

And more of the picture isn't automatically better. A context swollen with every past decision can drag the agent down the way a session left running too long does, because the bloat buries the signal, and only a person can tell which part is still load-bearing and which has turned to noise. A fresh, well-aimed start often beats a long one carrying everything it ever accumulated.

The judgment shows up before anything gets built. Which approach, and not the other. What scope is worth it. What "good enough" means for this team, this week. The agent builds what it's told. It doesn't decide whether the thing is worth building at all. Deciding that is direction work, and it's irreducibly human.

Over one Christmas I built a Rust evaluation engine that compiled to WebAssembly, a small program in a language I'd never written a line of. The same open-source specification had a separate implementation in five languages: Python, Java, .NET, TypeScript, and Rust, and each was quietly making slightly different decisions at the edges. One engine compiled to WebAssembly meant one set of decisions, every language inheriting the same behavior. The agent wrote every line. I couldn't have. What I brought

[11] AGENTS.md is an open format for agent instruction files (agents.md), adopted across a range of coding agents.

was the judgment around it: that the disagreement between the five was the actual problem to solve, that a single shared engine was the way to end it, and that a rough holiday prototype was enough to prove the idea. The agent built. I decided what was worth building, why, and how we'd know it worked.

That division is the whole shape of the work now. The agent on implementation. The person on direction.

It was always the more interesting half. The craft was never the typing. It was the judgment the typing crowded out. Implementation was just where the time went. Take the time away, and what's left is the part that took years to learn.

Direction has two failure modes, and the agent will flag neither.

The first is that nobody holds it at all. The work still gets done: tickets close, features ship, the agent never tires. But every decision is made against a local picture, and the local pictures slowly stop agreeing. The system becomes one thing in one corner and a different thing in another. Months later someone reaches a choice that no longer fits and goes looking for the reason it was made, and there isn't one, because the person who made it was answering a smaller question at the time. Nothing announces this. It's the most expensive kind of drift exactly because no single step ever looked wrong.

The second is subtler, because it looks like the problem is solved: one person holds the direction, and a direction held by one person is just an opinion. The moment someone builds against a different picture of where the system is going, the work diverges quietly, because nobody is wrong, they're just pointed differently. A Northstar only does its job when the team is steering by the same one. And a shared direction doesn't come from one person thinking clearly. It comes from a room.

That's the argument the rest of this book has to earn. For now it's enough to say the craft didn't end. One craft did, the implementation most engineers built their identity on. What replaced it is the judgment that implementation used to crowd out.

So the craft survives, moved up to direction, the high, slow plane, where the system is headed over months. But the agent erodes judgment closer to the

ground too: in what the tool quietly stops an engineer from noticing, one prompt at a time.

A Fool with a Tool is Still a Fool

"A fool with a tool is still a fool." — Grady Booch (attributed)[12]

If judgment is the craft that's left, what does it catch that the tool can't?

My first serious attempt with an AI coding tool was extracting an API package out of a large open source SDK. The goal was clean enough: separate the API from the implementation, two packages instead of one. Standard refactoring work, the kind of thing that should be mechanical once you know what you want.

I burned two days on it. Two days of reshuffling and redefining, no useful progress, the agent confidently helping me go in circles, producing output that looked plausible and went nowhere. I learned a lot. I produced nothing useful.

A week later I tried again. Same task, same tool. It worked.

The difference was that I now understood the system well enough to tell it what I actually needed. I'd failed my way into the right level of specification. I knew what to point out, what to constrain, what to leave open. The prompt was better because my thinking was better.

That's the thing that gets lost in the enthusiasm about AI coding tools. The tool doesn't make the engineer smarter. It makes the engineer faster, but only at the level of thinking they already have. If the engineer's understanding is shallow, it produces shallow output quickly. If the specification is vague, it fills the vagueness with confident guesses.

[12]Widely attributed to Grady Booch; the exact origin is unverified.

A fool with a tool is still a fool.

I got better over time. Prompts that would have taken me two days of burned tokens eventually took two hours. The tool started clicking once I understood what it needed to be told, what level of detail it required, where it would go wrong without guidance.

But here's what I noticed about that learning: it was entirely private.

I failed, observed, adjusted, retried. Nobody watched that process. Nobody else on the team went through it with me. I was building the understanding of how to specify, how to constrain, how to prompt for the kind of system we actually needed, and it lived in my head. It didn't transfer. It didn't accumulate anywhere the team could access it.

Which means everyone else who picked up the tool was starting from scratch. Their own two days of burned tokens. Their own private iteration loop. Learning the same lessons in isolation, making the same early mistakes, building the same understanding that never quite made it out of their head either.

AI's quietest effect on teams is the gradual privatization of operational knowledge. How to work with the agent well is hard-won understanding. And by default it goes nowhere.

I was getting faster. Tests were thorough. Implementation looked structurally clean. I had developed a review habit. Before handing anything to the team, I'd check my test cases, look for edge cases I might have missed, flag anything that seemed odd in what was being tested.

What I wasn't checking in the same way was the implementation's behavior. The agent wrote it, it passed the tests, it looked structurally sound. My attention had shifted, upward toward architecture and coverage, away from granular behavioral correctness at the line level.

There was a junior engineer on the team. First or second year. Exceptional eye for detail, the kind you don't develop, you either have or you don't. He started finding things. Regularly, in a pattern, nothing alarming, just consistent. Edge cases. Behavioral inconsistencies. Small gaps between what the code did and what the project specification said it should do.

Nothing catastrophic. But in a feature flag backend, behavioral correctness isn't optional. A flag that evaluates differently than the specification says it should is exactly the kind of bug that's invisible until it isn't.

He was catching things I'd stopped looking for. Not because he was better than me. Because he was still looking at the thing my attention had drifted away from. He read for behavior. I'd started reading for structure, because structure was what the agent handed back to me and what the tests confirmed.

The tool hadn't made me worse. It had reorganized where my expertise got applied. And the reorganization created a blind spot I couldn't see, because I was focused on the output the tool produced, not on what the output wasn't telling me.

That blind spot only became visible because someone else on the team was still looking at the full picture.

What "a fool with a tool is still a fool" actually means in practice isn't that AI makes engineers foolish. Most engineers using these tools are experienced and careful, and they develop real skill with the tool over time. The problem is subtler: the tool quietly shifts what the engineer pays attention to, and the things it shifts them away from don't announce their absence. They go unnoticed until someone else catches them, or until nobody does.

The team is the check on what the tool can't see.

It works as a different set of eyes at a different level of the system, not as a safety net. The junior engineer wasn't valuable because he caught my mistakes. He was valuable because he was still reading the code the way you read code before you've developed the habit of trusting the agent's output. He was closer to the behavior than I was.

But that check only works if the team is actually looking. And the same forces that push engineers toward private AI tools push them away from collective attention. Everyone heads down with their own context, their own loop, their own hard-won understanding of how to prompt well. The team starts to feel like a set of individuals who share a codebase rather than a group of people who share a system.

The agent only validates against the prompt. If a prompt reflects one

person's understanding of what the system should do, that's all it gives back. The team is the only mechanism that puts more than one person's understanding into the spec before the agent runs.

Prompting skill is the smaller part of what matters here. System thinking is the bigger one, and it's what gets built collectively: in the room, in the review, in the moment where someone with a different level of attention says that's not quite right, here's what the spec actually says.

A fair objection here: I just hadn't learned to use the tool yet. The two days of burned tokens were a skill problem, not a fundamental limitation of AI. And that's partly true. I did get better. The tool did get more useful as I understood what it needed. But that's exactly the point. The skill the tool requires didn't come with the tool, because it's the kind of thing that used to develop in the room, and when everyone learns to prompt privately, it develops alone or not at all.

Extreme Programming[13] made this argument in the nineties: understanding develops through collaboration, not in isolation. The industry mostly ignored it in favor of individual productivity metrics and frameworks that optimized the team away.

AI is making the XP argument unavoidable by forcing the team to do what the tool can't.

The junior was doing recognition work: catching the wrong shape before it compounds. It's a skill, and it doesn't only belong to juniors. An eval checks output against intent someone already encoded; recognition catches drift from intent nobody had encoded yet, before there's a test to write.

It's quiet, constant work. It surfaces in a code review, in exploring an unfamiliar part of the codebase, in the planning pass before a spec goes to the agent. An engineer reaching for a pattern the team abandoned two years ago. A hand-rolled utility that duplicates what the shared library already provides. None of it is catastrophic. All of it gets more expensive the longer it stays. And none of it is something the agent catches reliably, because it doesn't know what clean looks like in this particular system, accumulated over time. The engineer does, from having been in the system, from having seen the messy version enough times to recognize it on sight.

[13] Kent Beck, *Extreme Programming Explained: Embrace Change* — discussed, with citation, in the introduction.

The eye develops through exposure. It can't be specced, and an agent can't be prompted to have it.

What I do with the recognition matters as much as the recognition. Reviewing agent-generated code on the JoustMania codebase, I noticed it had built three separate structures to track one piece of state where a single one would have been clean. It worked. It passed the tests. And it meant every future change would touch three places instead of one.

I didn't fix it. I created a ticket.

Not because the fix was hard. Because fixing it in place would have conflated two things, the work I was doing and the problem I'd noticed, and muddied both. Noticing without immediately fixing, naming a problem without solving it in the wrong moment, is its own discipline. AI makes it harder to practice, because the agent wants to keep going and the flow state feels productive. The pause to write the ticket instead of fixing in place is a small act of system thinking over immediate output.

The Product Owner is Dead. Long Live Product Thinking.

"The king is dead. Long live the king." — traditional proclamation of royal succession

If the team can catch the wrong shape, who catches the wrong ask?

Open source maintainers are drowning.

The pattern is the same everywhere. Someone opens an issue: a bug report, a feature request, a question about behavior. A contributor picks it up, feeds it to an agent, and a pull request arrives. Thorough. Tested. Sometimes even clean. And wrong at the level of need: the issue wanted a quick way to verify a concept, but what arrived is sophisticated, extensible, and production-ready. The maintainer now has to review the generated work, explain why it's the wrong level of solution, and do it without discouraging the contributor from trying again.

Multiply that by twenty open pull requests.

Open source is just a distributed async team with an unstable roster and no onboarding. It always has been. Contributors come and go. The maintainers are the only constant. There's no planning, no shared context that builds over time, no Product Owner (PO) holding the thread between handoffs.

I know this from the inside. I maintain OpenFeature, a feature flagging specification where behavioral correctness matters. A contributor once opened a large pull request, too large to review properly, so I asked them to split it into two for a better overview and an easier follow.

They came back with two pull requests, split by module. I'd meant by feature. The agent had also duplicated the shared logic between them instead of keeping it in one place. Technically compliant with my request. Structurally worse than what we started with.

They hadn't been careless. The agent hadn't malfunctioned. The instruction, split into two pull requests, was given without the context that would have made it meaningful. The agent filled the gap with the most obvious interpretation of my words, not my intent. What was missing was minimum viable context: enough to rule out the obvious wrong reading. The agent did what it was told.

What followed was frustrating for both of us. A lot of back and forth. Tension that shouldn't have been necessary between two people who both wanted the right outcome. We got there eventually: two pull requests, one based on the other, a stacked diff that worked. But the path to it cost time, energy, and goodwill that a shared understanding of the problem upfront would have saved entirely.

Nobody validated the need, or the approach, before the agent ran.

What did I change after this? Honestly, not much, and that's worth saying directly. CONTRIBUTING.md gets ignored. Issue templates get skipped. In open source, a maintainer often can't have that conversation before a contribution arrives. The async distributed nature of it means the shared understanding has to be front-loaded somewhere else.

The best partial mitigation I've found is writing issues with more context: minimum viable context in the issue description itself, so the contributor, human or agent, has less to guess at. Not the full spec. Just enough to make the obvious wrong interpretation less obvious.

It doesn't prevent the problem. It makes it cheaper. And in open source, that's often the best available answer.

It's an old problem running faster.

In product teams the same chain exists, though the Curse of Sisyphus chapter's product manager and this chapter's PO are different roles that land in the same seat, not one person renamed. A stakeholder has a need. A PO interprets it. The team builds the interpretation. Somewhere in that chain the actual need gets distorted, not through malice, just through the

normal compression that happens at every handoff. By the time it surfaces, something is already built.

The cost is rework. Delayed delivery. A PO who has to go back to a stakeholder and explain why what was built isn't what they meant. That conversation is never easy. And it happens again.

Product Owner, product manager, "Product": the titles vary and the industry has never agreed on the boundary. This chapter is about the seat, not the title: whoever sits between the stakeholders and the team, carrying both the product thinking and the organizational interface.

The PO exists, in part, to manage that chain. To be the translation layer between what stakeholders want and what teams build. To carry the context that doesn't fit in a ticket. To field the questions that surface mid-implementation when something doesn't quite add up.

It's a structural role for a structural problem and the role works until it becomes the structure's failure point.

The role doesn't have an off switch. Everything that can't be handled elsewhere routes through the same person. Stakeholder pressure from above. Requirement gaps from below. Product decisions that have to be made before anyone knows enough to make them confidently. POs I've known carried it well, until the structure broke them. Not because they were weak. Because it was set up to eventually break anyone.

Implementation is cheap, discovering it was wrong is expensive.

In the open source example, the maintainer pays that cost. In a product team, the PO pays it first, then the team pays it in rework, then the stakeholder pays it in delayed delivery. The misunderstanding moves through the system and someone absorbs it at every stage.

Collapsing the implementation phase surfaces misunderstandings faster. The agent runs, the output exists, the gap between what was needed and what was built becomes visible sooner. That's good if the spec was right. It's expensive if it wasn't.

A spec written before implementation starts and reviewed by the stakeholder catches the misunderstanding at the cheapest possible moment. The stakeholder reads it and says "that's not what I meant." Nothing has

been built yet. The clarification costs a comment in a document, not two weeks of rework.

I do a lighter version of this without a spec at all. When a PO brings a ticket, I ask what the motivation behind it is: the need the ticket is standing in for. Often that need is smaller than the ticket. It could have been solved with something simpler, but the ticket has already been engineered into a full solution before anyone checked whether the need required one. The same wrong level as the open source contribution: a thorough answer to a question nobody validated. The question catches it one ticket at a time, before the build.

In the open source world that's a fast experiment before a sophisticated solution. In a product team it's a spec review before implementation starts. The form is different. The principle is the same: validate the need before the thing gets built.

The traditional PO carries two things simultaneously. Product thinking: challenging intent, representing user needs, asking why before the team builds what. And stakeholder management: navigating organizational pressure, translating business context into something a team can act on, carrying the political weight to push back when the request is wrong.

Those are genuinely different skills bundled into one role partly by necessity. Building the spec together changes the necessity.

When the team builds the spec together, when product thinking happens collectively, that first part starts to distribute. Senior engineers develop the habit of asking why before how. Tech leads challenge intent as part of the planning process, not as an afterthought. Product thinking becomes a broader team competency over time, rather than something concentrated in a single role.

That doesn't happen immediately. Some teams need a dedicated PO precisely because that muscle isn't there yet. It becomes a development goal, not a permanent structure.

The second part doesn't distribute. Stakeholder navigation requires years of context that most engineers don't have and don't want to develop: the organizational interface, the political skill of managing what comes into the team from outside, the credibility to push back on an executive request. It's what survives as a dedicated role.

I did a version of this as a team captain. Requests came down from above, strange ones, impractical ones, and I made the call on them: some we didn't pursue, some we implemented differently than asked. As long as we honored the defined interface, the implementation was ours. That judgment, deciding what to absorb, what to push back on, what to reshape, is the part that doesn't distribute. It's not product thinking, and it's not something a team picks up in a planning session.

Something more like a Stakeholder Navigator, not a Product Owner in the agile sense. The person who gets the spec in front of the right people at the right moment, manages the review cycle, and protects the team from the noise while they do the thinking.

I know how that sounds. A rebranded PO. Partly yes. The difference is ownership. The old model has the PO owning product thinking and managing stakeholders. This one has the team owning product thinking and the navigator owning the organizational interface. That's a real split even if the job title looks similar from the outside.

And it's a more sustainable job, more defined, with an off switch.

The open source maintainer and the burned-out PO are dealing with the same structural problem from different angles.

The spec review is the mechanism that closes the gap between what's asked and what's needed. A defined moment where the need gets validated before anyone builds anything: a checkpoint, not a planning ceremony. Where the stakeholder can say "that's not what I meant" when it's still cheap to hear it.

Birgitta Böckeler[14], analyzing the harness engineering practices that surround agent systems, names the limitation precisely: the harness can govern how code gets written without ever checking that it does what users actually need. Technical correctness and functional intent are different problems. The harness solves the first. The spec review solves the second.

The role evolution is a consequence of that. When the spec carries the shared understanding, when the team built it together and the stakeholder signed off on it, the translation layer the PO was providing becomes less necessary.

[14]Birgitta Böckeler, "Harness Engineering," in the "Exploring Gen AI" series (martinfowler.com). "Harness engineering" is a shared term — OpenAI uses it too (the Curse of Sisyphus chapter); the two arrived at it independently.

What remains is the person who managed the organizational interface well enough to get the spec in front of the right people in the first place.

That person still matters, maybe more than before, and they just don't have to carry everything anymore.

Roles shift and responsibilities redistribute: the product thinking spreads into the team, and the organizational interface narrows to the one role that protects the room while the team does the thinking. All of it assumes the room exists. A place where the thinking actually happens, and a team that knows how to use it. That's the thing nobody has built yet.

The Agora

The agora was the heart of the ancient Greek city, a public space where citizens gathered not just to trade, but to think, argue, and decide together. It was where the polis became more than a collection of individuals.

If no one person owns the understanding, how does a team build it together?

Every team has a place where the real thinking happens.

Sometimes it's a formal meeting. More often it's a corridor conversation, a Slack thread that ran long, a one-on-one where someone finally said the thing they'd been sitting on for a week. The technical lead and the product manager talking through a problem before it becomes a ticket. The senior engineer pulling a junior aside to explain why the approach won't work. The offhand comment in a code review that reframes the whole feature.

That thinking is the most valuable work the team does, and even in well-documented teams, part of it stays invisible, dependent on the right people being in the right conversation.

The Spec Session is what happens when a team makes that space intentional. It's not a new idea: XP called it the planning game.[15] What's different is the stakes: where XP had an individual coder, the Spec Session has an agent. And the quality of the shared thinking before it runs determines everything that follows.

The individual coder was doing a second job nobody ever named. A human who doesn't understand what they're building gets uncomfortable, gets stuck, walks over to a desk and asks whether we meant this or that. That discomfort was a safety net no process designed: it caught planning

[15] Kent Beck, *Extreme Programming Explained: Embrace Change* — discussed, with citation, in the introduction; "the planning game" is one of its practices.

failures in week two instead of production. An agent has no such habit. It runs with its best interpretation or it stops, and neither looks like walking over to a desk. It can ask a clarifying question, but it asks the person who wrote the prompt, from inside that framing. The old process could afford an ambiguous spec because the implementer and the circuit breaker were the same person. The agent split them apart, and the Spec Session is what replaces the walk to the desk: the same catch, moved to the cheapest moment.

Before the Spec Session starts, one thing needs to exist: a reason to build anything at all. The business need and the user problem, as well as the organizational imperative. These come first, and none of them is more important than the others. The Spec Session aligns the team on what exactly gets built and how, not whether it's worth building. Without a clear need upstream, the best-run Spec Session produces a beautifully shared understanding of the wrong thing.

This is a dedicated space and not a meeting: meetings have agendas, action items, owners. A Spec Session produces something different: a room full of people who hold the same picture of what's being built, why, and what done actually looks like. The document is just the record of that convergence. It doesn't need everyone to read it the same way, only to keep pointing at the same thing.[16]

That's why the path is the goal. A team could be handed a perfectly written spec and skip the session entirely. They'd own none of it, having inherited someone else's thinking rather than built their own. And when the agent runs and something unexpected surfaces, the team that built the understanding together knows what to do. The team that received the spec has to go find the person who wrote it.

I run climbing courses for children. At the start of every session we sit in a circle: everyone speaks, everyone listens, no rushing past. We reflect on the last time. We let the room breathe. We create what in German has a better

[16]The sociological term is a boundary object — an artifact "plastic enough to adapt to local needs but robust enough to maintain a common identity" across groups (Susan Leigh Star and James Griesemer, "Institutional Ecology, 'Translations' and Boundary Objects," *Social Studies of Science*, 1989).

word than English offers: Bewusstsein. A shared awareness. A common consciousness of where everyone is before anyone starts climbing.[17]

What happens after that circle isn't engineered. The peer learning emerges. The child who felt heard speaks up on the wall. The one who listened offers help without being asked. The group becomes something that supports itself.

When we skip the circle, whether because there's no time, the energy is off, or someone rushes past it, the session feels disconnected. Just individuals climbing in parallel rather than a group climbing together. The difference is subtle, real, and almost impossible to point to for anyone who hasn't felt it.

That's the Agora, the condition that makes a room more than a collection of individuals.

Software teams need the same thing, but as a disposition the team carries continuously and not as a ceremony. The habit of making thinking visible before acting on it. Of listening before building and of checking where everyone actually is before assuming the team already knows.

The climbing circle is one way to make that instinct visible, but it does not have to be. Some teams do it in how they write issues or in how they start planning. Some do it in the way a senior pauses before running the agent to ask whether the spec is actually clear. The form doesn't matter, but the instinct does.

The Spec Session is one way to create that condition. The async spec review is another. What matters is the moment of shared Bewusstsein before the agent runs.

The Spec Session is not a planning gate or waterfall with better tooling. The team still iterates. Still puts things in front of users. Still discovers that what got built isn't quite what was needed. The difference is that the team iterates with everyone holding the same picture, so when the direction turns out to be wrong, everyone understands why, and the correction is faster. In truly complex domains, where cause and effect only become visible in retrospect,

[17] Developmental psychology has a name for this: shared intentionality, the capacity for joint attention and joint goals that Michael Tomasello identifies as the distinctively human cognitive trick, present in infants before language (*Why We Cooperate*, MIT Press, 2009). Joint commitments carry norms: Gräfenhain, Behne, Carpenter, and Tomasello showed that even three-year-olds take leave before abandoning a joint activity rather than walking away silently ("Young Children's Understanding of Joint Commitments," *Developmental Psychology* 45, 2009).

the Spec Session doesn't replace iteration; it makes iteration less expensive by ensuring the team is wrong together rather than separately.

What that condition makes possible is more than shared understanding. It makes trust possible.

In climbing, trust has a physical form. Belaying, holding the rope that keeps another person safe, is the most literal expression of it. One person's life, in a meaningful sense, in another's hands. In my children's courses, we build to that point deliberately. The circle first. The shared awareness. The environment where everyone feels safe enough to be uncertain out loud.

And then something that still surprises me happens. Five year olds belay each other. Supervised, of course, but the trust is real, the technique is real, and the care they take with each other is genuine. I have watched five year olds belay more attentively than adults I've seen on real rock, because the environment built the trust before the task required it. That's the standard software teams are measured against.

How many engineering teams have the trust required to say "I don't understand this spec" in a planning meeting? To tell a senior their approach is wrong? To admit mid-implementation that something doesn't make sense and risk looking slow?

That trust is built deliberately, the same way the circle builds it before anyone climbs.

The Agora is where that trust gets built. Everything that depends on it is downstream of what happens in the circle: the Spec Session, the junior who raises the edge case, the senior who shares uncertain thinking out loud.

The hardest part is building this culture, regardless of the format.

A Spec Session breaks when people don't say what they actually think: the junior who spots a problem but doesn't feel safe raising it, the engineer who thinks the approach is over-engineered but defers to the senior, the concern that never gets voiced because the culture doesn't make it safe.

When those things stay unsaid, they surface later. In the implementation, when the edge case appears and nobody knows how to handle it. In the review, when the senior realizes the approach was wrong and the rework is expensive. In the retrospective, when someone admits they knew but didn't say.

Psychological safety[18] is the prerequisite for a Spec Session. Without it, the team has a meeting where people perform agreement and the real thinking still happens in corridors afterward.

As a team lead, creating that space was always my goal. Not through a named process, but through behavior. Asking the question that made it safe to disagree. Thanking the person who raised the uncomfortable concern. Making it visible that the junior's edge case was the thing that saved two weeks of rework. Modeling the culture rather than mandating it.

The Spec Session makes that culture structural rather than personal. It says: this is the time, this is the room, this is the moment where it's not just safe but expected to say what you actually think. The format protects the behavior that good leads were trying to create informally.

The Spec Session produces minimum viable context: the agent runs without guessing, the team reviews without re-learning, the stakeholder recognizes what they asked for when they see it.

Acceptance criteria are a contract, not an understanding.

The session needs to surface three things before it closes.

What are we actually building. The specific behavior, the boundary conditions, the thing that would make a skeptic say "yes, that's done." The team needs to be able to describe it without looking at the document.

Why this approach and not another. Not for the audit trail, but for the moment six months from now when the system has changed and someone needs to know whether to revisit this or discard it. The why is what survives the what becoming obsolete.

What we're not building. The explicit scope boundary. The thing that's out of scope this iteration, that will be a different ticket, that the agent should not attempt. Without this, the agent fills the silence with reasonable guesses, wrong half the time.

When those three things are in the room and have been shared, challenged, and confirmed, the session is done.

[18] Amy Edmondson, "Psychological Safety and Learning Behavior in Work Teams," *Administrative Science Quarterly* (1999); later popularized via Google's Project Aristotle research.

The spec that comes out is not a user story wearing a new name. A story was a placeholder for a conversation that would happen during implementation, because the implementer could ask; the spec is the record of a conversation that already happened, because the implementer can't.

The difference fits on half a page. The ask, as I typed it that day: this pull request is too large to review properly, split it into two for a better overview and an easier follow. What a session would have recorded instead: two pull requests, split by feature, not by module, each reviewable on its own. The shared logic stays in one place; neither duplicates it. Why: a better overview, and a follow that doesn't require holding two diffs in your head at once. Not doing: no restructuring of the existing module layout, no new abstractions for the shared code. Done when each pull request reviews cleanly by itself and the shared logic exists exactly once.

That's one task's record, not a form. Yours will look different, because your gaps are different.

Consensus is too high a bar and too slow to reach. The Spec Session produces consent, the decision-making distinction sociocracy has used for decades[19]. Nobody objects strongly enough to block, and that's enough. The team can move with a shared picture even when individuals would have made different choices. The document captures them. The team owns them. The agent can run.

Async Spec Planning is the same thing with a different form factor.

For teams that can't be in the same room, distributed, different timezones, permanently async, the session becomes a review request on the spec. The draft opens. Comments come in. The what, the why, and the not-building get challenged asynchronously. The merge condition is when the team has converged, meaning the concerns have been addressed and the understanding is shared.

It's slower than a synchronous session. It's faster than the alternative, which is a document that gets written, inherited, and silently misunderstood.

But async spec planning carries a risk synchronous sessions don't. What stops one developer from using an agent to write the spec proposal, and another from using an agent to review it? If both sides of the conversation

[19] "Consent, not consensus" is a core distinction in sociocracy, formalized in Sociocracy 3.0 (sociocracy30.org).

are AI-mediated, the human understanding loop is bypassed before the code loop even starts. The spec merges. The understanding was never built.

The mitigation is minimum viable context, but applied to the spec proposal itself. A proposal rich enough that a reviewer cannot respond meaningfully without actually thinking about it. One that surfaces the assumptions, names the edge cases, explains the reasoning behind the decisions, demanding engagement rather than permitting rubber-stamping.

The same culture requirement applies. The async version needs people to actually comment, actually raise the concern, actually say "I don't think this covers the edge case where X." If the review request gets approved because nobody wanted to slow things down, or because an AI generated a plausible-sounding approval, the team hasn't had a Spec Session. The result is a document that collected silent agreement.

The test for whether async spec planning worked is whether the team could explain the decisions without looking at the document, not whether the review request merged. If the understanding lives only in the artifact, the session failed.[20]

Spec Sessions can become the new endless grooming when the team circles the problem, surfaces every concern, explores every edge case, and never converges. Three hours later there's a rich document and no decision.

The antidote is a rotating session lead: someone whose job that day is to get to convergence, to close the discussion when the understanding is good enough, to make the call when the team is going in circles, to protect the session from becoming the thing it was designed to replace.

Good enough for the agent to run without guessing on the things that matter, not perfect, not exhaustive, just sufficient.

The session lead owns the session, not the product; their job is to get the team to the point where the spec can close and the work can start.

---

[20] Chad Fowler's Deletion Test poses the same question at the artifact layer — imagine deleting the implementation, then ask whether the knowledge survives or the code was the only place it lived. Same structure, applied to code instead of a spec document. Chad Fowler, "The Deletion Test," *Phoenix Architectures* (aicoding.leaflet.pub).

Most teams don't have this space. The thinking happens, but it happens in fragments.

The Spec Session is the replacement for the fragments, not a ceremony to add to an already full calendar; it's the one place where the thinking that was happening anyway gets done together, gets documented, gets owned by the whole team.

The room has to be made, and protected; everything the agent produces is downstream of what happens in it.

If you want to try this, start small. Smaller, even, than the team this book will argue for.

Grab a coworker before your next task. Define the plan together. Define the prompt together. Then hand it to the agent and let it run against what you agreed. If you want to change something mid-run, stop: that's another spec, not an edit to this one.

Let the output tell you whether the spec was good enough.

If it worked, iterate from there. You don't need to change your whole process from one day to the next.

If it didn't, ask why together, not each of you privately at your own desk. Where did the agent take the wrong turn? Would a better explanation have prevented it? Did it fill a gap you didn't know you'd left? Did you hand it so much context that the signal drowned? There are many ways a run can fail, and you and your coworker are the only ones who can tell which one you hit. A failed run is the cheapest teaching material you'll ever get, but only if the lesson lands in both heads. A lesson learned alone is the problem this book is about, in miniature.

Figure out what works, and build on it. One experiment at a time, one task at a time. That's the thing engineers are good at anyway: fixing broken things.

Tools will make this easier. They already are: better async collaboration, lighter spec formats, smarter review workflows. The question isn't whether to use them. It's whether the conversation happens before or after you do.

Tools are beginning to encode this instinct.[21] Birgitta Böckeler[22], a Distinguished Engineer at ThoughtWorks, has analyzed the leading tools and identified the central gap: all of them assume a single developer does the requirements analysis. None address what happens when multiple people need to arrive at the same picture together. That's the Agora, the human condition that makes any spec-first approach work. A tool can encode the instinct, but it can't replace the room. A CONTEXT.md nobody owns is just a Confluence page.

ThoughtWorks, in their Technology Radar[23], flag a real risk: spec-driven workflows become "lengthy" and "elaborate and opinionated" when the spec becomes the goal. The Spec Session is the antidote, not an instance of it. The document is proof the conversation happened, a record of reasoning.

Kief Morris[24] introduces the closest external framing to this book's argument: humans "on the loop," building and maintaining the harness. The difference is precise: Morris solves where humans sit in the process. This book solves what they need to do before it starts. Böckeler's harness engineering[25] completes the picture: it enforces how code is written but, as she notes, "does not validate that it does what users actually need." The Spec Session fills that gap, ensuring the team holds a common picture of what users need before the agent writes a line.

The Spec Session is where the work happens. Which leaves one question worth sitting with: why does it have to be people in the room at all?

[21] Further reading on spec-first tooling: GitHub's Spec-Kit (github.com/github/spec-kit), the BMAD-Method (github.com/bmad-code-org/BMAD-METHOD), and Matt Pocock's AI Hero (aihero.dev) — including the `/grill-with-docs` skill, which makes the Spec Session executable. For the concept itself, see "Spec-Driven Development: From Code to Contract in the Age of AI Coding Assistants" (arxiv.org/abs/2602.00180, 2026).

[22] Birgitta Böckeler, "Understanding Spec-Driven Development: Kiro, Spec-Kit, and Tessl," "Exploring Gen AI" (martinfowler.com/articles/exploring-gen-ai/sdd-3-tools.html). She analyzed Kiro, spec-kit, and Tessl.

[23] ThoughtWorks Technology Radar, Volume 33 (2025), on spec-driven development (thoughtworks.com/radar).

[24] Kief Morris, "Humans and Agents in Software Engineering Loops," "Exploring Gen AI" (martinfowler.com).

[25] Birgitta Böckeler, "Harness Engineering" — discussed, with citation, in the Product Owner is Dead chapter.

The Ever-Agreeing Genie

Kent Beck calls his AI pair a genie: it grants wishes without judgment, exactly what you asked for, whether or not it is what you needed.[26] *The danger was never the genie. It was the wish.*

If a team builds understanding together, what keeps it honest?

I'm building a demo for my next KubeCon talk on JoustMania again: OpenTelemetry and OpenFeature wired together into what I'm calling an agentic nervous system, a game that runs its own experiments over time and tracks what happened.

I got it working and watched it run.

And right behind the working demo came the doubt. Is this worth building, or does a framework already exist that does this better? Observability data is what the system already tells you about itself while it runs, so is it actually a sound foundation for the agent's experiments, or a convenient one I've talked myself into? I don't know which. The talk isn't even accepted yet. Nobody has seen it.

I research the answers with AI, and I notice what that research doesn't give me. It's fluent inside the architecture I hand it: ask whether the design is sound and it reasons about the design I described. Ask whether something already solves this and it searches for the thing I named, not the thing I didn't know to look for.

So I have a system that works, and a talk that might never get a stage. Either way, I still don't know if the shape of it is right.

[26] Kent Beck named the metaphor first. See "Genie Wants to Leap," *Tidy First?* (tidyfirst.substack.com/p/genie-wants-to-leap).

There are four things you don't get from an AI, four things only another person can provide.

The first is weighted validation. When someone with their own hard-won experience, their own perspective, their own reasons to disagree, looks at what you're building and says "yes, that's right," it changes something, because they're independent of your framing. Their agreement costs them something: they considered it and chose to agree, not that they were designed to be helpful.

Someone who has built observability pipelines telling me the foundation holds would settle something the model's agreement can't.

AI agreement is cheap. Human agreement from the right person is not.

The second is scope judgment. Am I reaching too high or not high enough? Is an agentic nervous system too ambitious for a conference demo, or not ambitious enough to deserve a stage? I can't see that from inside it. An AI will help a person execute whatever direction they've chosen; it won't tell them the direction is wrong. That takes someone who has seen both failure modes: the plan that collapsed under its own weight and the one that never got anywhere, and can recognize, from the outside, which one the work is heading toward.

The third is the unknown unknown. The tool no one in the room is aware of. The approach that would make everything simpler but simply hasn't been encountered. If a framework already does what I built over the demo, the model and I are both searching for a name I don't know.

AI can surface unknown unknowns within the space it has been pointed at. Ask it "what am I missing here" and it will often find something useful. The harder gap is context-specific: the thing someone with direct experience of the situation would see that neither you nor the model would think to name. That requires someone looking in from outside the framing.

The fourth is teaching. This one has no JoustMania version. There is a difference between a thing being explained and a thing being taught. AI explains well. It can walk a person through a concept, surface the relevant literature, answer follow-up questions patiently. But teaching requires someone who recognizes when the learner doesn't know something yet and can hold them through that state until the understanding arrives. The

scar tissue matters, and so does the presence of someone who notices they're there and knows what to do about it.

Research is a substitute for teaching, slower and lonelier.

The Spec Session is where all four of these happen, before the agent runs.

The distributed async team loses these things first. The timezone gap, the Kanban board, the scoping documents that arrive as finished artifacts, all of it slowly drains the moments where weighted validation happens, where someone challenges the scope, where an unknown unknown gets surfaced by someone who happened to know the thing no one else did.

And AI accelerates the drain through usefulness. The tool is faster, always available, never in a meeting, so the maker stops reaching for the person and starts reaching for the tool.

Until you have a working demo and no stage to test it against, and the question of whether the shape is right sits with you in a way it wouldn't if someone else had checked the architecture first.

Those individual relationships matter, but they're not the structural answer. The structural answer is the team.

This isn't only the solo builder's problem. The same gap opened on a team a few chapters back, where the junior catching what the agent and I had both stopped seeing was supplying exactly this, a check against reality that no prompt produced. Take that junior out of the room, let everyone review alone against their own prompt, and the agreement gets cheap again.

The team is the environment where independent judgment can happen, not a delivery mechanism but the place where judgment gets tested. Where someone pushes back from a starting point that isn't yours, the one place an agent, working from your framing, can never stand.

AI is frictionless by design. It finds a way to make an instinct seem reasonable. It smooths the edges off the direction already chosen. That's useful for execution and dangerous for judgment.

The team has friction by nature. Different experience, different exposure, different mental models of where the system is going and why. That friction

is the mechanism. The reason the team's judgment is different from the AI's agreement is precisely that the team doesn't have to agree.

A form of friction can be engineered from an AI: build the context to argue back, load it with counterarguments, instruct it to push back. But the pushback is still downstream of the maker's framing. The AI can challenge stated assumptions. The unstated ones are invisible to it.[27][28]

None of the four is a capability gap. Capability gaps close; that's what model releases do. These four are consequences of the one fact that doesn't: however capable the genie gets, it never gets to want. That's why the human doesn't move out of the loop as the model improves. We were never in it because we were better at something. We're in it because the loop is for us.

Understanding is only useful if it can be challenged. And the place where it gets challenged, by someone who doesn't have to agree, is the team.

The friction is the point.

Weighted validation, scope judgment, the unknown unknown, teaching.

Three of the four keep your judgment honest while you build. Teaching does something the other three don't. It is how the judgment survives the person who has it, and it is the first thing the agent takes. So where does the next person learn to be the one who doesn't have to agree, when the tool is always the faster answer?

[27] Alibaba's Qwen team makes the same point formally in "The Verification Horizon: No Silver Bullet for Coding Agent Rewards" (2026, arxiv.org/abs/2606.26300): every verifier is only a proxy for human intent, never the intent itself.

[28] Anthropic's own harness engineers report the same from the inside: agents grading their own work reliably skew positive, and making a generator critical of its own output proved far less tractable than tuning a separate, skeptical evaluator — whose judgment, in turn, had to be calibrated against a human's. Notably, their multi-agent harness also converged on a spec-first ritual: generator and evaluator negotiating what "done" looks like before any code is written. Prithvi Rajasekaran, "Harness Design for Long-Running Application Development" (anthropic.com/engineering, March 2026).

The Alexandria Problem

The Library of Alexandria was the ancient world's greatest repository of knowledge, and its greatest single point of failure. It didn't burn in a single night. It declined across decades, through neglect, through the slow erosion of what it had been. By the time it was gone, almost no one had noticed it going.

If the understanding holds today, how does it survive the people who leave?

I learned to be an engineer in code reviews.

From watching seniors think out loud, never from documentation or courses: how they searched for the problem beneath the problem, how they argued about tradeoffs, how they asked questions that reframed everything before anyone had looked at a single line of code.[29]

The most important thing I learned was how to ask the right question. How to formulate a problem so clearly that the solution could emerge from it, not knowing the answer, but knowing how to construct the context that makes the answer findable.

I learned that most deeply in open source. Suddenly the person reading the issue knows nothing. They weren't in the meeting. They don't know the system, the constraints, the previous decisions. I have to build the entire picture from scratch: structure the problem so a stranger can understand it, provide every piece of context someone outside my head would need to engage with it meaningfully.

That's a harder discipline than explaining something to a colleague who

[29] This is Peter Naur's claim from "Programming as Theory Building" — cited, in full, in the introduction. Naur held that a new programmer can only acquire the theory of a program through active engagement with the people who already hold it, and that the theory itself cannot be written down. He drew the conclusion at maximum strength: a program whose team has left should be discarded and rewritten, because the theory died with the room.

shares the same background. A colleague fills in the gaps from shared experience. A stranger doesn't. Open source teaches the contributor to write for the stranger. To assume nothing. To make the problem legible to someone with no obligation to figure out what they meant.

But there's a second discipline open source teaches that's equally important. Not over-sharing.

As little as possible, as much as needed. Too little context and the stranger can't engage. Too much and the contributor has made their problem the stranger's problem. They have to sort through noise to find what actually matters. The discipline is knowing what to leave out: what's assumed, what's load-bearing, what would change the answer if it were different.

It's the same discipline as writing a good prompt. Not the longest prompt, the tightest one. Too much context and the agent loses focus, chases the noise, produces something that addresses everything it was told and misses what was meant. Too little and it fills the gaps with confident guesses. The discipline runs past the single prompt. A session that has run too long carries everything it ever touched, accumulating the same noise an overlong prompt does, and knowing when to clear it and start fresh, scoped to the real problem, is part of the same skill.

There's a failure mode minimum viable context is designed to prevent. The XY problem.[30] Someone has problem X, thinks the solution is Y, asks for help with Y, and the person helping them never finds out about X. The answer to Y lands, it doesn't solve anything, and everyone wasted time.

It's endemic in open source. Someone asks a very specific question about a very specific implementation detail, and three comments in a maintainer asks "what are you actually trying to do," and the real problem surfaces. Which has a completely different, usually simpler solution.

It's equally endemic in AI prompting. The engineer prompts for Y, the agent implements Y confidently, and X remains unsolved. The agent doesn't ask what they're actually trying to do. It assumes Y is the right question because they asked it.

Minimum viable context means surfacing X before asking about Y. It means providing the right context, not just enough of it. The actual problem, not the assumed solution.

[30] Coined by Mark Jason Dominus in a 2001 comp.lang.perl.misc post; popularized since via xyproblem.info.

That's what the senior in the code review was doing when they asked "what are you actually trying to solve here." They were checking for the XY problem before engaging with the implementation. The junior who learns to ask that question of themselves, before reaching for the tool, before writing the prompt, before opening the issue, has internalized the most important thing open source teaches.

When I watch junior engineers struggle with AI tools today, it's almost never a technical problem.

It's a problem formulation problem.

They're bad at structuring the question, not because they're bad engineers but because they haven't yet developed the habit of asking "what am I actually trying to solve here" before reaching for the tool. They go straight to the agent with a vague sense of what they want. The agent produces something. They take the output and keep going.

The senior in a code review would have stopped them before that. "What are you actually trying to solve here?" Asked enough times, by enough seniors, in enough reviews, that question is how they internalize it. How they start asking it unprompted before anyone else has to.

Without that environment, the junior keeps getting output. Just not always the right output. And they don't yet have the experience to tell the difference between a good answer to a bad question and a good answer to the right one.

The AI doesn't teach that distinction. It's too helpful. It tries regardless of how vague the input is. It doesn't say "your question is too vague." That's useful once they know how to evaluate the output. It's a trap when they don't yet know what good looks like.

I watched it happen in a mob programming session.

Mob programming works because everyone is in the thinking together. The navigator holds the direction. The driver holds the keyboard. Everyone else watches, questions, catches what the driver missed, builds the shared understanding that makes the session worth more than the code it produces. The output is almost secondary. The point is what happens in the room while it gets made.

Then someone, not the navigator, just someone, started prompting on their own machine. Through efficiency, not malice. The agent was faster. A solution arrived. The group looked at it. We moved on.

The mob didn't break dramatically. Nobody announced a change of approach. Someone just quietly did it differently, the output appeared, and the group followed the path of least resistance. The session continued. The mob stopped. The fun went with it, and so did the learning. Knowledge that used to build through watching each other reason, through the disagreement and the question and the "wait, what about this edge case," stopped accumulating the moment the output arrived pre-formed.

We were still in the room together. We had stopped thinking together.

That's the Alexandria Problem: not an event but a process. The library doesn't burn in one dramatic fire. It burns one session at a time, one private prompt at a time, one skipped conversation at a time. The knowledge that used to accumulate through proximity and repetition stops accumulating the moment the friction disappears. And AI removes the friction so efficiently that no one notices the burning until the smoke is already gone.

It's the Codex problem from the first chapter, seen from the inside this time. What never leaves someone's head is exactly what the next agent can't reach, and a private prompt is how it stays there.

The Spec Session is the fix: it gets what the team knows together out of private heads and into a form the next agent can reach.

It happened in code reviews, in pair programming, in the hallway conversation where a senior explained why the approach was wrong while the work was still an idea. It happened by sitting next to someone and watching how they thought: the iteration, the failed attempts, the moment they realized they'd framed the problem wrong and started over.

AI tools, used individually, break that transfer. Just by being available. The junior doesn't need to ask the senior. The agent is faster and less intimidating. The senior doesn't need to explain; they can just do it themselves. The moment of friction that used to produce a teaching interaction gets resolved by the tool before it becomes a conversation.[31]

[31] Anthropic put numbers on this in a randomized controlled trial: junior engineers learning a new library with AI assistance scored 17% lower on comprehension minutes later — nearly two letter grades — with the largest gap in debugging, while the speed gain was not statistically significant. How they used the tool decided the outcome: those who delegated generation or leaned on AI to debug scored lowest; those who asked conceptual questions or interrogated

The junior develops their own AI workflow in isolation, without the senior's eye on the process.

What they're not developing is the underlying skill. The problem formulation. The question that focuses the work before a prompt is written. That skill requires a human to model it, and a human to ask "what are you actually trying to solve" enough times that the junior starts asking it unprompted.

The Spec Session is the new learning environment, arrived at by consequence rather than design.

When the team is in the room together, building the spec before the agent runs, the junior watches how seniors frame problems. They see the questions that get asked before anyone proposes a solution. They see an experienced engineer push back on a direction not because the implementation is wrong but because the problem statement isn't tight enough. They see what "ready for the agent" actually looks like, and more importantly, what it doesn't look like yet. They learn it from watching humans think through what needs to be generated and why, not from reading generated code.

The mob exploration works the same way. A new library, a new approach, a PoC the team runs together. The junior isn't just evaluating the technology. They're watching how experienced engineers evaluate tradeoffs. What questions they ask. What concerns they raise. What "good enough for now" means versus "this will hurt us later."

It runs both ways. We were evaluating high-availability tooling once: paid caching, Redis, the works. A junior asked why we didn't just poll and persist to a volume on the cluster. We'd been too deep in the sophisticated option to see the plain one. He dragged us out. He was learning how the team weighed tradeoffs by sitting in the room, and from outside the hole we'd dug, he asked the question we'd stopped asking.

You can't get that from a spec document. You can't get it from reading the agent's output. You get it from watching how the thinking is done before

---

the generated code preserved their learning, and the conceptual-inquiry group, errors and all, was among the fastest. The authors note the study used a chat assistant, not an agentic tool, and expect agentic impacts to be more pronounced. Judy Hanwen Shen and Alex Tamkin, "How AI Assistance Impacts the Formation of Coding Skills" (Anthropic, January 2026; arXiv:2601.20245).

the answer exists.

None of this is formal teaching. Nobody stands at a whiteboard explaining problem formulation. The learning happens because experienced engineers make their thinking visible. The junior learns how the answer gets found.

The most important skill for working with AI is the one that can only be learned by watching humans work: problem formulation, context construction, asking the question that makes the answer findable.

If the team stops being the place where that happens, everyone heads down with their own tools, and the next generation develops faster in some ways and not at all in the one that matters most.

It's not a prompt engineering course but a career spent in rooms where people think out loud, ask hard questions, and show the junior what careful problem formulation looks like by doing it in front of them.

Seniors develop prompting habits nobody else sees. Architects discover patterns nobody else inherits. An experienced engineer figures out a better way to frame a class of problem, and that insight stays private, because there's no code review to surface it, no pairing session to transfer it, no hallway conversation to pass it on. Knowledge that used to become team knowledge remains personal knowledge.

The junior feels the impact first, but the loss belongs to the whole team.

When the senior's attention shifts upward, from implementation to architecture, the junior is often left downstream, catching the behavioral edge cases the senior no longer has the bandwidth to notice. That division of attention can work, but only if it's deliberate.

The senior whose attention has shifted to the architectural plane has a new responsibility in the Spec Session: to pull the junior up with them. To ask the questions that make architectural thinking visible: why this approach, what happens at scale, what would break this in six months. And to ask them not just in private, but in the room, with the junior present and expected to engage.

If the senior's attention shifts but the junior is left alone to catch whatever the agent missed, the junior develops a different and narrower skill: valuable, but not the same as learning to think architecturally.

The skill that makes a great engineer is the one thing the agent can't teach, because it's learned by watching a human do it. If the room goes quiet, the next generation inherits the tools without the judgment that makes the tools worth having.

But if the room is where that judgment transfers, how does anyone know the transfer is working? How does anyone know the understanding is actually shared and not just assumed?

The Oracle

The Oracle at Delphi was the most authoritative voice in the ancient world. Kings and generals sought its guidance before every major decision. The oracle always answered. The answers were always true. And they were almost always misread.

If you can't see understanding, how do you know it's there?

I was once told I had a bad image because I hadn't posted anything in the demo channel.

My direct lead pointed it out. He wasn't seeing my contribution and was questioning whether I was contributing at all.

What I had been working on was a package-private, domain-driven setup for a Java service: clean encapsulation, limited visibility of methods and classes, as little side effect as possible. The kind of structural decision that makes everything built on top of it easier, safer, and more maintainable. The bedrock that lets features ship reliably without unexpected interactions. The foundation: nothing shippable, nothing to demo.

Nothing to show in a demo channel. Nothing that looked like progress from the outside.

I nodded and ignored it. The measurement seemed wrong and I knew the work was right. But my role was eliminated shortly after. There were other factors; there always are. But I've never been able to fully separate the two. When a metric doesn't see someone's work, the people reading the metric don't see it either.

What did I do after being told? I pushed back. I didn't start posting in the demo channel. I informed the team about what I'd been told, questioned it openly, and named it as a bad metric, in the same category as review request

count, lines of code, or tokens burned. Metrics that measure activity rather than value, and that get gamed the moment they become targets.

Questioning the oracle. The signal was clear but read as the wrong thing. Naming that is a more useful act than changing your behavior to satisfy it.

Because the metric couldn't see it. The demo channel saw nothing. The dashboard showed nothing. The decision-maker saw nothing. And when I think about what might have changed the outcome, it wasn't doing different work. It was being asked a different question. Not "why aren't you in the demo channel" but "what are you working on." The first has a measurable answer. The second requires a conversation.

The metric was visibility. The work was invisible. The oracle gave a true signal, and nobody thought to question the reading.

Teams have always rewarded the visible over the valuable. The feature over the refactor. The new thing over the stable thing. The demo over the test suite.

But AI makes it catastrophic. Because now the demo channel can be full every day without much effort. Ship a feature, open a review request, close a ticket. The numbers look incredible. And the invisible work, meaning the quality, the structure, the foundation, gets starved of attention because it doesn't register anywhere that matters.

The person doing the invisible work gets told they have a bad image. Or loses their role. Or quietly stops doing the work that nobody can see.

We used to confuse effort with value. AI lets us confuse output with value.

DORA's research on generative AI[32] puts numbers on this. For every 25% increase in AI adoption, local process metrics improve: code quality, documentation, speed of code reviews. The same increase correlates with a 1.5% drop in overall delivery throughput and a 7.2% drop in delivery stability. Developers using generative AI report higher flow and personal satisfaction while spending less time on what DORA categorizes as valuable work.

[32]DORA, *Impact of Generative AI in Software Development* (dora.dev/research/ai/gen-ai-report), the generative-AI companion to the *Accelerate State of DevOps Report 2024*. The figures are DORA's estimated associations, not causal claims: a 1.5% reduction in delivery throughput and a 7.2% reduction in delivery stability for every 25% increase in AI adoption.

These are correlations, and DORA is careful to say so. The explanation it offers is a hypothesis, not a finding: AI lets a team produce far more code in the same time, changesets grow, and bigger changes have always been slower to land and quicker to break. The time saved gets lost downstream: in toil, in reviews that can't keep pace, in merges too large to be safe. The oracle measured the right things at the wrong level.

DORA isn't the only signal pointing this way. When METR ran a controlled trial[33], experienced developers worked 19% slower with AI on code they knew well, and came away convinced it had sped them up by about as much.

The metrics we use to measure engineering output were already wrong before AI. AI just made them easier to game and faster to break.

Lines of code. Story points. Number of review requests. Tokens burned. Each of these measures something real in isolation. Each becomes meaningless the moment it becomes a target.

Goodhart's Law[34]: when a measure becomes a target, it ceases to be a good measure. The metric stops describing the thing that matters and starts describing the behavior the metric incentivizes.

An AI power user can inflate every one of those metrics simultaneously. Lines of code go up. The agent writes more than a human would. Story points get closed faster. The agent implements tickets in hours. Review requests multiply. Why open one when five will do. Tokens burned increase, and that's just the cost of running the agent.

And the dashboard looks great. Leadership sees throughput. The demo channel is active. The burndown chart is trending in the right direction.

The review queue is growing underneath it. The specs are getting thinner because there's no time to write them properly when the agent is already running. The codebase is accumulating decisions that made sense in isolation and don't fit together. The team is heads down, individually productive, collectively drifting.

---

[33]METR, *Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity* (metr.org, July 2025; arXiv:2507.09089). Sixteen experienced developers were 19% slower with AI on repositories they knew well, yet believed it had made them roughly 20% faster.

[34]After the economist Charles Goodhart (1975); the familiar phrasing is Marilyn Strathern's (1997): "When a measure becomes a target, it ceases to be a good measure."

These are all individual metrics. Lines of code one person wrote. Story points one person closed. Review requests one person opened. Tokens one person burned.

That made partial sense when individuals were the unit of delivery. When one engineer owned a feature end to end, designing it, building it, testing it, shipping it, measuring the individual said something real about the output.

In a team-delivered, agent-executed workflow, the individual is no longer the right unit. The agent handles implementation. The value comes from what the team understood and agreed on before the agent ran: the spec, the shared context, the collective judgment about what to build and why. None of that shows up in an individual's metrics.

Worse, individual metrics actively work against shared understanding. If story points closed is how an engineer gets recognized, the incentive is to close tickets, not to spend time in a spec session building context that benefits the whole team. If review requests opened is the signal, the incentive is to keep the agent running, not to slow down and question whether the spec is right.

The metric gets the behavior it incentivizes. And individual output metrics, in a team context, incentivize exactly the wrong behavior.

The usual response when this becomes visible: add another skill to the agent. Improve the prompt. Use a better model. Find a way to make the AI produce better output with less review.

The output isn't the problem. The process that produces it is. Adding capability to the agent doesn't fix a broken spec. It produces better-written code against the wrong intent, faster.

Measuring the wrong thing incentivizes the wrong behavior. And if the behavior gets fixed by adding more AI capability without fixing the measurement, the incentive has been made more powerful, not less.

More tokens burned. More review requests. More story points. Better image in the demo channel.

The metric that actually matters is predictability.

Predictability, not velocity, not throughput, not any measure of how much output the team is producing.

Can the team tell a stakeholder when something will be done, and then actually do it by then?

That's the signal that the process is working. That the specs are good enough to estimate from. That the team understands the work before it starts. That surprises are rare because the thinking happened upfront.

A team with high velocity and low predictability is guessing fast. A team with lower velocity and high predictability has shared understanding. The second team is more valuable, more trustworthy, and more capable of scaling, because when capacity is added to a predictable process, the predictability holds. When AI is added to a broken process, the breakage accelerates.

Predictability is also the metric that can't be easily gamed. Either it was called right or it wasn't. No amount of demo channel activity changes whether the thing shipped when the team said it would. No number of tokens burned makes a forecast accurate.

And predictability is a team metric, not an individual one. One person can't be predictable alone. Predictability requires shared understanding of the work, shared ownership of the estimate, shared commitment to the timeline. It requires the Spec Session, or something like it: a moment where the team agrees on what is being built and how long it will take, while the work is still on the page.

None of this makes predictability immune to gaming. A team can pad the estimate, promise six weeks for work that takes three, and hit the date every time. But that only holds as long as nobody outside the team asks why three-week features take six, and stakeholders eventually do. Sandbagging predictability takes the whole team agreeing to the same lie and holding it quarter after quarter. Inflating throughput takes one person and an agent for an afternoon.

Teams have always known that upfront alignment produces better estimates, and that's not a new idea. What's new is that AI makes the alternative, skipping the alignment, running the agent, seeing what happens, feel so productive in the moment that the incentive to do the upfront work disappears.

The demo channel fills up. The forecast gets noisier. The image looks great.

Output is easy to buy now. A thousand tokens spent implementing the wrong thing creates exactly the same value as a thousand hours spent implementing the wrong thing: none.

The first team has four times the review request volume of a year ago. Tickets close fast. The demo channel is active. The burndown chart looks healthy. Reviews are backing up. Commitments slip. Stakeholders have stopped trusting the estimates. Rework is quietly eating the gains.

The second team ships less visibly. Fewer review requests. Quieter demo channel. But when they say something will be ready on Thursday, it's ready on Thursday. Reviews are fast because the specs were tight. Stakeholders plan around their commitments. Surprises are rare.

The first team is optimizing for output. The second team is optimizing for understanding.

From a dashboard, the first team looks more productive. From a planning meeting, the second team is more valuable.

Reliable forecasts are valuable because they're evidence that the team actually understands the work.

When a team repeatedly predicts correctly, it tells a manager something deeper than delivery dates. It says the team shares a model of the system, the problem, and the solution. That the specs were good enough to estimate from. That the thinking happened in the Spec Session. That surprises are rare because the alignment was real.

The thing to measure isn't how much the team is producing.

It's whether the team can say what they'll produce, and when, and whether they're right.

Output is abundant now, but shared understanding isn't. The teams that turn understanding into reliable commitments are the ones creating value.

Everything else is just numbers getting easier to game.

Predictability shows that the team understands the work. What it doesn't show is how that understanding gets built.

The Trireme

The trireme was the fastest warship of the ancient world, not because of its size, but because of its structure. Three banks of oarsmen, each pulling in coordination. Remove one bank and it slows. Remove two and it barely moves. The power was never in the number. It was in the arrangement.

If predictability is the proof, what kind of team produces it?

So what does the team actually look like?

Every team is different, and what follows is a starting point, not a prescription.

Team sizes have always been a function of the bottleneck. In traditional agile, 7 to 10 people was the recommended unit, calibrated to sustain an implementation pipeline. The number was the answer to a specific question: how many people does it take to keep implementation moving?

Agents change the answer, because they change the question. The Spec Session is now the bottleneck. It gets slower as the session fills up and convergence takes longer to build a shared picture.

So the logic that produced 7-to-10 person teams no longer applies. The right team size is now the one that can run an effective Spec Session: as many as the process requires, as few as it can sustain.

Two people can challenge each other right up until they agree — and then the agreement has no one to test it. Heads down together, they stop seeing what they've stopped seeing. The third perspective isn't there to referee disagreement; it's there to be outside the consensus the pair forms.

Two people also fail the redundancy test, and it's the same test. One person holding critical system knowledge is a single point of failure. Losing one person from a pair doesn't split the difference — it leaves an individual, which is the single point of failure you started with. Three is the smallest number where losing anyone still leaves a pair: still someone to challenge, still someone to be challenged.

What redundancy has to preserve was never headcount. It's the capacity for friction. Three is the floor because it's the smallest team that stays a team after losing anyone. That's the minimum viable unit — not asserted, but what's left once you ask what two people can't do.

Someone who holds the system direction, the direction work from the End of a Craft chapter, now named as a role: the person who tracks what the system is becoming and whether the decisions being made are consistent with that, regardless of title or seniority. In a small team this is often the same person who runs the Spec Session. But there needs to be someone in every team.

Someone who watches the ground. The behavioral edge cases, the implementation details, the things that drift when nobody is looking. This is often where juniors do their most valuable work. The junior catching what the senior's attention has moved past, the dynamic from the Fool with a Tool chapter, is by design.

Someone who manages the outside. Stakeholder pressure, organizational noise, the requests that arrive before anyone has had time to spec them. The Stakeholder Navigator from the Product Owner is Dead chapter. With that role filled, the room stays protected long enough to think.

The agent is the fourth, running only after the other three have done their work.

I've watched these functions show up without anyone assigning them.

In OpenFeature, a contributor once noticed something the maintainers had stopped seeing. Certain providers broke in a subtle way when reused, not because the code was wrong, but because it relied on something the code never said. If you knew the assumption, everything worked. The code didn't carry the assumption. The people did.

The cheap fix was obvious. Write it down. A documentation change, a warning in the README.

We were close to taking it. Then one of the maintainers pushed back. An implicit contract that lives in documentation is a contract that fails the moment someone builds without reading it, and someone always builds without reading it. He argued for an explicit interface instead. More work now, less archaeology forever.

None of this happens unless someone accepts the issue first, looking at a report from outside the project and deciding it deserves the project's attention. That sounds like housekeeping. It isn't. It's the seat where the project meets everyone who uses it and will never show up in person.

Three kinds of attention, and nobody assigned any of them. Someone with eyes on the ground, catching what familiarity had made invisible. Someone answering to the people outside the room. Someone holding direction, thinking past this pull request into the years after it. Not roles in a process document. The structure of the project just makes them show up.

And underneath them, redundancy. The fix didn't go through on one person's say-so. Other maintainers weighed in, pushed on the interface argument, approved it. No single mental model decided the outcome, and no single departure could have taken the understanding with it.

Still, some organizations will operate at 15 or 20 people, for good reasons: scale, domain breadth, regulatory complexity. What changes is how those teams structure themselves internally.

The model that works is stream-aligned teams[35]: smaller units, each running their own Spec Session on their slice of the problem, each holding deep shared understanding within their domain. Subject matter experts who own their territory. The larger team doesn't need a single room where everyone understands everything. It needs enough boundary knowledge between units that handoffs don't lose context.

Across the whole, what everyone shares is the overall goal, the direction, and the constraints, not the implementation detail of every domain. That shared picture is wider and shallower, answering different questions at different scales.

---

[35]Matthew Skelton and Manuel Pais, *Team Topologies: Organizing Business and Technology Teams for Fast Flow* (IT Revolution, 2019).

As teams grow larger, agents multiply, and output speeds up, the gap between what the system is doing and what the team collectively understands grows, and the book doesn't pretend to have closed it. Spec Sessions narrow it. Re-alignment across domains narrows it. A platform that makes the invisible visible narrows it. None of them remove it, because the pressure toward speed and the pressure toward drift are the same force under two names.

It's worth being exact about where this model holds and where it stops. The three-role unit contains drift inside a domain: someone holds the direction, someone watches the ground, someone manages the outside, and the picture stays shared because the room is small enough to keep it shared. What it doesn't contain on its own is drift between domains: the gap that opens when ten of those rooms each stay aligned internally and slowly stop aligning with each other. No structure closes that gap by itself. It stays closed only as long as the stream-aligned teams keep choosing to re-share what they've learned, often enough, and early enough, that the divergence never compounds.

That between-domains problem is the layer where Team Topologies operates. Skelton and Pais, whose vocabulary this chapter borrows for its own domain units, map how teams should be shaped and interact at scale, so cognitive load never exceeds what a team can carry: the inter-team question of how the rooms hand off to each other. This book sits one layer down, on the understanding a single room has to build before the agent runs. Team Topologies assumes the work is already understood and optimizes the handoffs; this book argues the understanding has to be built first, and that it takes at least these three functions in the room together. The two are adjacent, not competing; cognitive load exceeding capacity and shared understanding failing to form are the same symptom seen from two layers: teams drifting, decisions made in isolation, systems nobody fully holds.

That's the honest limit. The structure buys aligned domains, not an aligned organization. The only thing that buys the second is the thing the book has been pointing at all along: a room people actually use, repeated at every level that needs one. What has to hold is the room, and the connections between rooms. Whether they exist, and whether people use them, decides everything else.

Desert and Forest

Kent Beck describes two working environments: the desert, open, fast, unobstructed, where you can move without friction, and the forest, dense, slower, where the ecosystem is richer and learning accumulates in the undergrowth.[36]

If the right structure holds understanding, what keeps it alive over time?

Beck built Extreme Programming[37] on the conviction that software development is a social activity, that the team is not overhead but the mechanism through which good software gets made. XP was largely set aside in favor of frameworks that separated thinking from building, planning from implementation, people from each other. This book is, in part, an argument that XP was right, and that AI is forcing us to rediscover why.

AI is the most powerful desert tool ever built. It removes almost every obstacle to individual speed, and a developer can move faster alone with an agent than any 10x engineer[38] could move alone before. The desert has never been more accessible, or more tempting.

Which makes the forest more important than it's ever been, and more fragile.

The Trireme chapter gave the structural answer: someone to hold the

[36]The forest-and-desert metaphor is Beth Andres-Beck and Kent Beck's. See "Forest & Desert," *Tidy First?* (tidyfirst.substack.com/p/forest-and-desert), and Martin Fowler, "Forest And Desert" (martinfowler.com/bliki/ForestAndDesert.html).

[37]Kent Beck, *Extreme Programming Explained: Embrace Change* — discussed, with citation, in the introduction.

[38]"10x engineer" traces to Sackman, Erikson, and Grant's 1968 study on individual programmer productivity variance, popularized in software culture via Steve McConnell's *Code Complete*.

direction, someone to watch the ground, someone to manage the outside, with enough redundancy that no single person is a point of failure.

All of it depends on the same precondition: a room where uncertainty is discussable.

The Agora chapter called psychological safety the prerequisite. The sharper version is epistemic safety: the conditions where people reveal what they don't know. Where the junior asks the question without calculating the risk first. Where the senior shares the reasoning, not just the conclusion. Where the half-formed thought gets voiced instead of suppressed. Where assumptions become visible before they become decisions.

The room needs the freedom to be uncertain out loud.

That room is the precondition for everything the book has argued. A team can't build shared understanding without it. A team can't get weighted validation without it. A team can't catch the XY problem without it. A team can't grow the next generation without it. A team can't have constructive friction without it.

The process creates the container; the safe room is what fills it.

What breaks the room is rarely dramatic. The subtler damage comes from well-intentioned behavior that lands wrong.

The engineer who questions everything in a review request. Rigorous, thorough, evidence-based. From the inside it feels like raising the standard. From the outside it feels like being interrogated. The junior stops raising concerns because last time it became a debate. The senior stops sharing uncertain thinking because uncertain thinking gets challenged. The room gets quieter. Safer-feeling on the surface. More fragile underneath.

Or the 10x engineer who is genuinely fast, genuinely capable, and experiences the team as an obstacle. Optimized for the desert, not malicious about it. They've been rewarded for individual speed their whole career. The Spec Session feels like overhead. The review process feels like a bottleneck. The team feels like it's slowing them down, because it is, because that's partly what the team is for.

I was somewhere on that spectrum. Not hoarding knowledge or dismissing

others, but asking too many questions in the wrong format, ending messages with exclamation marks out of habit that read as aggression rather than enthusiasm, and once, memorably, sending colleagues links to blog posts explaining why direct messages destroy flow and that we should stick to processes.

I was right about the substance. Direct messages do interrupt flow. The blog posts probably made good arguments. But sending links to colleagues to explain why their behavior was wrong, however valid the point, reads as condescending. The message was about process. The impact was about respect. I knew as soon as I sent it. I apologized. The escalation came anyway.

The room was telling me the room was breaking, and not gently.

The ego that breaks the room isn't usually the ego that knows it's an ego.

It's the engineer who cares deeply, has strong opinions, asks hard questions, and doesn't yet know how the questions land. The desert virtues, namely rigorous, fast, evidence-based, and process-oriented, are genuinely good engineering habits. They just work differently in the forest.

In the desert, questioning everything in a review request optimizes for correctness. In the forest, it optimizes for silence. The question that makes someone feel tested produces a room where people stop volunteering the uncomfortable truth. And the uncomfortable truth is exactly what the room needs.

AI accelerates this dynamic. The engineer already inclined toward the desert gets a tool that makes it so productive the team starts to feel like friction.

Previously, leverage meant individual output. The 10x engineer was the one who wrote code fastest, shipped most, closed the most tickets. That definition made sense when implementation was the bottleneck.

When the agent handles implementation, the bottleneck moves. The constraint becomes the quality of the understanding before the agent runs: the spec, the shared context, the room where the direction got challenged before anyone built anything. The engineer who improves the quality of that room now has more leverage than the engineer who writes code fastest.

The 10x engineer of the AI era is the person who makes the room better. Who

asks the question that surfaces the XY problem. Who creates the conditions where the junior speaks up. Who slows down the Spec Session just enough to catch the misunderstanding before it becomes rework.

Until the spec was wrong. Until the direction was off. Until the edge case surfaced after the agent had run three times. Until the junior who would have caught it in the room had stopped speaking up.

A counterargument worth taking seriously: some of the best software ever written came from individuals or tiny teams. Linus Torvalds. Wozniak. The early internet. The desert can produce extraordinary things. What those examples don't prove is that the desert scales. They show what's possible when one person holds the full picture. The question is what happens when the system gets complex enough that no single person can hold it, when the architecture spans more context than one mind can carry, when the edge cases live in the gap between what one person knows and what their colleague knows. That's where the forest earns its place: by surviving what the desert can't.

I've developed a small habit at conferences and events. When standing in a group, never close the circle completely. Leave a gap. Physical space that says: this conversation isn't finished, you're welcome to join. No invitation required. Just an open space where someone could step in.

It's the same instinct as switching to English the moment I know there's someone nearby who isn't a German speaker. Not because they can't understand, often they can. But because engagement is easier when the barrier is lower. The appreciation shows immediately, even from people who would have followed in German. Something relaxes. The conversation opens.

It's about inviting collaboration before anyone has to ask for it.

The desert closes ranks: tight clusters, known quantities, optimized for the people already in the room. The forest leaves space before it's requested. It accepts the friction of the unexpected perspective because it understands that the unexpected perspective is often the one that matters.

The Spec Session works the same way. The session that starts with "we've thought about this and here's what we're doing" has closed the circle. The one that starts with "we have some thinking but we haven't decided" leaves the gap. The junior's edge case, the stakeholder's clarification, the concern

nobody voiced in the corridor: those arrive through the gap. They don't arrive in a closed circle.

The gesture is small. The effect compounds.

Building the room is slower than moving alone. It requires attention to how things land, not just whether they're correct. It requires making thinking visible even when visible thinking feels inefficient. It requires the senior who shares uncertainty, the lead who thanks the junior for the uncomfortable question, the culture that treats friction as information rather than resistance.

It requires, sometimes, someone telling an engineer that their blog post links were not well received.

The forest grows slowly but survives what the desert cannot.

A team with a safe room and moderate individual capability will outperform a collection of 10x engineers who broke the room. Not because capability doesn't matter. It does. But because the room is where the capability compounds. Where the junior's edge case catches the senior's blind spot. Where the Spec Session surfaces the misunderstanding before the agent burns the budget. Where the different perspective changes the direction before it's too late to change.

Team size is the Trireme chapter's question, not this one. But two things hold at any size. A team needs friction; a team where everyone always agrees is a mirror, and mirrors don't catch mistakes.

A team needs the room. Without people saying what they actually think, nothing else holds.

The desert produces output. The forest produces learning.

Learning is also what the organization keeps losing, privately, locally, one engineer's hard-won configuration at a time. That cost is real, and it compounds. It stays quiet, which is why the desert always looks cheaper than it is.

The Astrolabe

The astrolabe was the instrument that made celestial navigation possible: a tool for calculating position from stars that were always there but previously unreadable. Before it, sailors estimated. After it, they could know.

If one team can sustain it, how does it scale to many without thinning?

Every developer is currently running their own agent.

Not every team, every individual. Local machine. Local workflow. Local configuration. Local prompting habits built up through private experimentation that nobody else sees. The AGENTS.md[39] file, if it exists at all, was written by whoever set up the project and hasn't been touched since. The skills and MCPs each engineer uses reflect their own discovery process alone.

A twenty-person engineering organization doesn't have four or five different agent configurations. It has twenty. The fragmentation is at the individual level, which makes it harder to see and harder to fix.

This is where software teams were before platform engineering existed. Before anyone decided that deployment pipelines, observability stacks, and infrastructure configuration were too important to leave to individual teams with different standards and no shared visibility. Every team managed their own servers. Improvements stayed local. Knowledge didn't transfer. The cost of doing it badly was paid independently by each team, invisibly, with no way to see the pattern across the organization.

The same dynamic is playing out now with AI infrastructure. And the solution is probably the same one.

[39] AGENTS.md is discussed, with citation, in the End of a Craft chapter.

The cost of a bad prompt is currently invisible.

When an engineer runs the agent against a thin spec and gets the wrong output, they rerun it. Better prompt this time, or different framing, or more context. The agent runs again. Maybe it works. Maybe it runs again.

That cost, in tokens, in time, in engineering attention, goes nowhere. No ticket captures it. No metric tracks it. The AI budget is treated like electricity: a fixed cost of doing business, not something traced back to individual decisions or process failures.

Which means nobody knows which prompts are expensive. Nobody knows which rewrites were caused by bad specs. Nobody knows whether the token spend last month produced shipped value or produced rework. The cost is real and growing; the visibility is zero.

This is the measurement problem from the Oracle chapter, one layer deeper. Nobody can measure what they can't see. And right now, in most organizations, almost nothing about AI spend is visible at the level where decisions get made.

A centralized agent infrastructure changes that.

It won't be the only way to run an agent. Engineers will still experiment locally, the same way they still have local terminals despite CI/CD existing. The platform becomes the production path and the source of truth, not the sole execution environment.

The paved road is attractive, not mandatory. But when agent work runs through the platform, something changes. Every run is traceable to a team, a ticket, a spec. Rerun rates become visible. The correlation between thin specs and expensive implementations becomes measurable. The organization can see, for the first time, what a bad prompt actually costs, then trace it back to the process failure that caused it.

That's organizational telemetry for AI development. The same way a team instruments its system to understand its behavior, it instruments its agent platform to understand its process.

Platform engineering exists because reliability, deployment, and observability were too important to leave to individual teams with local standards and no shared visibility. Not because individual teams couldn't build those things. Because the platform team sees across every team simultaneously. More signal. More patterns. More ability to spot what's working and what isn't. And the mandate and resources to act on it.

The same logic applies to agent infrastructure. An individual team maintains their own agent configuration with local data and limited resources. A dedicated platform team, agent platform engineers effectively, sees across every team simultaneously. They observe token usage patterns, rerun rates, prompt failure modes. They spot the skill that's producing consistently poor output and fix it centrally. They notice that three teams have independently solved the same prompting problem and consolidate the solution into the shared foundation.

Platform engineering turns local discoveries into organizational assets. One engineer figures out a better way to frame a class of problem. Under the current model, that insight stays private. Under the platform model, it gets contributed, reviewed, and inherited by every team. The discovery compounds instead of dying locally.

The improvement that would have lived in one person's AGENTS.md file becomes the new default for the whole organization. One better skill, inherited by every team. One fixed prompt failure mode, gone everywhere at once. One shared MCP that three teams were building independently, now maintained centrally and available to all.

That's the rate at which the organization gets smarter about working with agents. The platform is valuable not because it saves money but because discoveries stop dying locally.

The open-close principle[40] applies. The foundation is closed: stable, trusted, maintained by people whose job it is to make it better. The team-specific layer is open: customizable, extensible, without breaking what the foundation provides. Teams are users of the platform who can also contribute upstream when they find something better. The same model open source has operated on for decades.

---

[40]The open/closed principle, one of the five SOLID principles, originated with Bertrand Meyer (*Object-Oriented Software Construction*, 1988) and was later popularized as part of SOLID by Robert C. Martin. Applied here metaphorically to platform architecture, not OOP design.

Prompting knowledge is currently concentrated in individuals. The engineer who figured out how to frame a class of problem well, the team that discovered the right MCP combination for their domain, the workflow that produces consistently good output: that knowledge lives in personal habits and local configuration files. When the engineer leaves, it goes with them.

Shared infrastructure makes that knowledge organizational rather than personal. The skill that one engineer developed through private experimentation gets contributed to the platform and inherited by everyone. The MCP that one team built for their domain becomes available to teams facing similar problems. The agent configuration that produces good output in one context informs the default configuration for the whole organization.

It reaches the agent's memory too. The End of a Craft chapter warned about the version kept in one person's local setup, where the direction persisted but only for them, and any drift from a colleague's copy was written down where nobody else could see it. Held in the platform instead, that inverts: one shared picture, versioned and visible, handed to every agent in the organization. Someone still has to keep it true, but now it's one picture, in the open, where drift shows up instead of hiding.

None of that is simple to build. Shared memory is the hard version, where the moment many agents and many people write to the same store, what's one person's, what's the team's, what's global, and who reconciles two entries that disagree all become real questions. They're the platform team's to own, because it's too big for any one engineer to carry alone.

Operational knowledge that used to burn in Alexandria gets written into stone.

Through infrastructure: the living artifact that encodes what the organization has learned about working with agents, maintained by people whose job is to keep it current and improve it continuously.

But the platform only solves one layer of the Alexandria Problem. The deeper knowledge, meaning system understanding, architectural context, the reasoning behind past decisions, the edge cases discovered through years in the codebase, still lives in people. The platform can't capture it. The Spec Session is still the mechanism that keeps it shared and alive. The platform makes the operational foundation stable enough that the team's cognitive energy goes toward that deeper work instead of reinventing local

agent configurations.

Partly resolved. The rest still requires the room.

Individual engineers don't need to carry this cognitive load.

That's the other half of the platform argument. Right now every engineer is simultaneously trying to be productive with AI tools and develop their own understanding of how to use them well. That's a significant overhead: private experimentation, local configuration, keeping up with model changes, figuring out which skills work for which problems. It's the kind of overhead that experienced engineers absorb at the cost of other things, and that junior engineers struggle with because they're still building the foundation underneath it.

A platform removes that overhead from the individual and places it with the team best equipped to handle it. Engineers focus on the spec, the problem, the system thinking: the work that requires their expertise and judgment. The platform handles the agent configuration, the skill maintenance, the model selection, the infrastructure for running agent work reliably at scale.

The company grows as a unit. A shared foundation that improves continuously, maintained by people who see the whole picture, available to everyone who needs it.

A bad spec produces a rerun. The rerun costs tokens. The tokens are traced to the spec. The spec is traced to the team and the ticket. The cost of skipping the Spec Session, of running the agent before the understanding was shared, shows up in the data.

It's a feedback loop. For the first time, the organization can test whether the Spec Session matters. Not assume it. Not argue for it philosophically. Test it.

Does shared understanding before the agent runs produce fewer reruns? Does spec quality correlate with first-run success? Does the team that invests in upfront alignment ship more predictably than the team that skips it?

The platform creates the visibility to answer those questions. The argument the book has been making from the first chapter, shift left and invest in

understanding before a line is written, becomes falsifiable. There's no need to promise a specific result. The data just needs to exist.

A team that's an outlier in rerun rates gets a signal they didn't have before. It's a question, not a blame mechanism. Why are we rerunning more than comparable teams? Is the spec quality lower? Are prompts less specific? Is the agent running before the understanding is actually shared?

That question leads somewhere actionable. A dedicated session on prompting quality. A closer look at whether the Spec Session is producing shared understanding or just producing a document. A review of spec size: are tickets too large for the agent to act on reliably, or too small to carry enough context?

The platform team can facilitate that. They see the pattern across the organization. They know which teams improved their rerun rates and what changed. They can bring that knowledge to the outlier team, not as a top-down intervention but as "here's what we've seen work elsewhere."

That's organizational learning at scale. The platform sees what works, the outlier gets a signal, the improvement spreads. The Alexandria Problem in reverse: knowledge flowing from the center outward, continuously, as the organization learns what good looks like.

While writing this book, I shared the draft with a CTO who had independently built an AI-native workflow called PullOps. The conversation felt less like introducing new ideas and more like comparing notes. The workflow had converged on many of the same conclusions: specification before implementation, human validation at decision points, shared capabilities instead of individual tooling, and team review as a cultural practice that infrastructure alone cannot replace.

Interestingly, he started building it before he knew about the book.

Convergence on its own proves little. Put enough people on the same hype cycle, the same conference talks and the same launch posts, and they reach the same answer because they read the same things, not because the answer is true. What matters is what they converge *on*. The hype cycle pushes one way: more autonomy, fewer humans, the agent running unsupervised. PullOps went the other way: human validation at decision points, team review as a practice infrastructure can't replace. That's not the fashionable conclusion. It's the inconvenient one, and he reached it independently, by

hitting the same wall. The book names the pattern. The pattern was already emerging.

Charity Majors reached a version of the same conclusion from a different angle: production, not the spec.[41] Once code is cheap to regenerate, she argues, discipline relocates to wherever judgment still has to happen; for her, that's the telemetry and evals that catch what already shipped. This book relocates it into the room before the agent runs instead. Same anxiety, opposite pole of the loop. Maybe both are true. A spec nobody understood doesn't get safer because the telemetry after it is precise, and precise telemetry doesn't help if nobody held the intent behind what shipped.

A platform can make the cost visible. It can't make the choice. The data can show that the room is emptying, but whether it gets filled again is the one decision no instrument makes. That's the astrolabe: it tells you where you are, but never where you go.

[41] Charity Majors, "AI Demands More Engineering Discipline. Not Less." (charitydotwtf.substack.com, June 2026).

The Phoenix

The phoenix is a mythical bird that burns at the end of its life and rises from its own ashes. Not despite the destruction — because of it.

If all of this is possible — what do you choose?

If you've felt something was wrong but couldn't point to it, this book was written for you.

Not for the confident adopters. Not for the organizations announcing their AI transformation. For the engineer who is moving faster than ever and somehow feels further behind. The team lead whose team is still there on the org chart but the room has gone quiet. The person who senses the ground shifting and doesn't know yet whether to call it progress or loss.

We've been here before.

At some point — in your team, your organization, your career — someone decided that process and structure were slowing things down. The planning meetings, the reviews, the deliberate conversations before anyone wrote a line of code. Overhead. Friction. Things that got in the way of shipping.

So they went. Or they shrank. Or they became ceremonies that everyone attended without really being present. And for a while it felt like freedom. Faster. Lighter. More autonomous.

The decisions that used to get made in the room started getting made alone. The junior who would have raised the concern stayed quiet because there was no moment to raise it. The senior's reasoning stopped transferring because the senior was heads down in their own context. The system

accumulated decisions that each made sense in isolation and didn't fit together.

AI is offering the same deal at a larger scale.

Move faster. Skip the room. Optimize the individual. The tool is right there, frictionless, never tired, always ready to help. Why wait for the Spec Session? Why slow down for validation? Why sit in a room with people when the agent is ready to run?

The illusion is more powerful now. The output is real. The review requests are real. The tokens are burning, the tickets are closing, the demo channel is full. It looks like progress from every angle that gets measured.

But the room is emptier. The understanding is thinner. The junior is learning alone. The senior is prompting alone. The architect is deciding alone. And somewhere in the gap between all that individual speed, the system is accumulating something that will surface later as something nobody can quite explain. You can feel it before you can name it, and that feeling is the signal.

This book is an argument for a different choice.

The argument isn't against AI, and it isn't against speed. It's against the assumption that the team is overhead. Against the idea that shared understanding is a luxury you can't afford when the agent is ready to run. Against the optimization that removes the friction without noticing that the friction was doing something.

The Spec Session is slower than prompting alone. The Agora is harder to maintain than a solo workflow. The room where people say what they actually think is more fragile than a process that runs without it.

Teams matter — not as an abstract principle, but because the forest produces something the desert cannot. Because the junior watching the senior think out loud is building something that no prompt will ever teach. Because the pushback that changes your direction before the agent runs is cheaper than the rework after. Because the room where uncertainty is discussable is the room where the right thing gets built.

And this isn't only conviction. The numbers from the measurement chapter

point the same way[42]: DORA found AI lifting local speed while overall delivery slowed, and a controlled trial found experienced developers working slower with AI even as they felt faster. The forest choice has evidence under it, not just feeling.

The question is whether we choose before the room goes completely quiet, or after.

If you feel lost, you are not behind. You are noticing something that the metrics aren't capturing and the dashboards aren't showing.

The Phoenix doesn't rise by accident; it chooses to. That choice is the difference between what happened to Sisyphus and what can happen here. Sisyphus had no choice but to push the boulder. Teams do. Organizations do. Individual engineers do. The drift toward the desert, toward individual speed, toward the ever-agreeing genie — none of it is inevitable. It's a series of small choices that compound quietly until the room goes empty.

The renewal is also a choice. Keep the team. Protect the room. Slow down for the conversation that feels like it's getting in the way — because that conversation is often the work. Push back when the metric replaces the judgment. Say the thing that's uncomfortable to say in the Spec Session. Ask the question that surfaces the XY problem before the agent runs.

Choose the forest. Even when the desert is faster. Especially when the desert is faster.

Understanding what to build, together, before anyone builds it was always the hard part. AI didn't change that; it just made it easier to forget.

We can choose to remember.

[42]DORA, *Impact of Generative AI in Software Development*, and METR's July 2025 controlled trial. Both are discussed, with citations, in the Oracle chapter.

Acknowledgments

Glossary

About the Author

Simon Schrottner is a software engineer who has spent more than a decade working on the tools other engineers build on top of. He maintains OpenFeature, including its flagd implementation, and contributes as a CNCF and AAIF Ambassador. He speaks about OpenFeature and the systems around it at conferences across Europe and beyond.

Left of the Loop grew out of a blog series argued in public, post by post, stress-tested by readers who pushed back before any of it became a book. The failures in it are his own. That's still how he prefers to work things out.

He lives in Austria and writes at schrottner.at.