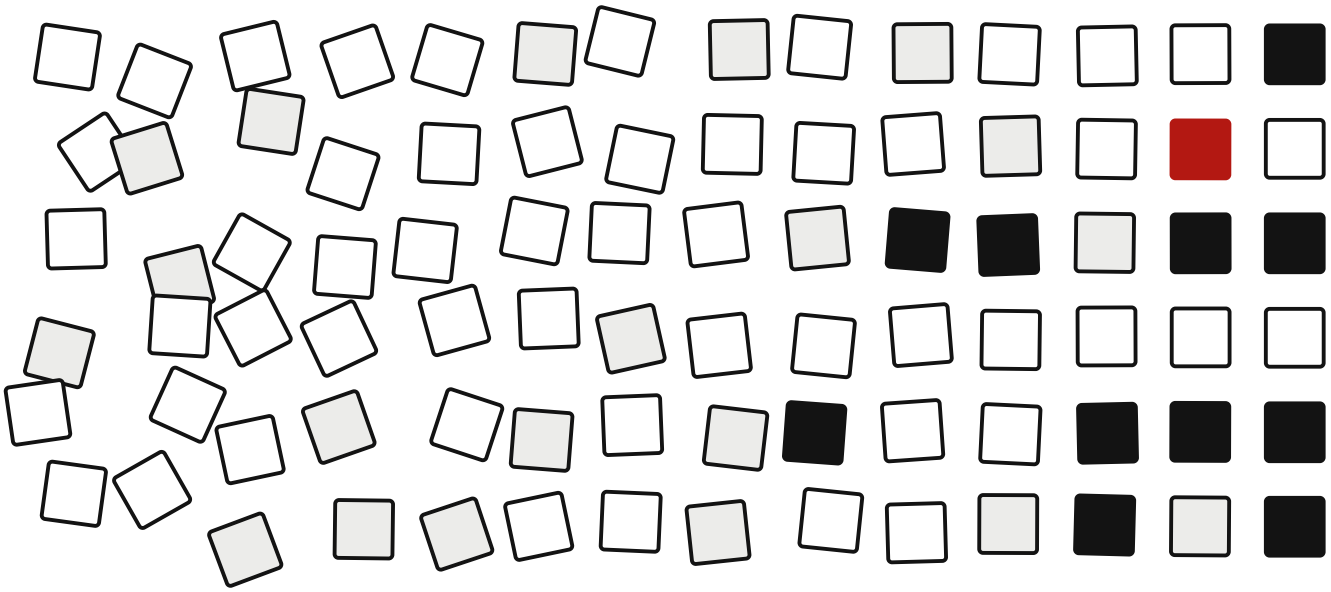
AINATIVESOFTWARE.ENGINEERING

Free guide
for leaders
30-MINUTE READ

Leading AI-Native Engineering

A 30-minute guide for CTOs who want their teams to build with AI agents



From the author of *AI-Native Software Engineering* (O'Reilly)

Alfonso Graziano

Why I Wrote This Guide

I help engineers work with AI agents on real code, from writing specs and providing context to checking the results. I also support heads of engineering and CTOs bringing these practices to their organizations. They ask what their people need to learn, which changes need leadership support, and why their investment in AI hasn't produced the improvement they expected.

In the past year, I've worked with tens of teams in different companies, and their leaders kept asking similar questions. This guide brings the answers together. It takes the practices from my book, *AI-Native Software Engineering* (O'Reilly), and looks at what they mean for you as a leader, including rollout lessons the book only touches on.

In the book, the harness has two layers. The *agent harness* comes with the coding tool and manages context, tool calls, and sessions. The *user harness* is what your team builds and maintains: repository instructions, specs, skills, integrations, and verification checks. You can customize parts of the agent harness, too.

For simplicity, I'll use *agent* for the coding tool you download, such as Claude Code, Codex, Cursor, or GitHub Copilot's agent mode. From here on, I'll generally use *harness* for the user harness, since that's our main focus, naming the specific layer when the distinction matters.

Contents

	Why I Wrote This Guide	2
1	You Bought the Licenses. Delivery Barely Moved.	3
2	What AI-Native Engineering Actually Is	5
3	What Your People Need to Learn	7
4	What Has to Change in the System	9
5	Your Job: What Only a Leader Can Do	13
6	Resistance Is Rational	15
7	A Rollout That Doesn't Break People	16
8	Measure Practices, Not Seats	19
9	Your Monday List	21
	About the Author and Free Tools	22
	The Book	23

1 You Bought the Licenses. Delivery Barely Moved.

"We bought the licenses, and things changed much less than we expected." I hear this often from engineering leaders. They paid for a coding assistant for every engineer and expected a noticeable improvement in delivery. Six months later, cycle time is a little shorter, but the backlog is about as long as it was, and there are a few more incidents than they are comfortable with. People are using the tools. The improvement just hasn't reached the organization in the way leadership expected.

That doesn't mean the tools failed. Most engineers who use them every day can point to tasks they finish faster. Follow one of those tasks through the rest of the process, though, and you may find the saved time disappearing. The change still waits for a reviewer and for product to decide whether it can go live. Making one engineer faster helps that engineer, but the team also needs to change how work moves between people.

The Bottleneck Moves

In its 2025 technology report, Bain & Company found that writing and testing code account for about 25 to 35% of the time from idea to launch. Halving the time spent on that work would shorten the whole cycle by about 13 to 18%, assuming everything else stayed the same and the team could use all the time it saved. That's the best case. The report also found productivity gains of 10 to 15% among teams using AI assistants. Figure 1 shows why faster coding has a limited effect on the whole cycle.

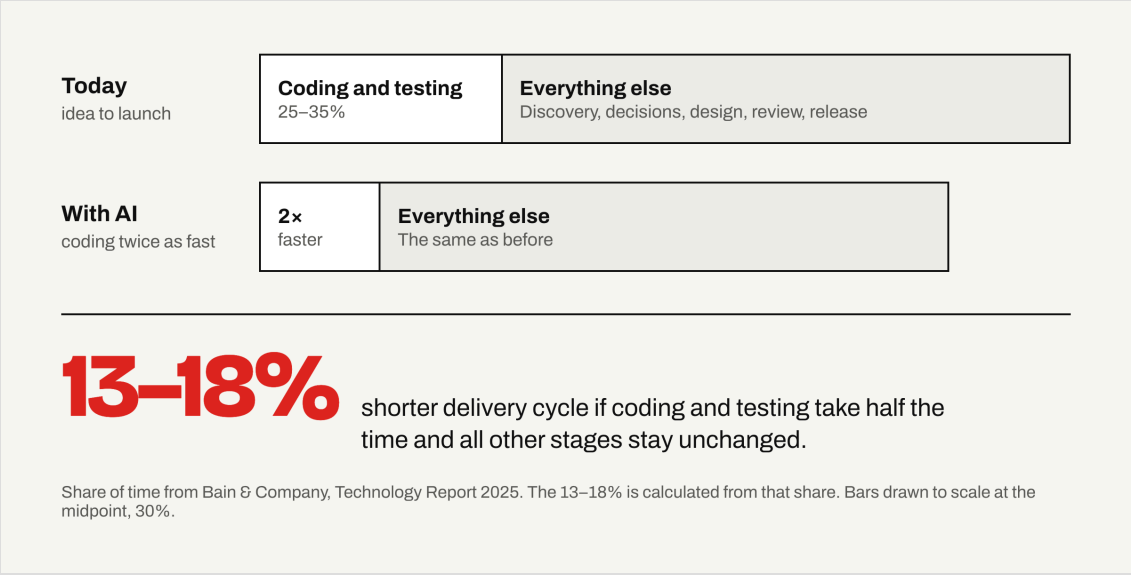


Figure 1. *Doubling coding speed shortens the whole cycle by only about 13 to 18%, at best.*

For you as a leader, the remaining 65 to 75% deserves attention. If coding gets faster while review capacity and product decisions stay the same, more work arrives at those stages than people can handle. The bottleneck moves, and the queue in front of it grows. Your next investment needs to address where the work is waiting now.

Three Failures You Only See at Team Scale

When engineers adopt AI individually, they can each be doing useful work and still create problems for the team. I see three of these problems in almost every team I work with.

Fragmentation

One engineer configures an agent with detailed rules. Another uses a different agent with very little context. Both are editing the same codebase, but their agents follow different conventions. Even when someone finds a setup that works well, the person next to them may have no way to reuse it.

Silent divergence

A spec leaves something open to interpretation, and two agents resolve it differently. Each pull request looks reasonable on its own. The team only discovers the disagreement weeks later, when the two pieces have to work together and nobody can explain why they made different assumptions.

Trapped knowledge

Someone learns how to structure context so the agent produces much better results. That knowledge stays in a private chat history. It never becomes a shared rule or a skill the rest of the team can use, so other engineers spend their time discovering the same thing.

These problems need agreement about how the team works. Buying more licenses leaves that work undone and brings more agents into the same fragmented setup.

A Mirror and a Multiplier

Google's summary of the 2025 DORA report calls AI "a mirror and a multiplier." If a team has shared standards and strong tests, agents let it do more work within those standards; in a fragmented team, they produce more changes that disagree with one another. Giving both teams the same tools can therefore widen the gap between them.

Figure 2 puts these dimensions together so you can assess a team's ability to work with agents alongside the standards its members share.

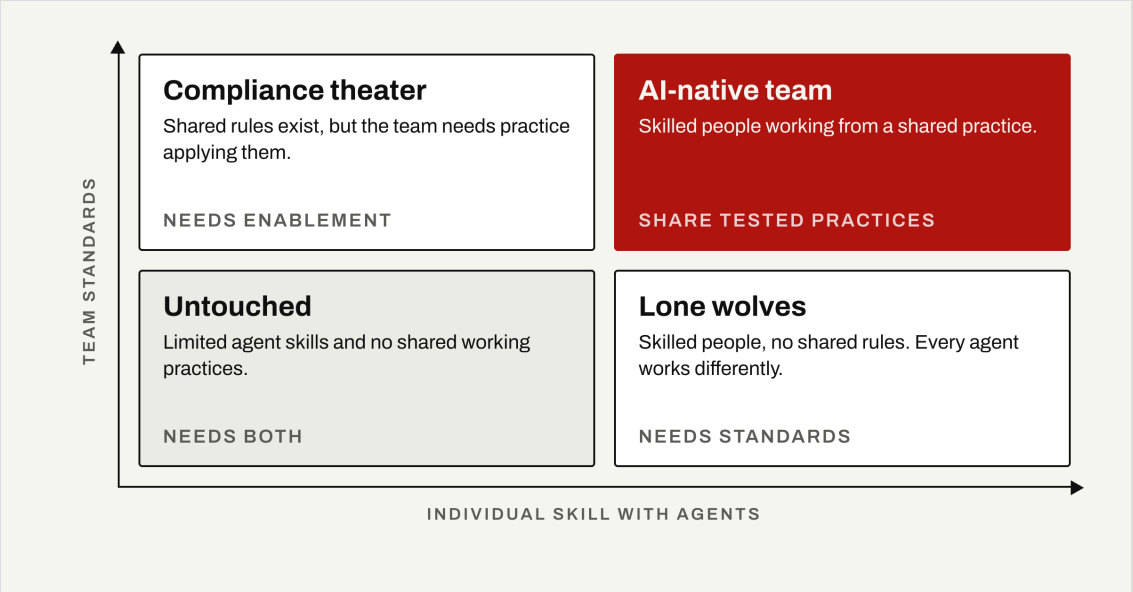


Figure 2. *Skill gaps need training and practice; system gaps need shared standards. Most teams start in the bottom-left corner.*

A team of lone wolves has capable people working without shared rules. It needs standards. In the compliance-theater corner, the team has rules on a wiki but hasn't learned how to follow them in real work. It needs training and practice. The book calls these a *system gap* and a *skill gap*. The distinction helps you fund the right work. Another training course won't create a missing team standard.

> TRY THIS ON MONDAY
>
> Pick a change shipped in the last two weeks where AI clearly sped up the coding. With the people involved, sketch a timeline from the first discussion of the need to the production release. Include product discussions, architecture and design decisions, internal agreements and approvals, implementation, code review, testing, and any DevOps work needed to deploy it. For each stage, record time spent working and time spent waiting, plus what or whom it was waiting for. Compare coding time with the full elapsed time. Choose the longest recurring wait and agree with its owner on one change to reduce it.

2 What AI-Native Engineering Actually Is

By AI-native engineering, I mean building software with agents as part of the team's everyday work. You can apply it to an ordinary product with a backlog and paying customers, whether or not the product contains AI. The practice concerns how you build it. Your choice of agent is part of that, but the tools change every few months, so your teams need working habits they can carry from one tool to the next.

In February 2025, Andrej Karpathy described *vibe coding*. You give the model a loose description, accept the code it produces, and paste errors back until it runs. That can be useful for a weekend prototype, when you want to try an idea quickly. Carry the same approach into production and you end up responsible for decisions nobody consciously made. The code may run while the team struggles to explain it, and the tests may confirm very little about whether it does the right thing.

Two Engineers, One Ticket

Chapter 1 of the book uses a payment example to show the difference. Two engineers receive a ticket asking them to limit each user to 10 payment requests per minute.

The first engineer sends the ticket straight to the agent. In under a minute, it produces code that counts requests by IP address, complete with passing tests. The change merges. Three days later, a corporate customer's whole office gets locked out because everyone shares one IP address. Meanwhile, a fraudster whose home IP keeps changing avoids the limit altogether.

The second engineer spends ten minutes writing down what the task means in this system. The limit applies to the logged-in user. A retry with the same idempotency key doesn't count again. The implementation must use the approved cache library and emit metrics in the format the existing dash-

boards expect. With that context, the agent builds an implementation the engineer can check against explicit requirements. The engineer finds a small gap, updates the notes, and runs the agent again before shipping.

Both engineers had the same tool and the same ticket. The second made the decisions the agent would otherwise have guessed at, and had a way to check whether it followed them. Your teams need time and shared practices for doing that preparation.

From Implementer to Orchestrator

Engineers have traditionally spent much of their time turning requirements into code. Agents can now produce a plausible first draft of routine code in minutes, so more of the engineer's attention goes to the decisions around it. Someone still has to understand what the system is for and explain constraints that exist only in people's heads. Someone also has to decide whether the result is safe to ship.

Figure 3 shows how intent, constraints, and verification inform one another throughout the work.

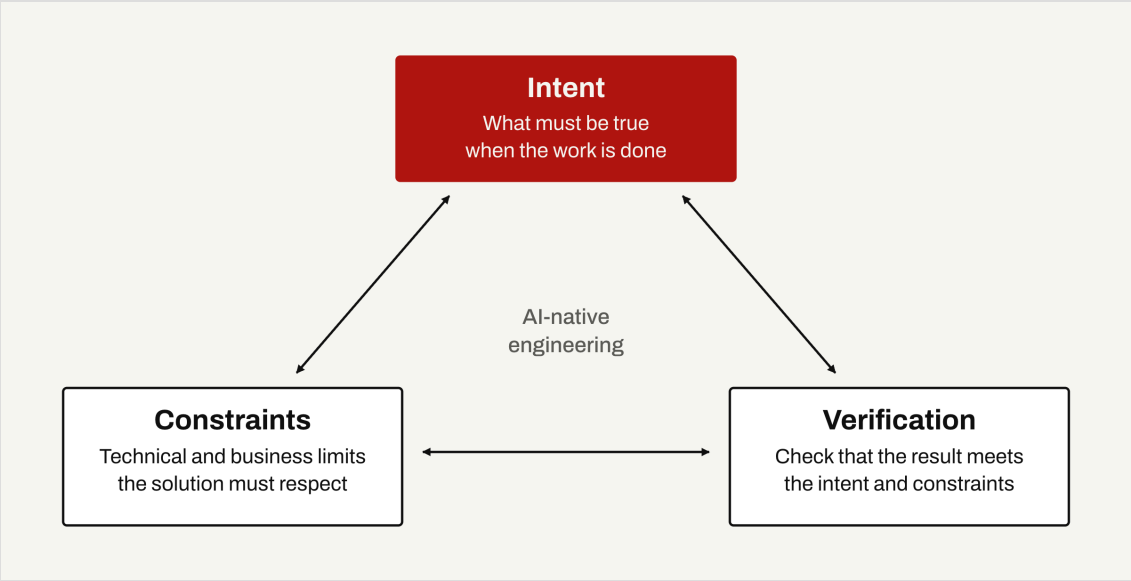


Figure 3. *Intent, constraints, and verification are connected in both directions. A finding in any one can lead the team to revise the others.*

Intent describes what must be true when the work is done. The engineer has to make it precise enough for an agent to follow. *Constraints* describe the technical and business boundaries the implementation must respect. An agent can easily break a rule that the team knows well but has never written down. *Verification* is how the team checks that the result meets the intent and stays within those boundaries, both before release and in production.

You'll recognize these as requirements, nonfunctional requirements, and quality assurance. They already belong to engineering. As implementation takes less time, they account for more of the work that determines whether a change is useful. A quick implementation won't save time if the team then spends hours correcting what it asked the agent to build.

An Agent Is a Model Plus a Harness

An agent is a model plus a harness, with the two layers described at the start of this guide. Your vendor supplies the *model* and the *agent harness* that manages its work. A competitor can buy the same tool. Your team's *user harness* adds the context, rules, skills, and verification checks specific to your system. That's where your knowledge of the codebase and the mistakes you need to prevent becomes part of how the agent works.

An OpenAI team reported building a product of about a million lines of code in five months, with Codex writing every line. Starting with three engineers and growing to seven, the team described the hard problems as "designing environments, feedback loops, and control systems." That's the work of building the harness. Your investment in that environment is something a competitor doesn't acquire by buying the same model.

> TRY THIS ON MONDAY
>
> Ask one of your strongest engineers to open your main repository and show you what the agent knows when it starts a task. Look for instructions explaining how to build, test, and run the system. If the explanation stops at "it reads the code," ask how it learns the team's rules and constraints.

3 What Your People Need to Learn

Your engineers will want to know which skills to invest in. An answer based on what AI can't do today won't help them plan very far ahead; those limits change every quarter. In the book, I describe eight skills that remain useful as the tools improve. Figure 4 shows them.



Figure 4. *The eight skills of an AI-native engineer. Spec literacy is the one the others build on.*

You probably already promote people for much of this work. Senior engineers and leads are expected to clarify requirements, judge risk, and think about how a system will change over the years. They also help product and engineering understand each other. Working with agents brings those responsibilities to engineers earlier in their careers. With good habits, someone two years into the job can take on work that used to require more experience. Seniority alone doesn't provide those habits; an experienced engineer can also generate more code without making the result more useful.

Seniors: The Craft Moves Up

In my experience, senior engineers have a lot to contribute here, even when they feel uneasy about the change. They have spent years developing the skills that an agent needs someone around it to exercise.

Design

: The senior engineer designs the environment the agent works in, including what it can see and what it is allowed to do, so the team can get reliable results across tasks.

Judgment

: Working out how to break an ambiguous problem into pieces that an agent can build and a person can check draws on years of engineering experience.

Verification

: Engineers use their knowledge of the system's difficult cases to establish whether the code is correct, even when nobody on the team wrote it by hand.

You don't need to turn every specialist into a generalist. A performance expert may contribute far more by owning the risky cases in their area than by spending the same time drafting product requirements.

Juniors: The Apprenticeship Changes

Agents now write much of the boilerplate that junior engineers used to learn by writing, and junior roles are harder to find than they were a few years ago. Your team still needs to develop its next generation of senior engineers. That requires a deliberate way to teach judgment when less of the learning comes from implementing routine code.

I recommend pairing on the agent, a practice also supported by GitHub's rollout guidance. A senior and a junior work in the same agent session. The junior sees why the senior accepts some output and rewrites other parts. Sometimes the senior discards the attempt and starts again. Those calls are difficult to teach in a document, but a junior can watch them happen and discuss them with someone who knows the system.

Expect a delivery dip while people pair heavily in the first weeks. Make it clear beforehand that you have planned for it and are paying for the learning. The investment can pay off quickly, but if you respond to the dip by demanding more immediate output, people will return to working alone and keep the gains to themselves.

What Learning Looks Like

The learning I've seen stick follows the same progression as the book, starting with how models and agents work and where they fail. People then learn context engineering and move into spec-driven development on real tasks, including how to verify the results. Use your own codebase throughout. The constraints that matter are the ones people will face when they return to work after the session. A one-hour prompting webinar can introduce the tools, but plan for months of practice. Section 7 describes the rough progression.

People also need help when they get stuck. Provide a support channel where someone can answer within minutes, and record community sessions so the learning remains available. Hackathons give engineers time to build something they care about with agents. Weekly pairing helps them bring what they learned back into everyday work.

> TRY THIS ON MONDAY
>
> Go through the eight skills with one team lead. For each skill, identify who could teach it and who needs help. Where nobody is confident, plan how the team will learn it. Put the learning time into the plan as funded hours per week.

4 What Has to Change in the System

As people learn to work with agents, they need a shared system for applying those skills. Otherwise, each engineer keeps their own way of working. Specs, shared context, and verification give the team a common basis for its work.

Specs Become the Shared Contract

A vague ticket leaves an agent room to make assumptions. It can choose a familiar pattern from its training data and keep building without making the assumption visible to you. By the time someone reviews the cleanly formatted result, hundreds of lines may depend on a decision the team never discussed.

In *spec-driven development* (SDD), you write down the goal, constraints, acceptance criteria, and scope before the agent implements the change. Your teams already do related work in product requirements documents and architecture decision records. SDD makes that preparation the first step of significant agent work. Keep the spec short enough to use and precise enough to check against the result.

Make this a conversation across roles. Product, design, engineering, QA, and sometimes security need to agree on the spec before implementation begins. A ten-minute discussion can expose a disagreement that would otherwise appear in code review, after an agent has written a thousand lines around the wrong assumption. The reviewed spec then gives every engineer and agent the same answer, which addresses silent divergence before it reaches the code.

That preparation has a straightforward cost argument. As I explain in the book, spending 20 minutes on a spec can prevent two hours of correcting an agent that built the wrong thing. The spec also stays in the repository, available to the next person or agent that changes the feature. Figure 5 shows the workflow and the points where a person approves the work.

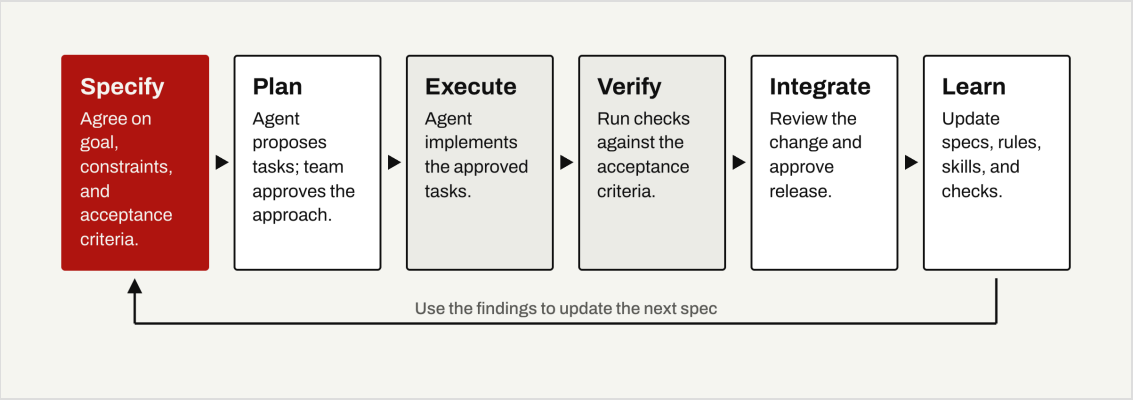


Figure 5. *Teams agree on the spec and plan before implementation. Verification and review determine how the change may proceed.*

Why Use a Spec-Driven Development Framework?

For a team adopting AI-native engineering for the first time, I recommend starting with an SDD framework. It gives everyone a standard process to follow, with agreed formats for specs, plans, and tasks, and a consistent place to keep them. The team uses the same commands and knows where approval is needed. An engineer can pick up a colleague's work without first learning how that person organized their agent session. Pairing and onboarding become easier because people are practicing the same workflow.

You could build those conventions yourself, but a framework gives the team a starting point it can use on real work immediately. Choose one whose defaults fit your team, try it on a real change, and agree on the parts you will use consistently. Keep the process light enough that people can follow it. Chapter 7 of the book compares the main options and their trade-offs. The benefit for you as a leader is having a shared practice you can support and improve as teams gain experience.

The Shared Context Is the Real Stack

Your coding tool needs context about how your team builds software. Much of that context can live in four shared artifacts.

Rules

Write down the nonnegotiables and make sure the agent loads them for every task. They give the whole team the same starting boundaries.

An AGENTS.md file in every repository

This is the agent's onboarding note. It explains how to build, test, and run the project, and where to find useful information. Keep it short and base it on what the team has learned from actual mistakes. More text doesn't necessarily help. Research published in 2026 found that instruction files generated by AI did not help agents solve more tasks and increased the cost per task by about 20%.

Shared skills and scripts

Package recurring tasks so a person or an agent can run them the same way. A useful procedure should be available to everyone who needs it.

An approved set of MCP servers

The Model Context Protocol (MCP) lets agents connect to systems such as tickets, monitoring, and code review. Agree on which integrations the team can use, so those connections have been reviewed by your security team.

Agree on the agent the team uses, the shared artifacts, the verification gates, and the spec workflow. Those choices affect everyone's work in the codebase. People can still use the editor they like and write prompts in their own style. Their agents need the same rules, and their changes need to pass the same checks.

Across several repositories, you also need a way to distribute the shared context. Copying rules into 40 repos creates 40 versions to maintain, and they will drift as people edit them. Give shared context the same treatment as shared code. Version it, pin each repository to a known version, and give someone responsibility for maintaining it.

Figure 6 shows how that environment improves over time. Even if you keep the same model from the first week to the third month, its results can improve because the team has changed what surrounds it. The rules answer questions that were previously left open. Shared skills capture useful procedures, and checks catch mistakes the team has already encountered. When people put those lessons back into the harness, the next task starts with better support.

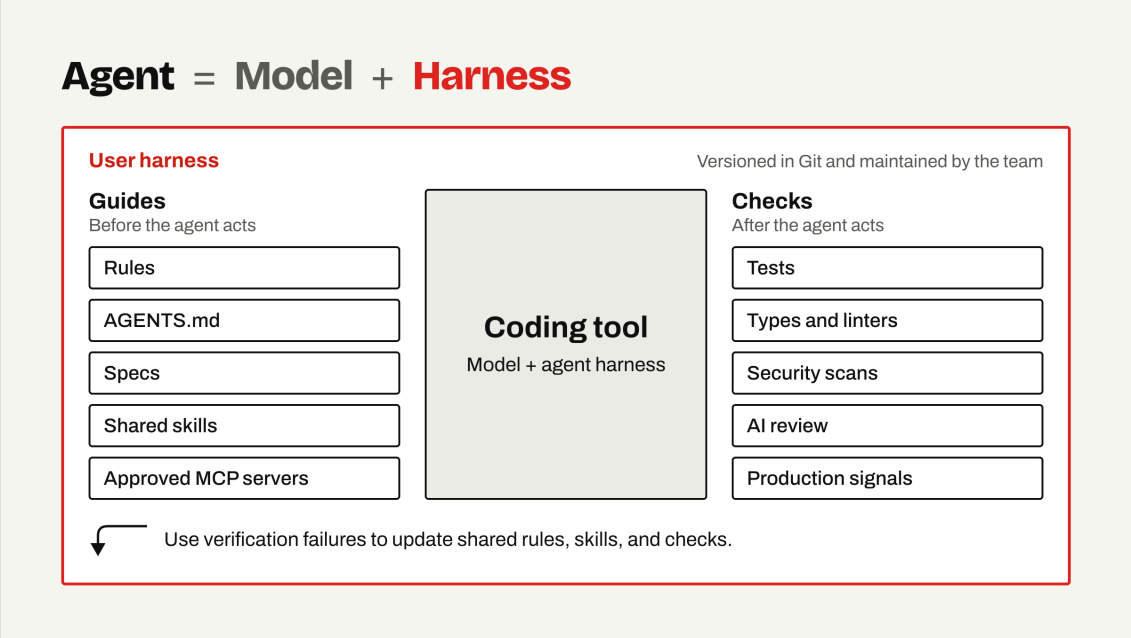


Figure 6. *The user harness is the team-maintained part of the full harness. It supplies context and checks around the model's work, alongside the agent harness that comes with the tool.*

Verification Has to Scale with Volume

Consider a team that moves from 5 pull requests a day to 25 after adopting agents. The same people still review them, with the same number of hours available. They can't give five times the work the same attention. If every change needs a full human review, either the queue grows or the reviews get

thinner. Asking reviewers to be more careful won't resolve that workload. The team needs a way to decide where that attention is necessary.

Figure 7 shows the layers of automated checking that take routine work off the reviewer's plate and help the team verify changes as volume grows.

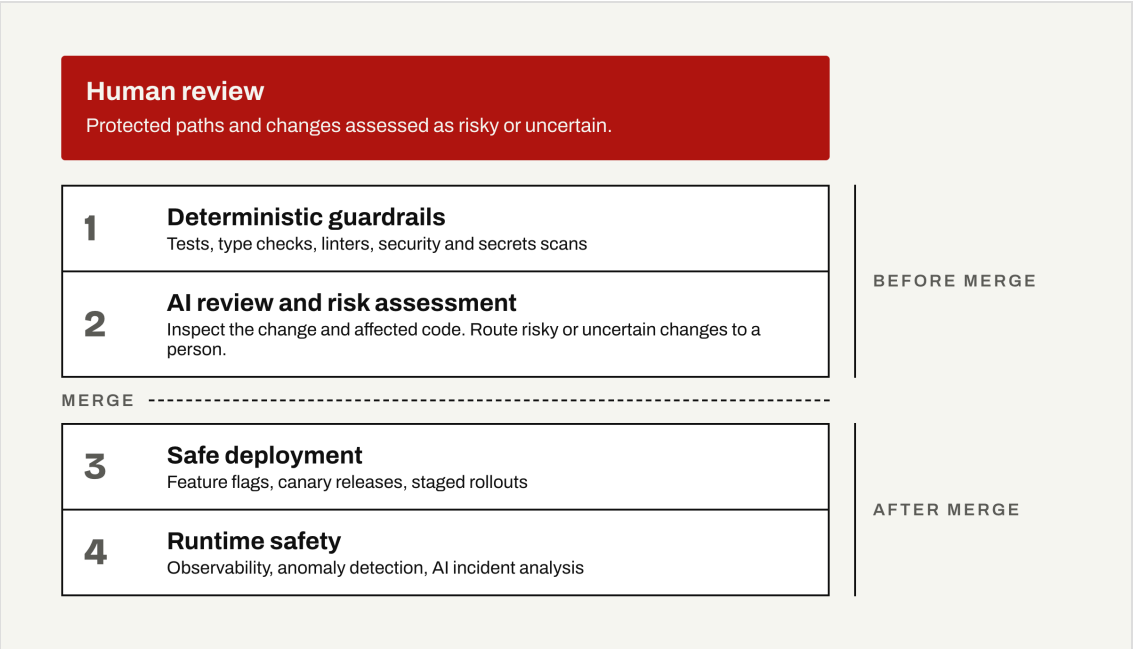


Figure 7. *Four layers automate verification, starting with the cheapest checks. People judge the changes and decisions that the layers route to them.*

Build these layers in order. Start with a solid test suite, strict type checks, a linter, and secrets scanning, all enforced as merge gates. The book puts that initial work at about a week for most teams.

Make an LLM-based reviewer part of your standard pipeline. Have it check the change against the spec and your team's conventions, including logic errors or weak tests that deterministic checks may miss. Run the review in a fresh session, ideally with a different model from the one that wrote the code. Its findings, alongside the test results, give the team evidence for deciding which changes need human attention.

Decide Which Changes Need a Person

Some more mature teams already allow selected changes through without someone reading every line. A small, reversible change can pass the automated checks and advance to a development or staging environment under an agreed policy. Production access is a separate decision. Small changes still need a risk assessment; a one-line change to authentication can matter more than a large documentation update.

The approach I describe in the book combines hard rules with an AI risk assessment. Protected paths always go to a person. For other changes, an AI reviewer inspects the change and affected code, estimates the risk, and explains its recommendation. The team sets the threshold for proceeding automatically. Uncertainty or a failed assessment sends the change to a person, and the AI cannot override a hard rule.

My Needs Human Review GitHub Action implements this approach. After continuous integration (CI) checks pass, it applies your hard rules and uses an AI reviewer to assess eligible changes. Start in shadow mode and compare its recommendations with your reviewers' decisions. Advisory mode adds labels and a status check while a person still merges. Autonomous mode can merge qualifying changes without waiting for a human review. Your deployment pipeline decides where a merged change goes; the Action doesn't grant permission to release it to production.

Software Factories

Some more advanced teams are moving toward *software factories*, where a written spec enters a pipeline of agents that plan, implement, test, and review the change. When nobody needs to steer the work in the middle, this is often called a *dark factory*. In the version I describe, people still set the goals and policy and review the resulting pull requests, with automation earned for narrowly defined changes. My software factory article maps the agents, isolated execution environments, quality gates, and review routing. The shared specs and verification in this guide are prerequisites for working that way.

> **TRY THIS ON MONDAY**
>
> Ask a team lead which changes wait longest for review. Pick a low-risk category and define the checks, risk threshold, and target environment it would need to proceed automatically. Compare the risk assessor's recommendations with human decisions in shadow mode before giving it authority to act.

5 Your Job: What Only a Leader Can Do

Your engineers can learn the practices and build the harness, but they can't allocate the organization's budget, change promotion criteria, or decide which risks the company is willing to take. They need you for that. In my experience, the following responsibilities have a large effect on whether adoption succeeds.

Name the north star and the bottleneck

Give the teams a concrete outcome and identify the bottleneck they need to remove. Include what must not get worse. A team can work toward shipping the renewal feature this quarter without raising the change-failure rate. A direction as broad as becoming AI-first leaves them to guess what you expect to improve.

Fund an owner for the shared stack

Give your platform team or a group of tech leads time to maintain the shared stack. Someone has to choose the default agent and approved tools, resolve disagreements about rules, and revisit those choices every few months. Put the work on their calendars. DORA's 2025 research found that 90% of organizations had adopted at least one internal platform, and that platform quality was associated with the value they got from AI.

Buy the learning time, out loud

Make pairing and practice part of the planned workload, and tell people you expect the early delivery dip. Revisit performance reviews and promotion criteria, too. Recognize people who develop deep AI-native engineering skills, improve the shared harness, and help colleagues use it well. Teaching needs to count alongside delivery. If someone spends an afternoon helping a team improve, that work should help their career. Give engineers a reason to go deep and share what they learn.

Set the risk boundaries

Agree on which changes always need a person and which may proceed on automated evidence. Payments, authentication, database migrations, infrastructure, and CI configuration are places to consider for mandatory review. Enforce those boundaries through protected paths. For other changes, define the risk threshold and the environments automation may reach, starting with lower environments. Set limits on the data tools can access, their permissions, and what an unattended agent may spend. Figure 8 shows how you can grant more authority as the team's results justify it.

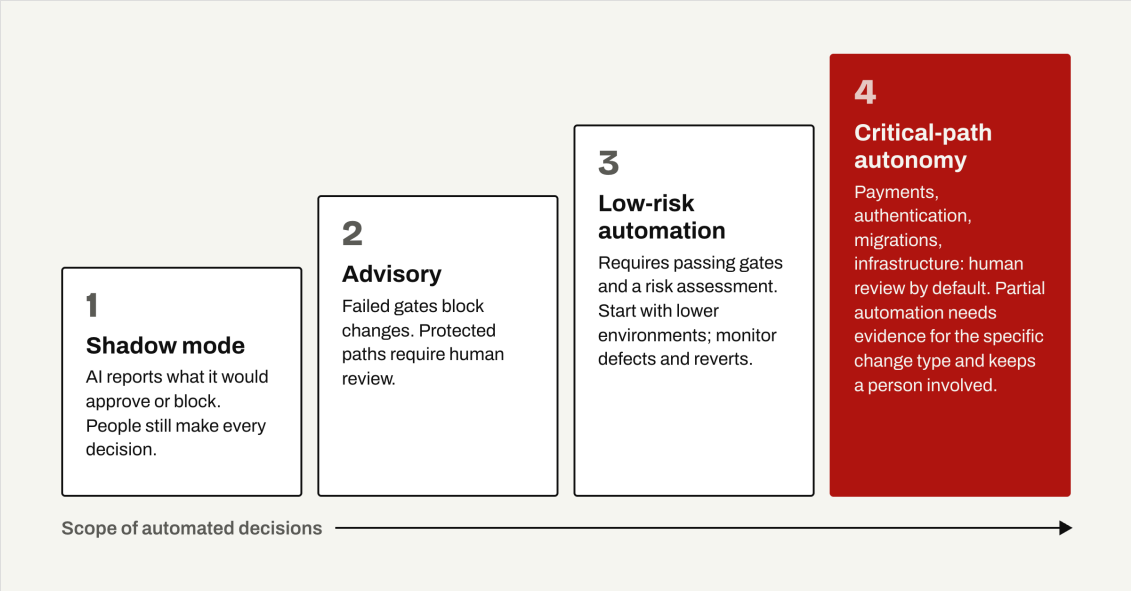


Figure 8. *Grant automation more authority when the preceding step provides enough evidence. Reduce that authority when the evidence no longer supports it.*

Bring product and design in

Include product and design in the change. Agents can help them prepare call transcripts, synthesize research, build prototypes, and draft specs or release notes. Decisions still need attention from the people accountable for them. A feature built in two days can spend five weeks behind a flag while people review copy, obtain legal approval, and decide on the launch. As engineering produces more work, those queues grow. Look at the approvals you own as well as the work you ask teams to speed up. Over time, invite support, operations, and other functions to explore their own uses of AI. Give them relevant training and approved tools, so the benefits extend across the organization and people can learn together.

Decide what the extra capacity is for

With heavy investment in skills and the harness, a team of four engineers can reach the output that used to take ten. That describes the top end of what teams achieve, and shouldn't become your assumption for every team. Even so, people will hear those numbers and worry that six jobs are about to go. You need to decide how to use the capacity. Cutting payroll gives you a one-time saving. Investing it in the product lets the same people raise quality, reduce risk, and address the debt and features they never had time for. I prefer that second option. Be clear about your choice early; people who expect their improvements to cost them their jobs have little reason to share them.

Hold the direction

Keep asking whether the team is building what the customer needs. When an agent makes implementation cheap, an engineer can produce an impressive feature nobody asked for or choose a complicated solution that used to take too long to consider. The maintenance work remains. Every feature has to survive migrations, incidents, and upgrades over the years, so assess roadmap items against the cost of owning them as well as the cost of the first implementation.

> **TRY THIS ON MONDAY**
>
> Ask your tech leads to propose a review policy: which changes always need a person, which low-risk changes could advance automatically, and to which environments. Check how the pipeline enforces those decisions and routes uncertain cases to a reviewer.

6 Resistance Is Rational

Some of the strongest engineers in the teams I work with are skeptical about using AI. Often they are the senior people whose support the team needs most. Before you conclude that they dislike change, listen to what they are worried about. Their objections usually fall into three groups, and each points to something you need to address.

"It's not good enough for my use case."

An engineer who knows a complex system well can see where an agent's familiar patterns conflict with its local rules. They have good reasons to care about that precision. In the 2025 Stack Overflow Developer Survey, experienced developers were the most distrustful group, with about one in five saying they highly distrusted AI tools' accuracy. Work with them on the specs, context, and gates described in Section 4, so they can test whether the output is trustworthy in their part of the system.

"I'll stop thinking and just prompt all day."

Skill atrophy is a fair concern. Catching a subtle mistake in an agent's output requires close reading and an understanding of the system. Those skills weaken if people stop practicing them. Protect deliberate practice, and make reviews a place where engineers explain their judgment about the change.

"I became an engineer to write code."

For some people, writing code by hand is the part of the job they enjoy most. Losing that work affects their identity, and being asked less often for help can affect their status in the team. You won't resolve that by pretending the job is unchanged. Acknowledge the loss, including your own if you feel it, and help them see the work that needs their experience now. Design, judgment, and verification have always been difficult parts of engineering. They take up more of the job as agents handle implementation.

Telling someone "just use it, you'll get used to it" dismisses the concern they brought you. You risk losing good engineers that way. Give them room to try the practices on real work, with specs and gates, and judge the result. A skeptical engineer who changes their mind after seeing evidence can become a particularly credible teacher for the rest of the team.

The Cost of Running a Fleet

You also need to look after the engineers who take to agents quickly. Steering three or four streams of work at once means reviewing each one and switching attention whenever a notification arrives. People describe reaching a point after a few hours where their head melts. At the same time, shipping in a day what used to take ten feels good enough to make them take on even more. That combination can lead to burnout.

Set expectations for a pace people can sustain. I can steer about three parallel streams well in my own work; the team in the book's closing story uses a similar cap. Choose the work carefully, too. Closely related streams share context, and a largely independent task can run in the background. Two tangled features demand too much switching. If someone is producing far more than the rest of the team, ask them to spend part of the week teaching. Passing on their judgment helps the whole team benefit without making constant high output their only contribution.

> **TRY THIS ON MONDAY**
>
> Sit down with a strong engineer who is skeptical about AI. Find out which of these concerns they have, and ask what they would need to trust an agent's output in their part of the system. Give the work needed to answer that concern a budget.

7 A Rollout That Doesn't Break People

The transition needs leadership and engineers working together. A management announcement leaves people without the skills and shared practices to act on it. Enthusiastic engineers can show useful results, but they cannot fund everyone's learning time, change incentives, or set policy across the company. Both groups need to take responsibility for the change.

As a leader, help people see what is possible. Bring the people who share the work into the same room, including product, design, and operations. Ask experienced engineers to demonstrate a real task with agents, from the first requirements through verification. Bring in trainers for workshops when the

team needs that help, and use the team's own work so people can connect the practices to their jobs. Alongside that support, you need engineering champions applying the practices every day and showing colleagues what works, including where an attempt failed and what they changed.

Start with a few teams before expanding. That gives leadership and the champions time to resolve problems together and build a practice others can use. Make room for people who need more time to learn as well as the early enthusiasts; the goal is to bring the organization with you.

Assess Before You Scale

Before choosing pilot teams, assess the organization, the way each team works, and the codebases they use. Each assessment answers a different question.

Where does the organization sit?

A maturity model, such as the one from Carnegie Mellon's Software Engineering Institute and Accenture, helps you describe the organization's position to the board. You'll still need to work out what that position means for the teams' next steps.

How does the team actually work?

Complete the team assessment together, answering for what you do today. It shows which practices are in place, where the gaps are, and which questions nobody can answer yet. A repository can contain tests and a spec template without showing whether people use them consistently or agree on what they can delegate. The assessment brings those questions into the team conversation.

What has each team decided, and what not?

I built the AI-Native Team Canvas, covered in Chapter 10, to help teams make these decisions together. Its eight boxes cover the adoption questions on one page, including the north star and how to measure progress. Allow 60 to 90 minutes for a first session. Ask the team to describe what it already does. Leave the gaps empty, so you can see which decisions still need work.

Can agents work safely in this codebase?

Give the readiness analyzer to a coding agent and ask it to audit your repository. It checks the evidence in the codebase: instructions and specs, tests and merge gates, tooling, and safety controls. Use it alongside the team assessment. The analyzer inspects what is in the repo; the assessment exposes working habits and disagreements an agent cannot establish by reading files. Compare both sets of gaps before deciding what to fix.

Pilot, Reflect, Expand

Choose a few pilot teams with people who are ready to champion the work. Give them real delivery tasks, so you learn how the practices hold up in your systems. Track what blocks them and what changes in the numbers. Listen to the objections as well. Use what you learn to improve the tooling, process, and learning path before bringing in the next wave of teams. Figure 9 shows how those cycles build toward adoption across the organization.

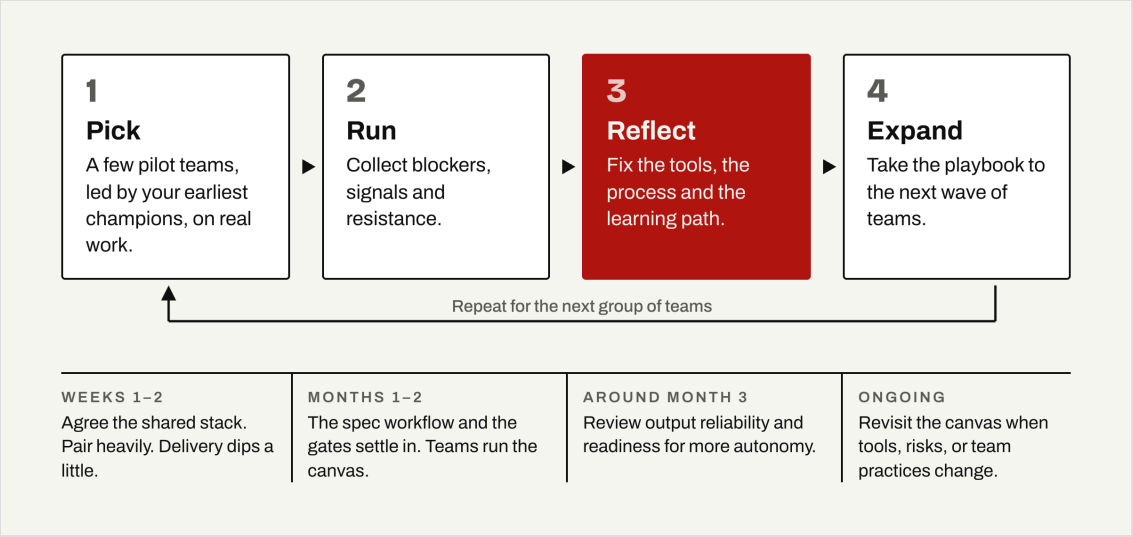


Figure 9. *Each pilot informs the next wave of adoption. The timeline shows an approximate progression; the team's starting point affects the pace.*

The timeline is a rough guide. Your codebase and the team's starting point affect the pace. Expect the first few weeks to be slower while people learn and set up the environment. The workflow then settles, and around three months in, teams often find that output has become noticeably more reliable. Use the team's evidence to decide when to expand.

Let Practice Travel Through People

Once a pilot team finds something useful, other teams need a way to learn it and a way to test whether it works for them.

Set up a working group with engineers from every team, tech leads, product and design, and the platform team responsible for the shared stack. They compare experiences and decide which lessons should become shared practices. Give champions time to run demonstrations and workshops, pair with colleagues, and bring recurring blockers to the leaders who can resolve them. As other functions start using AI, include their people in these sessions too.

Keep an official path for experimentation alongside the standard. AI tools change in weeks, and engineers will find things worth trying. If the standard gives them no way to test those tools, they are likely to work around it quietly. Give them permission, time, and a small budget to experiment away from production-critical paths. Ask them to share the results. The wider team should test a proposed replacement and agree on it before it becomes the default.

> **TRY THIS ON MONDAY**
>
> Choose two pilot teams for next quarter. Record who will champion the work and what each team aims to improve. Set a date for sharing what they learn with the rest of the organization.

8 Measure Practices, Not Seats

You'll need to show whether the investment is working. Be careful about which numbers you put in front of the board. Some are easy to collect but tell you very little about whether the team is delivering better software.

License counts and activated seats describe the tools you bought. Lines of AI-written code are also a poor target. Faros AI's 2026 analysis of telemetry from 22,000 developers found that teams with high AI adoption had pull requests about 50% larger, with more bugs and incidents per pull request. Suggestion acceptance rates don't establish whether the accepted code was good. Self-reported speed needs care, too; people can feel much faster while the delivery numbers barely move.

Figure 10 organizes the more useful signals into three layers.

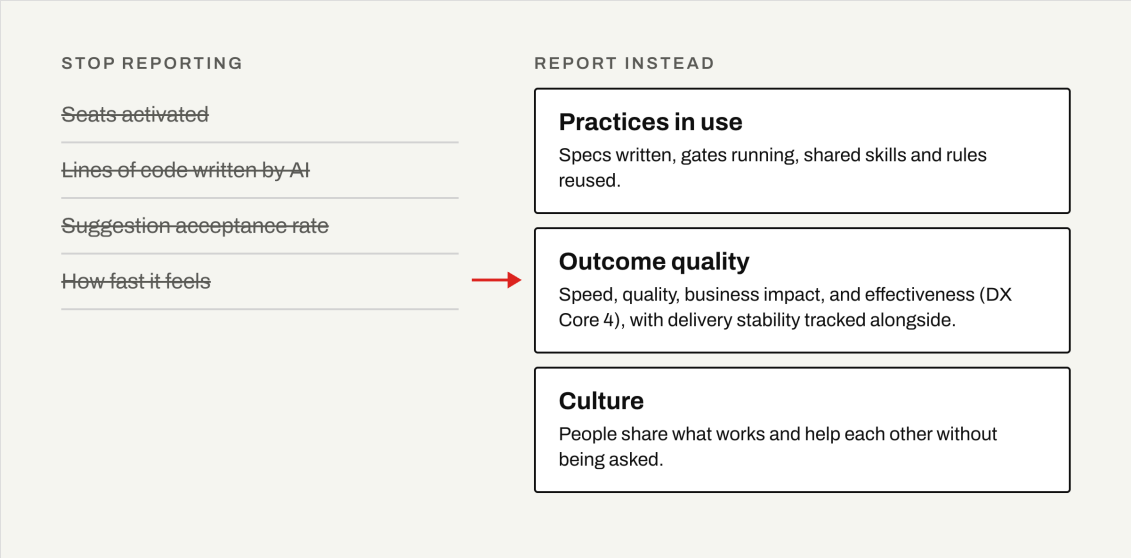


Figure 10. *Assess adoption through working practices, outcome quality, and culture. Tool-usage counts don't establish whether delivery improved.*

Practices in use

Look at whether teams write specs and use the verification gates. Check whether shared skills and rules are being reused in everyday work. These signals help you distinguish a change in working habits from people simply opening the tool.

Outcome quality

Use a framework that covers several dimensions of the result. DX Core 4 brings together the DORA, SPACE, and DevEx research across speed, effectiveness, quality, and business impact. Pay particular attention to stability. DORA's 2025 research links AI adoption to lower delivery stability unless strong tests and safety nets are in place, so report that alongside any gain in speed.

Culture

Notice whether engineers share useful discoveries without being asked and help one another when an agent gets stuck. This is harder to measure than cycle time, but it tells you whether people have made the practices part of how they work together.

Track the product side as well, since faster engineering puts pressure on the decisions around it. *Time to first evidence* measures how long it takes to turn a question about customers into an answer you can defend. *Decision latency* measures the wait between having enough evidence and making the decision.

> **TRY THIS ON MONDAY**
>
> Replace a vanity metric in your AI report with cycle time, change-failure rate, or reuse of shared specs and rules. Track it for a month, then check whether it helped you make a decision. If it didn't, choose a signal that would.

9 Your Monday List

Use this checklist to plan the work with your leads over the next month.

- [ ] Trace a recent change from the first product discussion to production, recording decisions, handoffs, and waits.
- [ ] Agree on the outcome you want, the bottleneck to remove, and what must not get worse.
- [ ] Give one person ownership of the shared stack and time on their calendar to maintain it.
- [ ] Have two pilot teams work through the AI-Native Team Canvas.
- [ ] Complete the team assessment with your pilot teams and compare it with readiness audits of your three most important repositories.
- [ ] Define mandatory human review and low-risk automation policies, including which environments automation may reach.
- [ ] Enforce tests, type checks, and secrets scans as merge gates; evaluate risk-based review routing in shadow mode.
- [ ] Fund learning time, workshops, champion support, and a small experimentation budget; recognize deep expertise and teaching in performance reviews.
- [ ] Put in writing how you intend to use the extra capacity.
- [ ] Replace one vanity metric with a signal that helps you make decisions.
- [ ] Make time to listen to your most skeptical senior engineer.

I would start with the third item. Someone needs the responsibility and time to maintain the shared stack for the rest of this work to hold together. In my experience, leaders too often assume that someone will pick it up alongside their existing job.

If you have a question or want to exchange ideas about AI-native engineering, I'd be happy to chat. You can reach me at info@alfonsograziano.it.

About the Author



Alfonso Graziano works as an AI Solutions Lead at Nearform, helping engineering organizations adopt AI-native practices and building AI agents for production. He has worked as a professional software engineer since 2020 and continues to write production code every week.

He wrote *AI-Native Software Engineering: Building with AI Agents and Spec-Driven Development* (O'Reilly). He has given more than 20 conference talks across Europe and built an MCP server for Node.js that has been downloaded over 100,000 times on Docker Hub.

alfonsograziano.it · ainativesoftware.engineering

Free Tools to Go Further

You can find these free tools on the book's website.

AI-Native Team Canvas

ainativesoftware.engineering/canvas

Use the eight-box canvas from Section 7 to work through adoption decisions with a pilot team. You can print it for the session.

How AI-native is your team?

ainativesoftware.engineering/assessment

Complete this with your team to identify practices, gaps, and unanswered questions. Compare the results with the repository readiness audit. No email needed.

AINE Readiness Analyzer

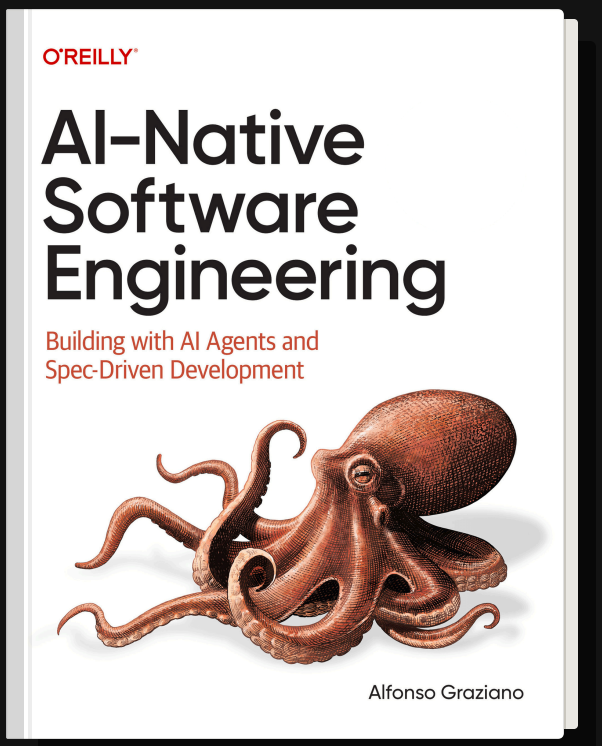
ainativesoftware.engineering/readiness

Ask a coding agent to audit your repo with this free skill. It scores the evidence and prioritizes fixes. Use it alongside the team assessment.

The 7-day roadmap

ainativesoftware.engineering/roadmap

Help your engineers get started with one topic a day, including theory, practice, and selected videos and articles. Free, no email needed.



READ THE BOOK

AI-Native Software Engineering

Building with AI Agents and Spec-Driven Development

This guide is the leader's view of AI-native engineering. The book takes your engineers through the practices in detail, including the trade-offs and exercises they can work through in your own codebase.

Chapters 1–2. The changing engineer's role, and how models and agents work.

Chapters 3–4. Context engineering and the Model Context Protocol.

Chapters 5–8. Writing specs, following the spec-driven workflow, choosing a framework, and verifying the result.

Chapters 9–10. Working with several agents and bringing the practices to a whole team.

Chapter 11. A story following one team as it applies the practices under a delivery deadline.

Start reading

learning.oreilly.com/library/view/ai-native-software-engineering

New essays and tools every few weeks: **ainativesoftware.engineering/newsletter**