AI in software: Scaling from activity to impact

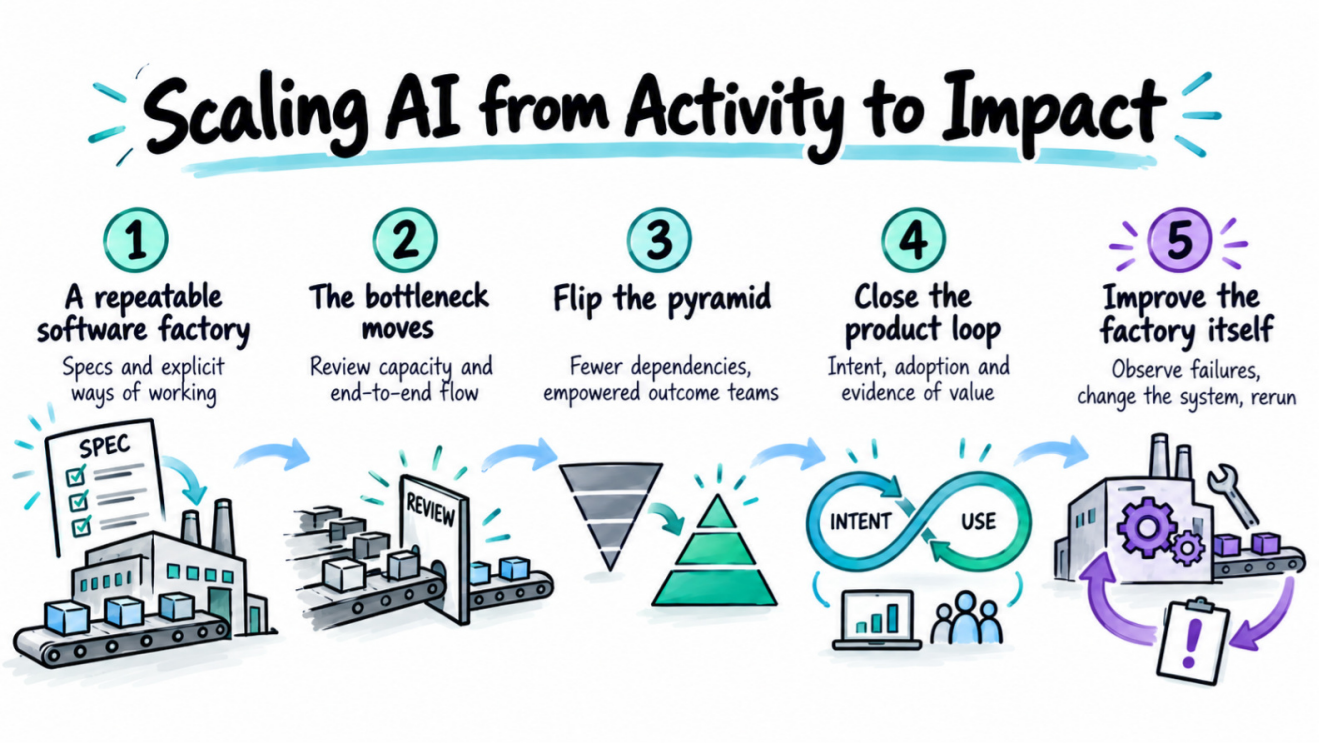
Yuval Yeret · Whitepaper · Revised November 7, 2026

AI changes what is possible to build, who can build it, and how we develop our products. Increased capacity in product and engineering creates room to rethink the product experience for both people and agents.

More activity and output do not automatically create impact. AI lets more people build, makes more ideas worth exploring, and makes work possible that used to require scarce engineering capacity. Faster coding still has to become effective delivery. Delivery has to turn into adoption and useful changes in customer behavior.

That takes work at three levels: the product, the software factory, and the company around them. The engineering loop has to become repeatable and trustable. People and agents need the structure and context to own useful work end-to-end. The product loop has to connect intent, discovery, delivery, adoption, and impact.

We also need to improve the factory itself. Eventually, the same engineering and product thinking can help us improve how the wider company works.



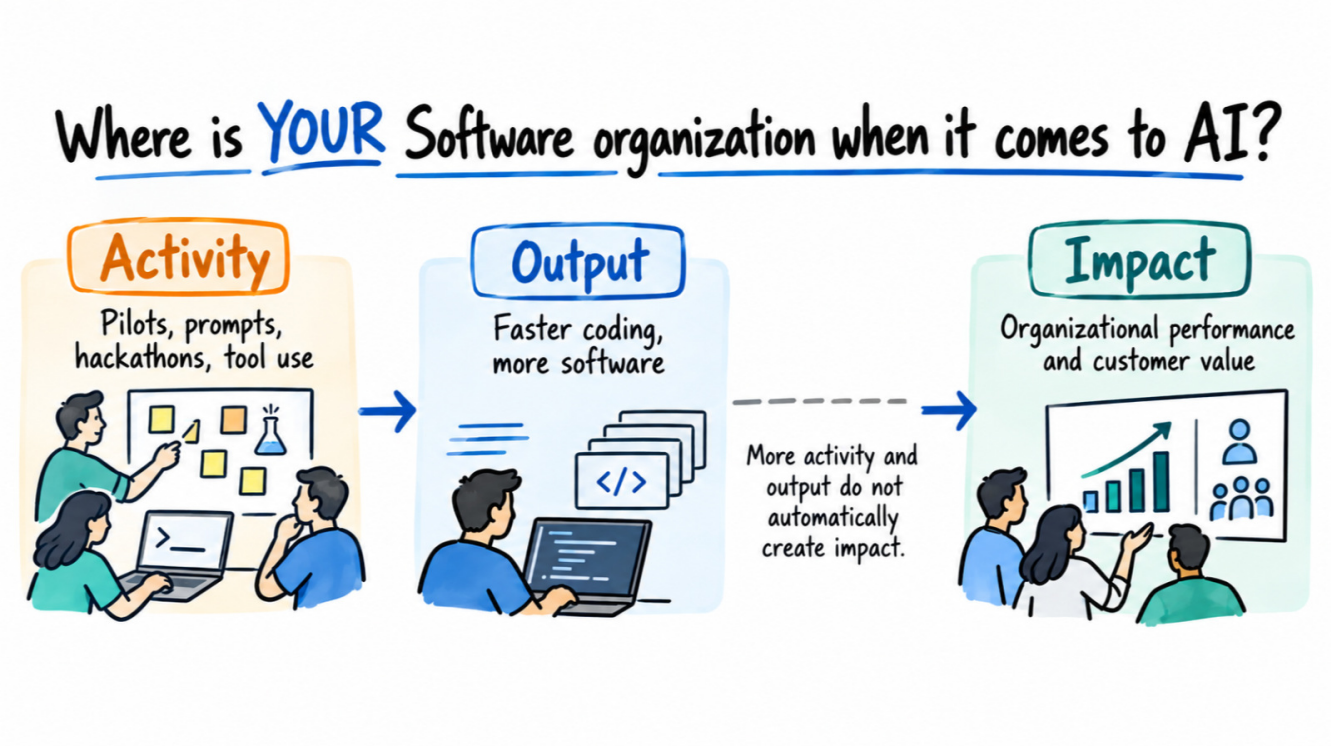
The full arc: a repeatable software factory, changing constraints, empowered teams, the product loop, and improvement of the factory itself

1. What counts as AI impact?

Most software organizations already have AI activity: pilots, prompts, hackathons, tools, and people trying things. Many also have local output. Developers can code faster and generate more software.

The next question is whether that local output becomes end-to-end throughput. Can the organization actually deliver more useful features to market? And beyond that, does the additional throughput create more value?

Those are different questions. A developer generating code faster doesn't establish that a feature reaches customers sooner. A feature reaching customers sooner doesn't establish that they use it or that it improves an outcome.



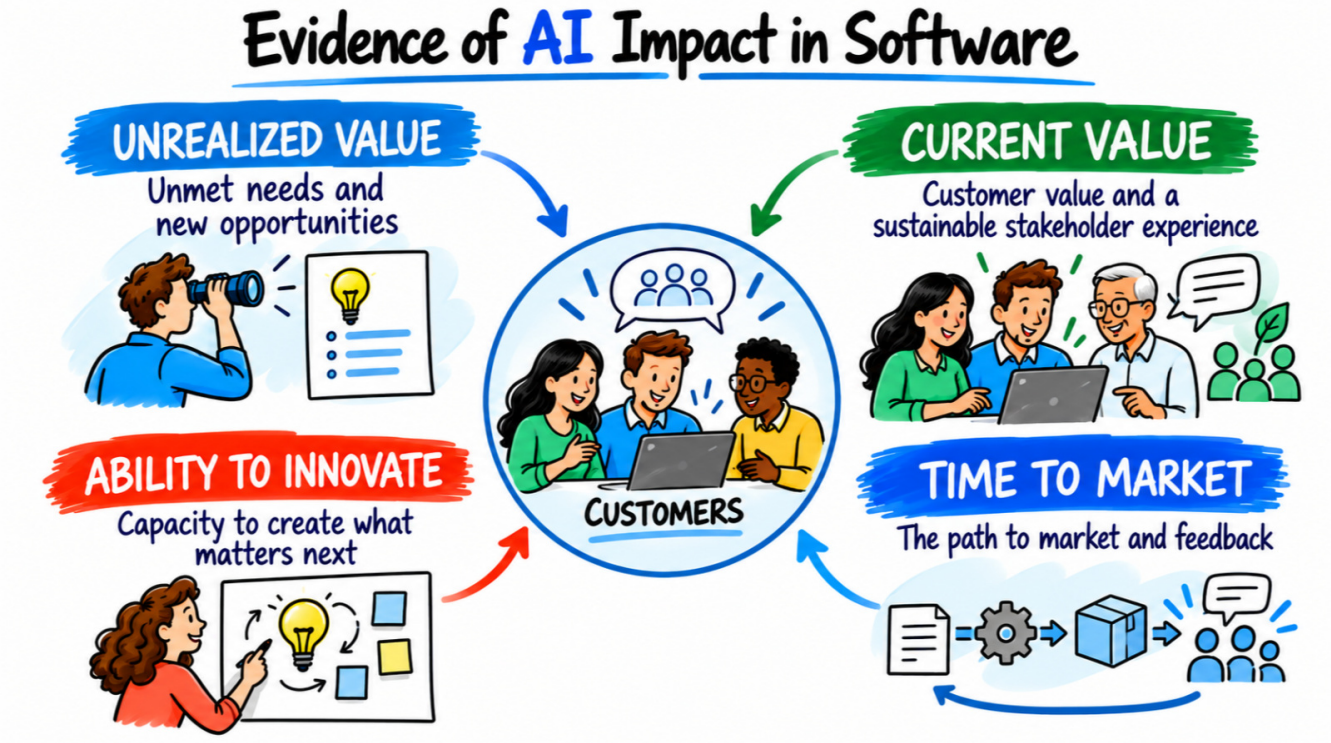
Activity, output, and impact are different measures of progress

I find the four value areas from Evidence-Based Management useful here. They predate AI, and they still help us ask what an organization is capable of doing.

Value area	The question for a software organization
Ability to innovate	Do we have capacity to create what matters next, beyond maintaining the product and keeping the machine running?
Time to market	How quickly can we deliver something and get feedback, so that we can sense and respond?
Current value	Are customers getting value from the product? Is the experience sustainable for customers, employees, and other stakeholders?
Unrealized value	Can we address unmet needs, serve new customers, or find new ways to deliver value?

AI can help in each area. It can increase the capacity available for innovation or shorten the path to market. It can become part of the product experience. It can make a new way of serving customers possible.

The value areas give us a way to examine those possibilities without treating AI adoption as the goal. When AI has an impact, we should expect to see something change in at least some of these areas.



The four value areas: ability to innovate, time to market, current value, and unrealized value

Time to market deserves care. We want to shorten the time to outcomes, but we shouldn't assume we delivered value just because we delivered software. Looking at time to market alongside evidence of current value keeps both questions visible: how quickly did we get something out, and what happened when people used it?

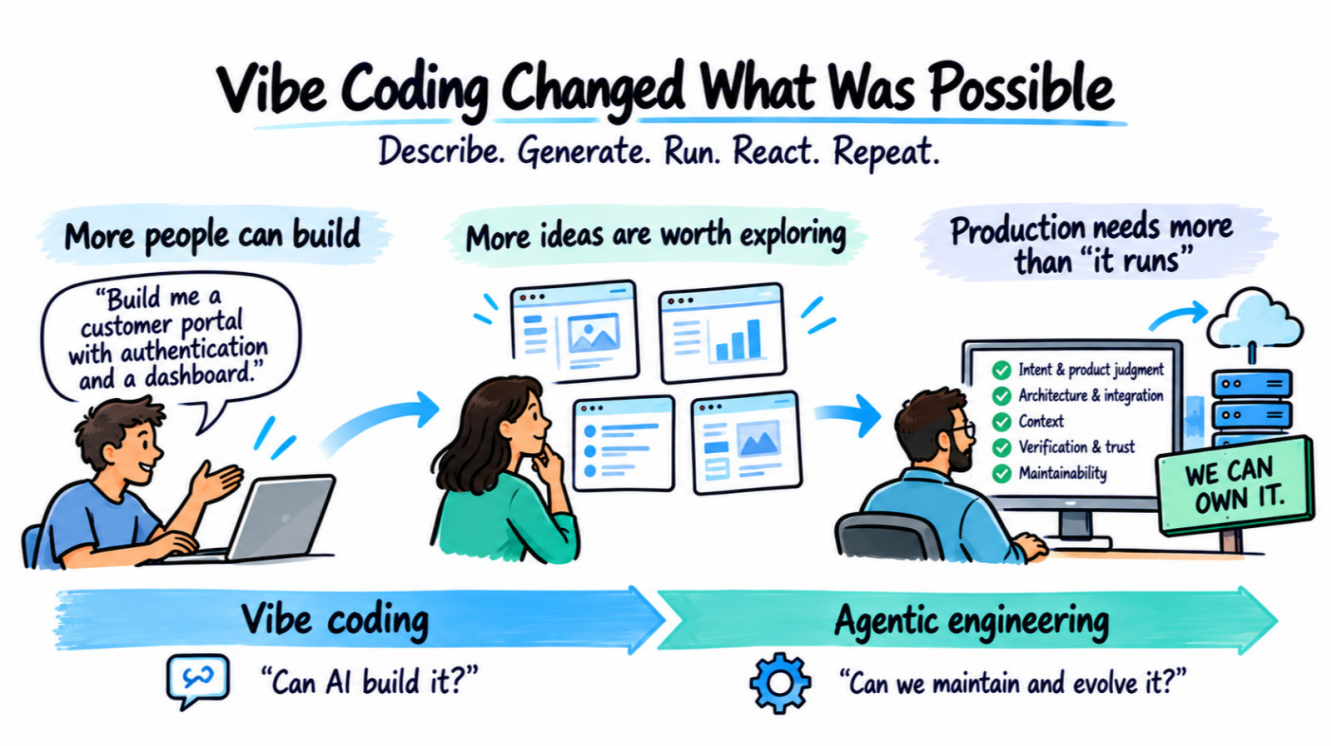
More capacity to build doesn't tell us whether we're building something useful.

2. Vibe coding changed what was possible

Vibe coding opened up software development to people who couldn't previously turn an idea into something they could try. Describe what you want, generate it, run it, react, and repeat. Product managers and engineers can get an idea onto the same page much faster when they have something concrete to explore.

That is useful. A prototype can help people understand one another and discover what they actually want. More ideas become cheap enough to investigate.

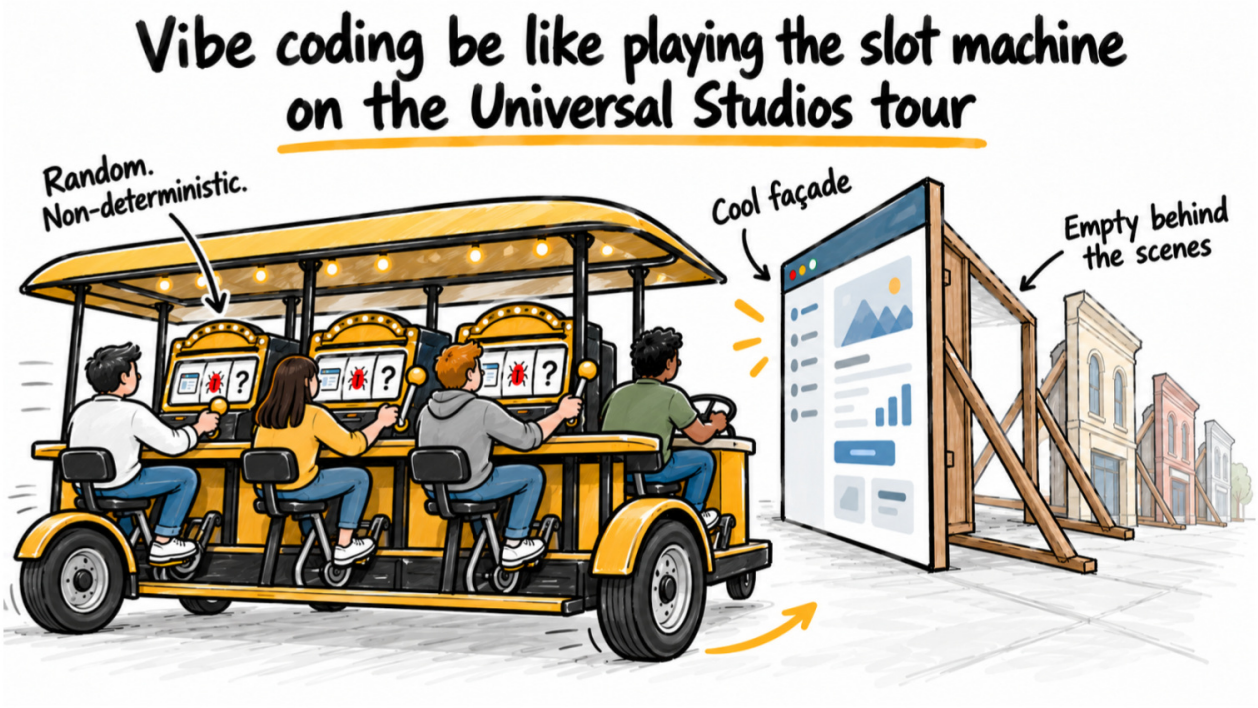
But a convincing prototype and a product we can maintain and evolve are different things. Production brings questions about architecture, integration, context, verification, and ownership. Existing products bring the additional reality of a brownfield system that people already depend on.



Vibe coding expands exploration; agentic engineering has to support maintenance and evolution

My experience of early vibe coding was often a combination of Vegas and the Universal Studios tour. You pull the slot machine and sometimes get something impressive. Sometimes you get stuck in a corner and can't work out what happened or how to get out. And even when the result looks good, it can be a cool facade with very little behind it.

That combination makes it hard to trust the work. We can explore quickly, but we don't necessarily know how the software was built, which decisions were made, or whether it follows the engineering discipline we need.



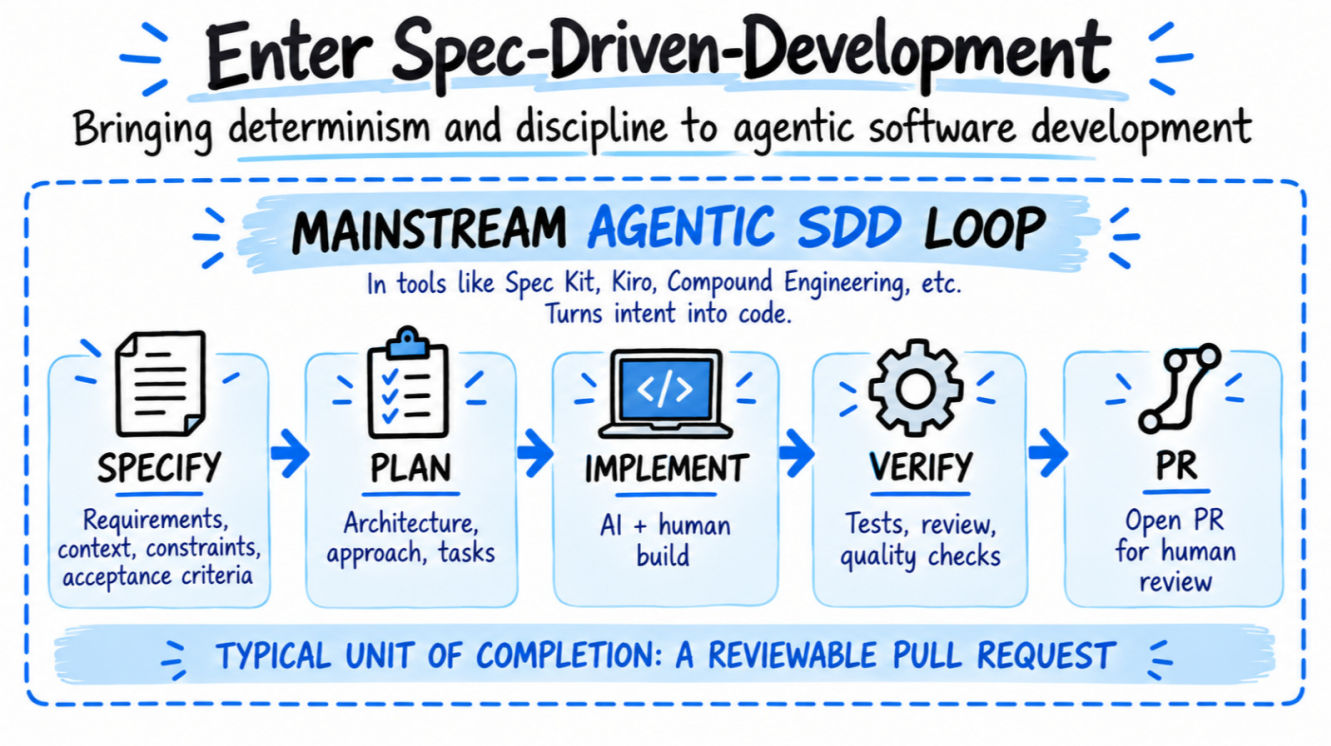
The slot machine and the Universal Studios facade: impressive results without a dependable system behind them

Specifying the engineering loop

Spec-driven development is one response. It makes the work more explicit: specify, plan, implement, verify, and produce a pull request. Requirements, context, constraints, and acceptance criteria guide the implementation. Architecture and tasks are considered before the agent starts building. Tests and review check the result.

Spec Kit, Kiro, OpenSpec, and Compound Engineering were among the spec-driven frameworks in use in October 2026. Their implementations differ, but they share an ambition to bring discipline and repeatability to agentic software development.

The model remains non-deterministic. What we can define more deliberately is the process around it: the inputs it receives, the boundaries it works within, the checks it must pass, and what happens when a check fails.

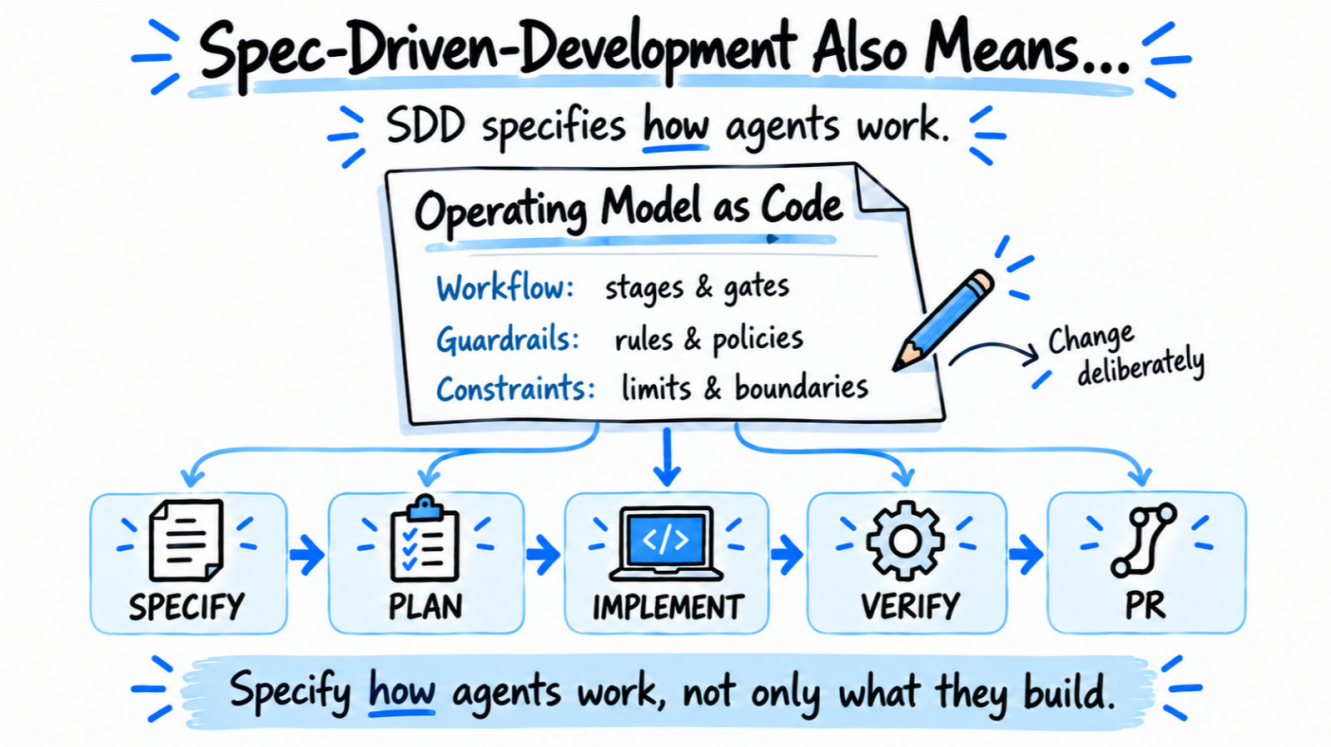


The mainstream spec-driven loop: specify, plan, implement, verify, and a reviewable pull request

For me, the most interesting part of spec-driven development is that it specifies **how we work here**.

It specifies the factory. What are the stages? What needs to happen at each stage? What are the gates, rules, policies, and constraints? What does good work look like in this organization?

That is our operating model as code. Often the code is Markdown, but it still defines behavior. Both humans and agents work within that definition.



The operating model becomes explicit: workflow, guardrails, constraints, and deliberate changes

We used definitions of done, definitions of ready, and definitions of workflow for similar reasons before AI. They gave a team something concrete to inspect and adapt. Giving agents autonomy makes the need for that clarity more immediate. Things that people could previously infer from experience now have to be made explicit enough for the system to use.

It also makes the operating model version-controlled. We can see what changed. If we add a test or relax a policy to enable exploration, that can be a deliberate experiment with a reason behind it. We can examine whether the change had the effect we expected.

Walls of Markdown aren't the goal

Making work explicit creates another problem if it produces documents that nobody reads.

One team described a familiar progression. Engineers and product people started with prototypes. They then introduced specifications, including functional and technical requirements, before development. Soon there were so many Markdown files that it became doubtful whether people understood what they were approving. They were using AI to read what AI had written.

The team began trying a web-based presentation of the material to make it easier for humans to digest. The useful part was the willingness to notice a problem in the experience and change the process.

It's about the developer experience and the product experience: the experience for the humans using this system. We should be thinking about the factory as a product that needs to evolve and find product-market fit. We should apply product thinking to the way we work.

The human contribution also has to include judgment. A leader described being asked for a second opinion on generated code before the person asking had formed an opinion of their own: “I'll give you a second opinion when I get the first opinion.” Passing along an agent's answer doesn't establish that somebody has understood it.

Nobody really knows the right way to do all of this. It's evolving by the week, by the day. Teams need space to explore and develop ways of working together.

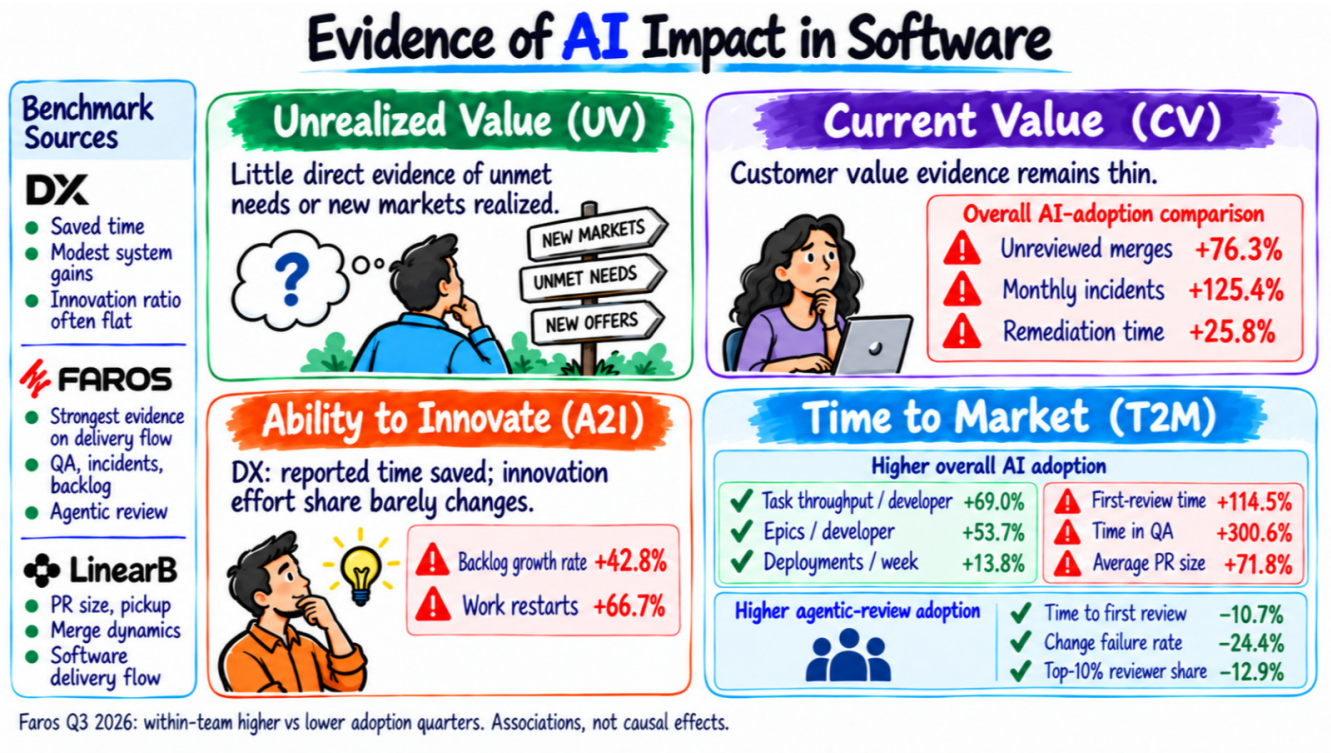
3. What the evidence tells us, and what it leaves open

The October 2026 benchmark reports give us evidence about delivery, quality, developer experience, and effort allocation. They give us much less direct evidence about customer adoption, changed behavior, or unmet needs being addressed.

Faros, DX, and LinearB looked at different populations and measures. Their findings shouldn't be combined into a single causal estimate of AI's effect. They do help us see where local gains coexist with pressure elsewhere in the organization.

In the Faros Q3 2026 report, higher overall AI adoption was associated with more completed work: task throughput per developer was 69% higher, and epics completed per developer were 53.7% higher. The same comparison also showed longer waits for first review, more time in QA, larger pull requests, and more incidents.

Higher overall AI adoption and higher agentic-review adoption are **separate comparisons**. The favorable first-review and change-failure results belong to the agentic-review comparison.[1]



The benchmark view separates overall AI adoption from agentic-review adoption

For higher overall AI adoption, median time to first review was 114.5% higher and time in QA was 300.6% higher. Average pull request size was 71.8% higher. Monthly incidents were 125.4% higher, and remediation time was 25.8% higher.

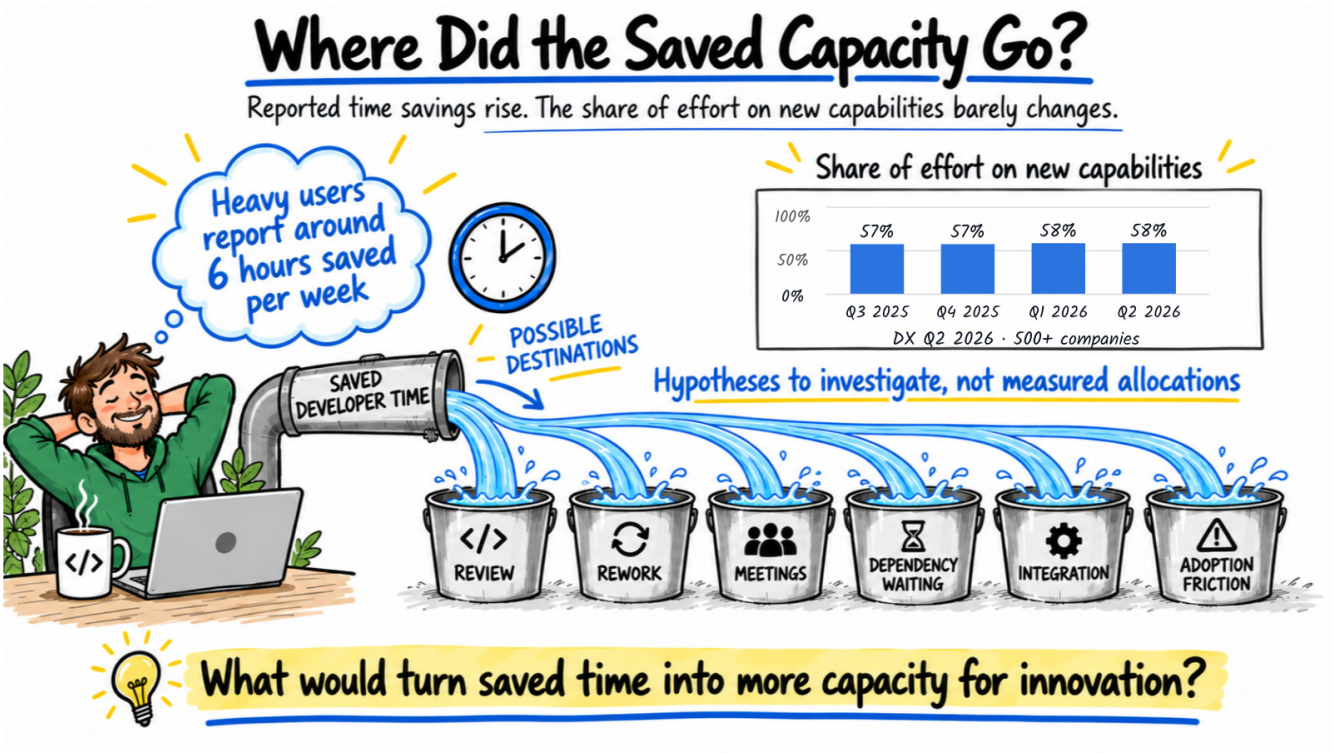
These are observational associations. They don't establish that AI, by itself, caused each change. They also don't measure customer value directly. They do make it hard to treat faster local production as sufficient evidence of a healthier software organization.

In the separate agentic-review comparison, higher adoption was associated with a 10.7% shorter time to first review and a 24.4% lower severe change failure rate.

Where did the saved capacity go?

DX reported growing time savings, with heavy users reporting around six hours saved per week. Yet the median share of effort spent on new features and capabilities barely changed: 57%, 57%, 58%, and 58% from Q3 2025 through Q2 2026.

That measure is an allocation of effort. An unchanged share doesn't mean the absolute amount of innovation, its quality, or its customer value stayed unchanged. More work could be produced within the same allocation.



Reported time savings and an almost unchanged innovation effort share raise a capacity-allocation question

Review, rework, meetings, dependency waiting, integration, and adoption friction are places to investigate. The buckets in the visual are hypotheses, not measured destinations for the same developers' saved hours.

In many organizations, coding wasn't the main constraint to begin with. Developers were already spending much of their time getting input, obtaining approval, coordinating, waiting for other teams, and integrating changes. Giving those developers a coding agent doesn't automatically change any of those conditions.

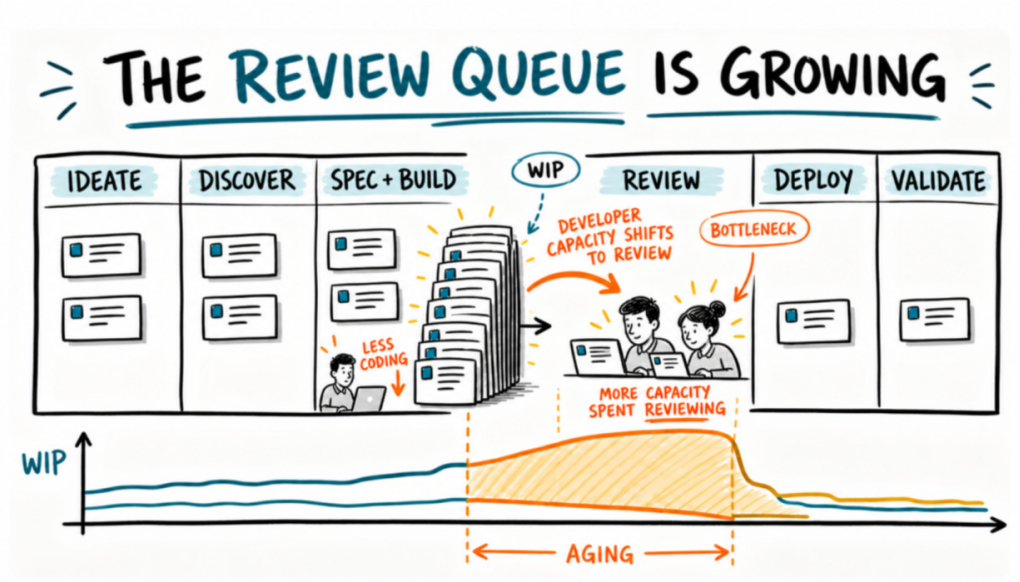
The evidence on unrealized value is thinner still. These reports don't settle whether established organizations are using AI to enter new markets or address needs they couldn't serve before. That remains a question for product and business evidence. Delivery telemetry alone can't answer it.

4. Manage the flow of the whole factory

Code review is a visible example of how the work changes. Less time spent coding can mean more time spent reviewing. Larger pull requests accumulate, and the people who can review them become scarce.

Before deciding what to do, we need visibility across the lifecycle. How much work is waiting for review? How old is it? How much of the total elapsed time does it spend there? What happens after review?

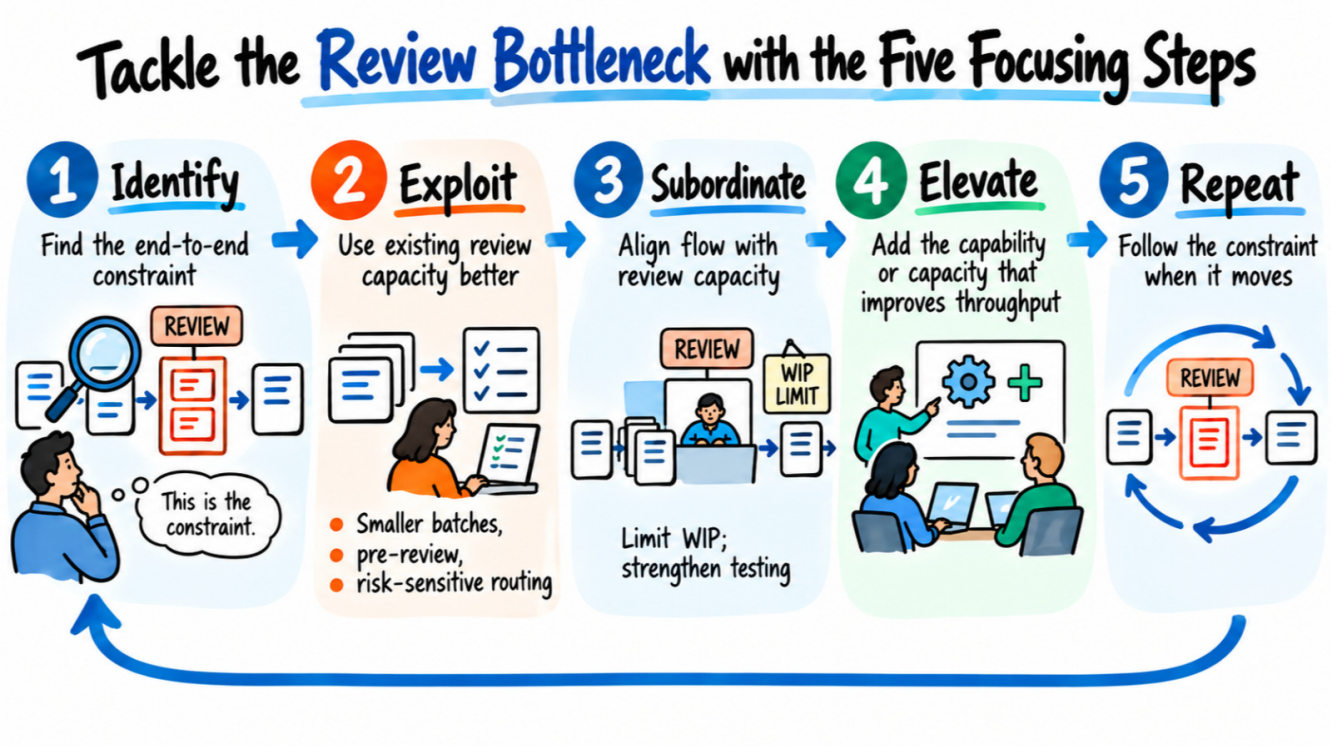
A pull request view can answer some of these questions. It may not show the relationship between that queue and the rest of product development. Whatever tool we use, we still need to understand the end-to-end flow.



Less coding can create a larger queue for review rather than more completed value

We've seen this pattern before. Long before generative AI, work accumulated between implementation and validation. Solving review can expose QA as the next constraint. Solving QA can expose deployment, adoption, or the ability to observe what is valuable and feed that information back into product decisions.

Goldratt's Theory of Constraints is still useful in this environment. The five focusing steps give us a disciplined way to respond.



Identify, exploit, subordinate, elevate, and repeat as the constraint moves

Identify the end-to-end constraint. Look at where work accumulates and what is holding back useful completion. The busiest person or most visible queue isn't automatically the constraint on the whole system.

Exploit the capacity already available. For review, that can mean smaller batches, smaller pull requests, and better preparation. If inexpensive automated tests can catch problems before an expensive human review, consider changing the sequence. Tell the agent what the reviewer needs to see. A reviewable artifact should be optimized for the human experience, rather than whatever format the agent happens to produce.

Risk matters too. Some work needs a full human review. Some might be suitable for selective review of important aspects. A bounded task with a proven safety net may eventually need less routine human involvement. The decision depends on what the system has demonstrated and what could happen if it fails.

Subordinate the rest of the flow to the constraint. When the queue reaches a limit, stop starting new work upstream. Developers and agents shouldn't keep generating specifications and code that will only join the pile.

Use the available capacity to improve review, strengthen testing, or change the system that creates the work. Work-in-process limits create that signal. They make space to innovate how we innovate.

Elevate the constraint when additional capacity will help. More people, more agent capacity, or more tokens can make sense once we understand how they improve the flow. Adding capacity upstream of a blocked review stage simply gives us more work waiting for review.

Repeat. Once the constraint moves, return to the first step. We should expect it to move as the factory improves.

These principles can work for an individual as well as a team. An individual can also start too much work, create a queue they can't review, and spend more time switching between unfinished items than completing something useful.

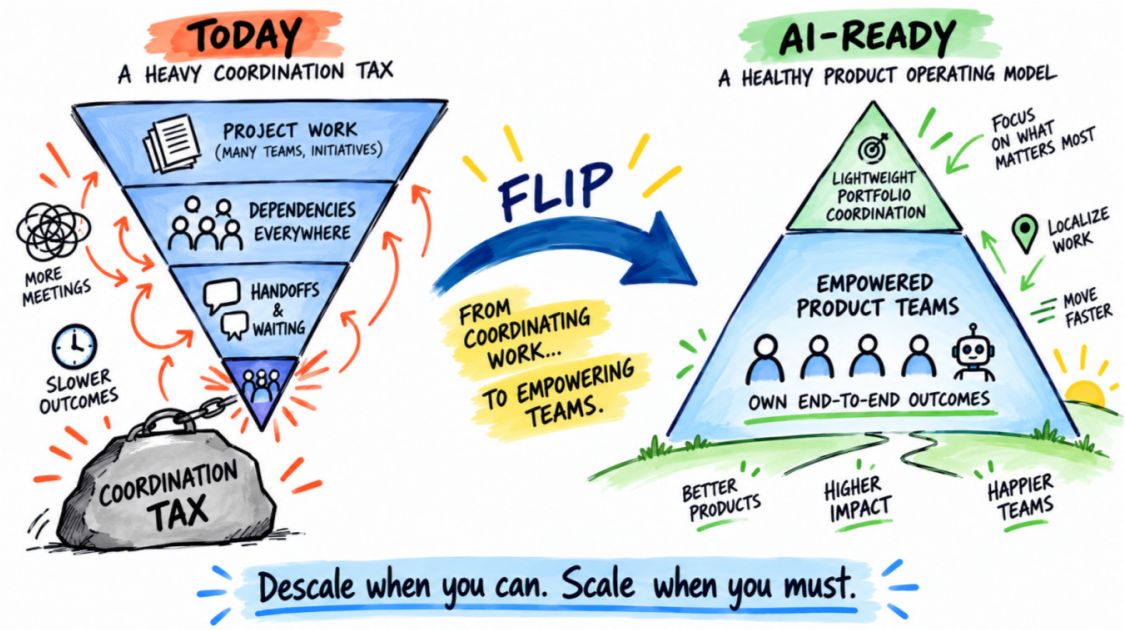
5. Flip the coordination pyramid

Agency needs the right structure.

Many organizations call themselves product organizations with cross-functional feature teams. But when we look at the work, very few teams can own an outcome end-to-end. Work cuts across specialized people, systems, teams, divisions, and portfolios. Handoffs and waiting are everywhere.

The result is a heavy coordination tax. More meetings and more people managing dependencies become necessary just to keep work moving.

If AI is used in single-player mode, each person works faster with the context and authority they already have. They still get stuck on the same dependencies. The local improvement doesn't remove the need to negotiate across the organization.



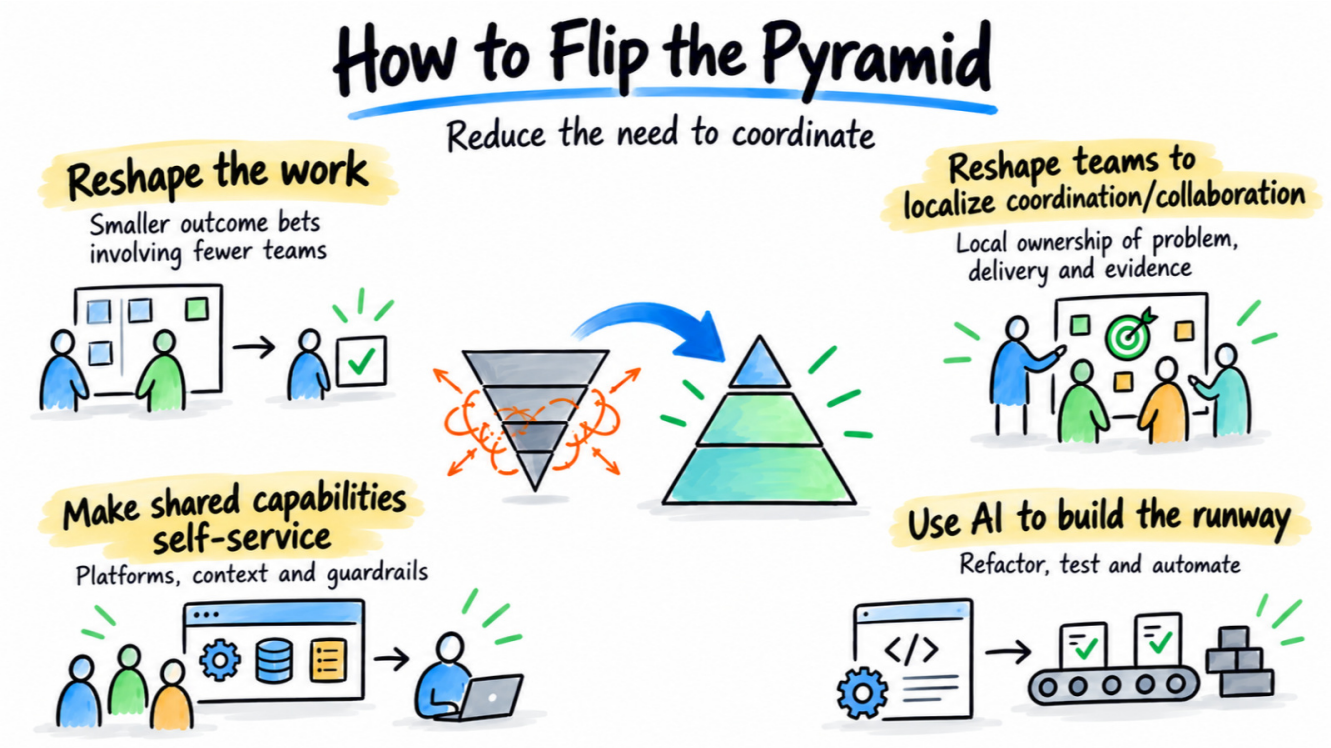
Flip the pyramid: reduce coordination tax and give teams end-to-end ownership

An AI-ready product operating model needs more work to flow within a team, with enough knowledge, context, and decision-making ability to own a useful outcome. We talk a lot about agency for agents. Humans need the structure that makes agency possible too.

The aspiration is for a small team, and the agents working with it, to understand a problem, explore it, deliver a solution, and close the loop on evidence. An agent may need context across multiple repositories and modules to do that. A team may need a different boundary or a different way to consume shared capabilities.

Small organizations can find this easier. As they scale, the challenge is to retain that end-to-end ownership. Organizations that already have a heavy coordination structure need to work out how to flip it.

Four ways to reduce the need for coordination



Reshape the work and teams, make shared capabilities self-service, and use AI to build the runway

Reshape the work. Smaller outcome bets can involve fewer teams. The question is whether a smaller group can own a useful change all the way through, rather than whether we can split a large project into more tasks.

Reshape teams to localize coordination and collaboration. Give a team the knowledge and skills to own the problem, discovery, delivery, and evidence. Calling a team empowered doesn't help if every meaningful decision or change still requires another team's involvement.

Make shared capabilities self-service. Some capabilities are complex and needed across the organization. Platforms, context, and guardrails can let teams use them without negotiating every change. The consuming team needs confidence that it can work safely. The team responsible for the shared capability needs confidence in how others use it.

Use AI to build the runway. Refactoring, automated testing, internal platforms, and better context were often postponed because capacity was scarce. AI creates an opportunity to do some of that work. It can help smaller teams work more broadly across a codebase and build safety nets around areas people are afraid to touch.

Internal capabilities can also be a good place to practice closing a loop. The users and interfaces may be more accessible than those of a new customer-facing feature. That doesn't make the work automatic, but it can make the usefulness of an improvement easier to examine.

Some organizations put this structural work on hold when attention and budgets moved to AI. The hope was that AI would make the harder organizational changes unnecessary. In practice, those changes often determine whether AI remains a local optimization.

Descale when you can. Scale when you must.

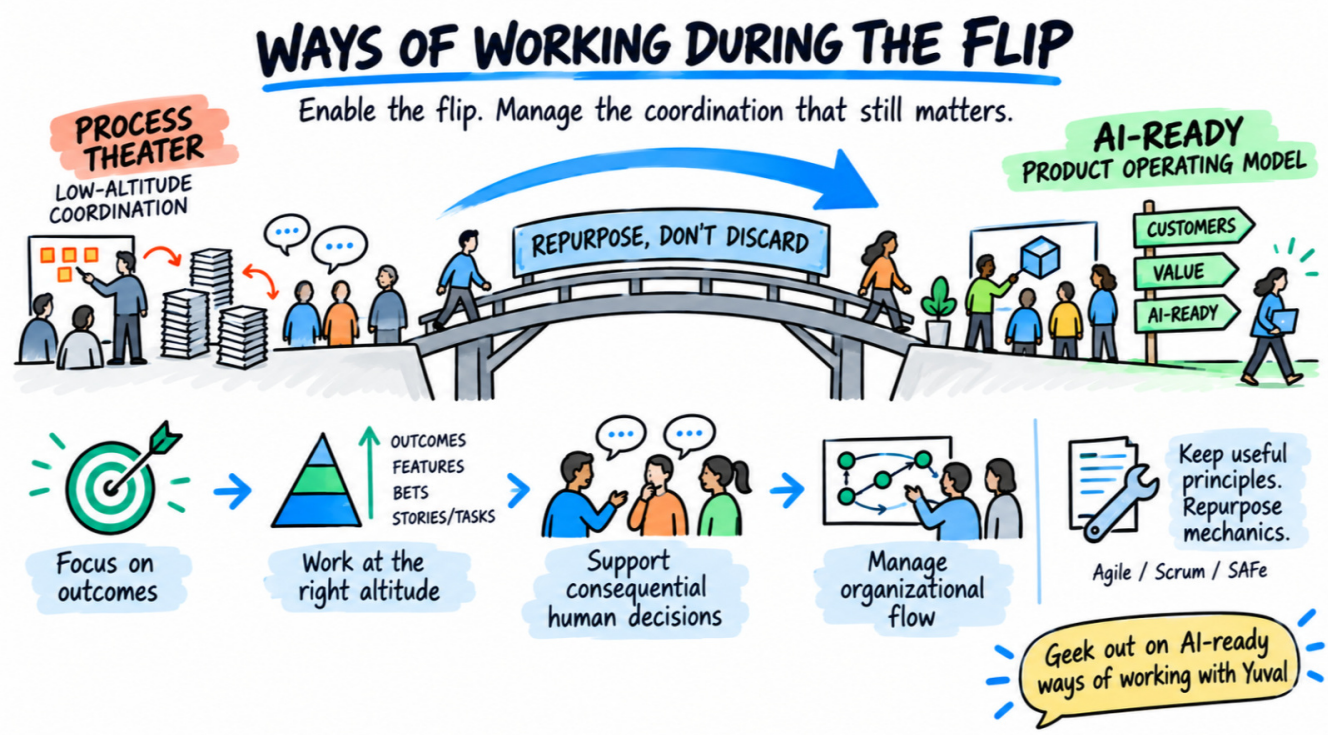
6. Evolve ways of working at the right altitude

AI gives people another reason to question existing processes. Do we need stories? Do we need weekly planning? What should happen to Scrum, Kanban, SAFe, or the other approaches sitting above the coding practice?

A lot of what organizations were doing had already become process theater: low-level coordination and project management dressed in the language of product teams and agility. AI makes the mismatch harder to ignore.

But removing the mechanics doesn't remove the need to manage work. Teams still need to decide what matters, make consequential decisions, coordinate where dependencies remain, and learn from evidence.

The useful move is to separate first principles from their manifestations. Flow, work-in-process limits, self-organization, outcome orientation, and empirical learning remain relevant. A particular meeting, artifact, or cadence may need to change completely.



Ways of working should enable the organizational change and support consequential decisions

Much of the lower-level work can be handled by agents. That frees people to work at a different altitude. Instead of spending their attention grooming stories, they can focus on the outcome a team is trying to create, the bets it is making, and what the evidence says.

Eventually, we may give a team or an agent an outcome and a leading indicator, rather than a list of features. It can then work out which features might improve that outcome. The team still needs an operating system for running that learning loop.

And while organizations are making this shift, dependencies don't disappear because the vision says they should. Coordination that still matters has to be managed. The tiny team that can do everything end-to-end is an attractive vision, but many organizations need time and deliberate work to get there.

There isn't a settled answer for the right cadence or the right division of work between humans and agents. Those are things to experiment with. Throwing out a work-management tool can remove frustration, but it doesn't answer the questions that tool was meant to help us address.

7. Connect the factory to product outcomes

We could do all of this well and still produce software that people don't use.

A factory measured on delivering software will optimize for delivering software. It won't necessarily ask whether customers adopted a capability, whether their behavior changed, or whether the change created value.

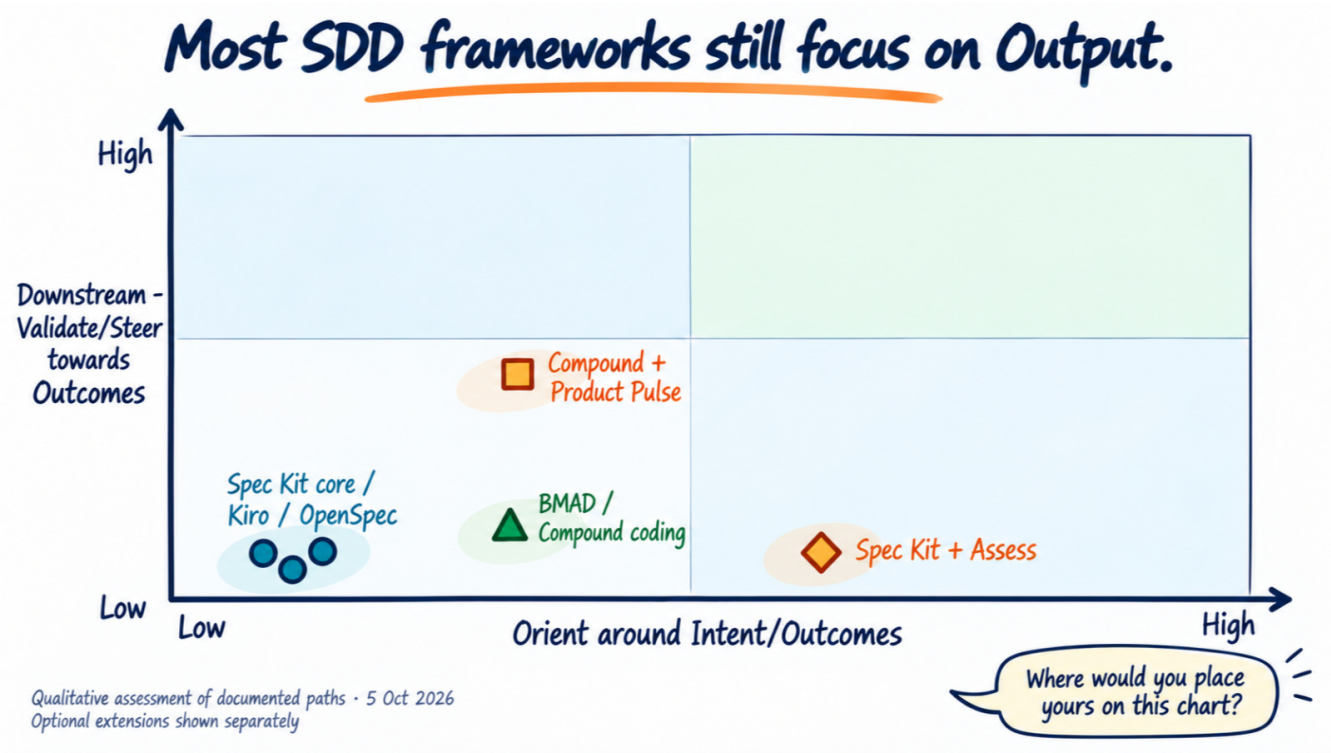
This is why outcome orientation has to extend into the work itself. When I look at repositories and work-management systems, much of the work is described as activities or outputs. It isn't clear what behavior we expect to change or why the work should matter. I've seen the same pattern in client backlogs.

Teaching product people to talk in outcomes gives better context to the agents and teams. Effective agency depends on alignment to what we're really trying to do. If we want to give agents the keys to choose which features to work on, we need to improve the outcome orientation.

Examine both ends of the framework

I examined several spec-driven frameworks along two dimensions: how strongly their documented workflows orient around intent and outcomes upstream, and how strongly they validate and steer toward outcomes downstream.

The October 5, 2026 assessment suggested that much of the activity was concentrated in the middle: turning specifications into implemented, verified code. Some optional extensions improved parts of the picture. The complete chain from intent through adoption evidence to an accountable outcome decision was much less established.



A qualitative October 2026 assessment of upstream intent and downstream outcome steering in spec-driven frameworks

This is a qualitative assessment of documented paths at that time. It isn't a measured performance ranking or a permanent statement about those tools. Optional configurations are shown separately from their core frameworks.

The useful questions apply just as well to a homegrown harness:

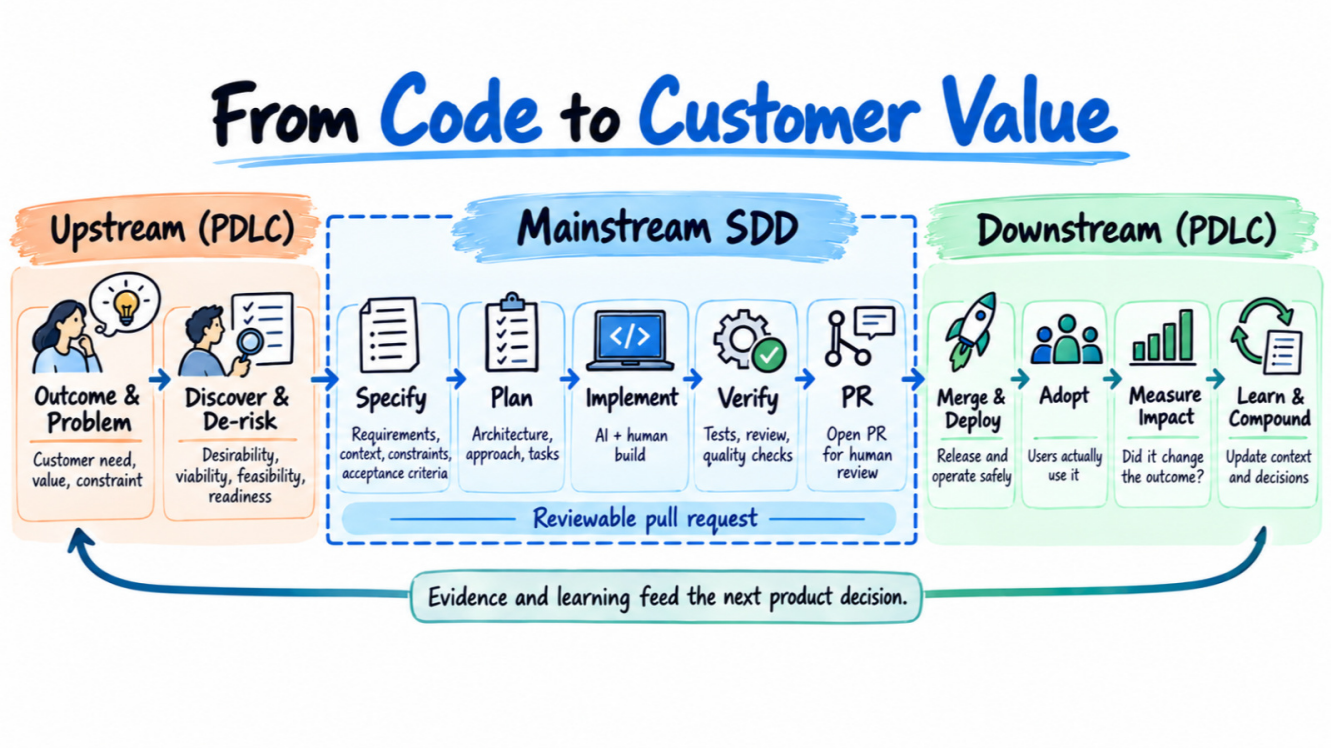
- Does it establish the problem, the intended user, and the outcome hypothesis before it commits to requirements?
- After deployment, does it examine adoption and actual behavior?
- Does someone use that evidence to decide what should happen next?

A workflow can be excellent at implementing a specification and still leave all three questions outside its scope.

8. Extend the software lifecycle into a product lifecycle

The shift from an agentic software development lifecycle to an agentic **product development lifecycle** adds work both upstream and downstream of the mainstream spec-driven loop.

Upstream, we establish the outcome, problem, and intent. Downstream, we examine adoption, measure impact, and use what we learn to decide where to go next.



The full product development lifecycle includes discovery before specification and adoption and learning after delivery

Upstream: establish intent and de-risk the bet

A Lean Product Canvas, Lean Canvas, epic hypothesis, or another useful framing can help establish who we are focused on and what outcome we want to see. The artifact is a means of making the thinking explicit.

We identify the users, the problem, the outcome for the organization, and a solution hypothesis. Then we examine how much conviction we have. What assumptions are we making? Do we need discovery to reduce uncertainty about desirability, viability, or feasibility?

We don't need discovery for everything. If we know enough, we can build. When an opportunity is valuable enough to pursue but carries meaningful uncertainty, de-risking it can be the better next move.

This is different from taking a feature request as given and asking an agent to write a detailed specification. The upstream work gives the agent and the team a reason to question the proposed solution and room to find a better one.

The engineering loop: build with discipline

Specification, planning, implementation, and verification still matter. We still need requirements, architecture, constraints, tests, and review appropriate to the work.

The difference is that the engineering loop now sits inside a larger product loop. A reviewable pull request is an important boundary in delivery. It is an intermediate result in the effort to create an outcome.

Downstream: close the loop on use and value

Deployment makes a capability available. Adoption tells us whether people actually use it. Measurement helps us understand whether it changed the outcome we intended to change.

Depending on the product, we might look at awareness, activation, retention, or other evidence of use and value. Internal capabilities need the same thinking, even when the relevant value isn't revenue.

The point is to follow through. Did the change enable a useful behavior? What did we learn? Does the next step involve another feature, a change to what we built, or a different approach?

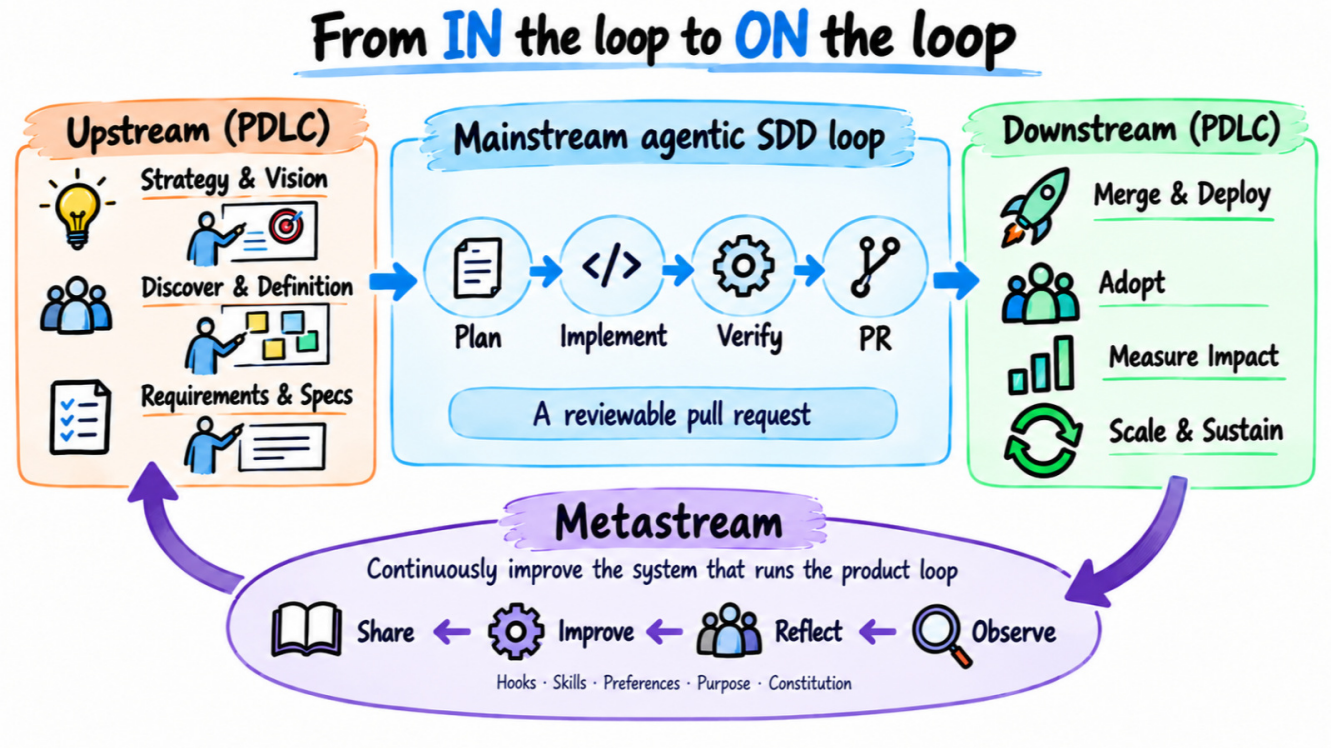
That evidence needs to come back into the next product decision. Otherwise we have a sequence of specifications and releases, with learning left to happen somewhere else.

The broader loop also changes who needs to work together. An engineering team can improve implementation on its own, but an end-to-end product lifecycle crosses product, engineering, operations, and the people who see what customers actually do.

9. Treat the factory as a product

Closing the product loop gives us evidence about what we built. We also need to learn from **how we built it**.

That is the metastream: observe, reflect, improve, and share. It continuously improves the system that runs the product loop.



The metastream learns from the product loop and feeds changes back into the operating system

The operating system has a current version. It includes the skills, context, preferences, hooks, constitution files, and other instructions that shape how humans and agents work. Those are things we can inspect and improve.

The team struggling with walls of Markdown provides a simple example. Moving specifications to a web page or Confluence is an experiment in the factory's user experience. The hypothesis might be that people will read more, understand decisions better, and contribute their own judgment.

We can try the change on a few specifications and examine whether that behavior changes. If people still don't understand the material, the experiment hasn't achieved its purpose. The next change may need to be to the content rather than the presentation. Perhaps the specifications need a clearer outcome and fewer details about every step the agent intends to take.

Agents can participate in improving the system

Humans don't have to do all of the reflection themselves. Agents can examine what happened and propose changes to the process. A proposed improvement might be a pull request against a skill, a context file, or a workflow rule.

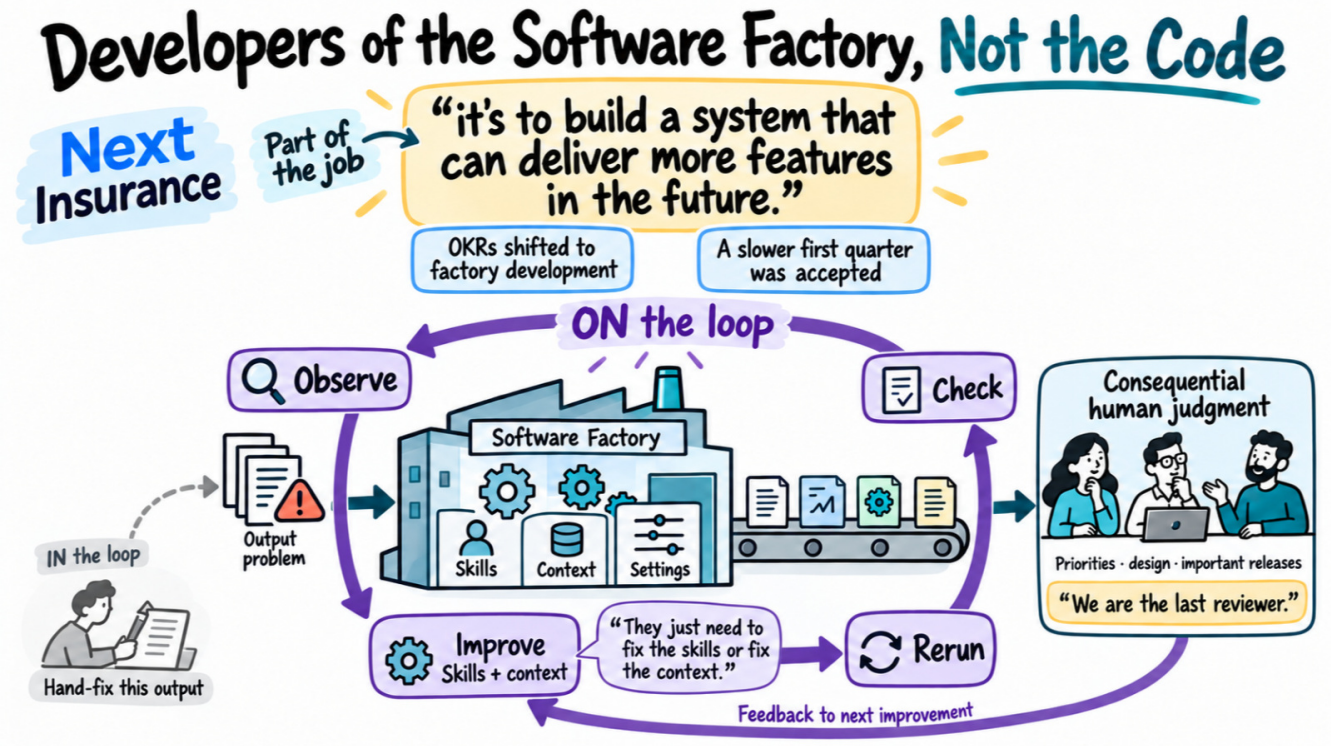
Initially, I would expect humans to stay closely involved in this improvement loop. They can review the proposed change, adjust it, and decide whether to incorporate it. Over time, some bounded improvements may themselves be delegated as the system earns trust.

10. Become developers of the factory

Shay Mandel's account of the work at Next Insurance makes this change concrete. The aim was to build a system that could deliver more features in the future. Part of the work, and some of the team's objectives, shifted toward factory development. Management accepted that the first quarter could be slower while the team built the machine.[2]

That changes what an engineer does when an agent makes a mistake.

The immediate instinct is to fix the output: correct the specification, change the code, or tell the agent what to do this time. The alternative is to inspect the skills and context that led to the decision, improve those, and run the work again.



Factory development means changing skills and context, rerunning, and checking the result

The distinction is between being **in the loop**, repeatedly hand-fixing outputs, and being **on the loop**, observing and improving the system that produces them.

A practical example came from a leader whose solutions engineer used a skill to make configuration changes through Git. The agent kept branching from the wrong place. The first reaction was that the person knew better and should correct the agent. But the same problem kept happening. The more useful diagnosis was that the skill was wrong.

When there's a mistake, do we fix the output, or fix the process, retry it, and see whether the system took the right approach?

That is the role of the factory's developers and maintainers. They need to understand the capabilities and limitations of the agents and design the tasks, context, guardrails, and escalation around them. If the factory repeatedly puts an agent in a situation it cannot handle reliably, the factory needs attention.

In Shay's account, this didn't mean abandoning consequential human judgment. Humans still reviewed important product and business decisions, technical plans, and important changes before production. The aspiration was for automated review to do enough useful work that people could become the last reviewer, rather than the first.

The engineering challenge moves

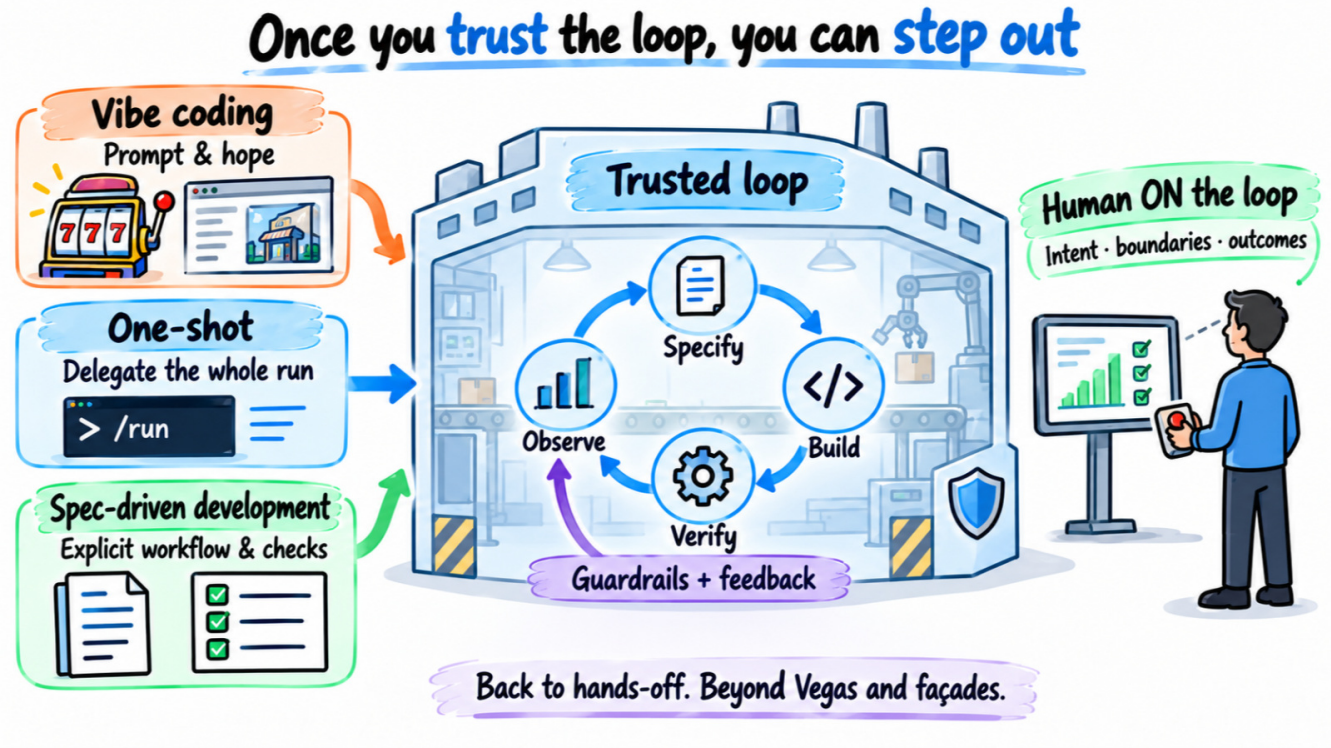
I've been coaching a senior engineer who found the move to AI frustrating. Much of their enjoyment came from troubleshooting hard problems. Looking at the new work as babysitting agents made it seem less interesting.

Owning the agent system changed that perspective. Building a trustable software factory is a difficult engineering problem. Skills, context, verification, observability, and the way they work together all need design and troubleshooting.

It also asks us to change where we take pride. Over time, we should be able to take more pride in a system that can produce and verify useful work, with appropriate intervention, than in the amount of generated code we personally reviewed.

11. Earn the right to step out of routine execution

We want to get to the point where we can name a skill, a feature, or an outcome we want to achieve, and trust that AI will take it end-to-end. We trust the system we've created: we've sampled it and run it multiple times. Sometimes it will fail, and we have a process for fixing it.



A hands-off request can run through an explicit engineering loop, with the human governing intent, boundaries, and outcomes

Vibe coding, one-shot execution, and spec-driven development can converge in this sense. A low-friction request can have real engineering machinery underneath it. One-shot execution can mean delegating a whole disciplined run, rather than hoping a single prompt produces something sound.

You don't necessarily need the slow, spec-driven, step-by-step process with a human in the loop each and every time. Over time, we should shift from in the loop to on the loop: toward the outcomes we want the system to create and closing the observability loop.

Verification has to evolve too

Verification doesn't have to mean reading every line of code. It can include tests, outcome checks, observability, and sampling. Over time, more of our work may involve verifying that the verification agents and the surrounding system are trustworthy.

That still leaves hard questions. If agents verify other agents, who verifies the verifiers? When the system changes its own skills or workflow, what evidence do we need before trusting the revised version? How much can a person reasonably understand at once?

Modular work that can be verified in smaller units can help people reason about the system. But I don't think we have a complete answer to how far this can go. These are questions for engineering leaders to work through as the capabilities develop.

In the meantime, sampling and human judgment remain important. The appropriate checks will depend on the consequences of failure and on what the particular loop has demonstrated. Trust in one kind of task doesn't establish trust in every other kind.

Accountability stays with the people building the system

I often return to the analogy of a compiler. It is another layer we depend on without manually reproducing everything it does. The analogy doesn't make a non-deterministic model equivalent to a compiler. It raises the question of what we, as developers of the surrounding system, are responsible for specifying and checking.

If an agent pulls from the wrong branch, ignores needed context, or works outside an intended boundary, we need to examine the system that guided it. Did we define the guardrails? Did we provide the context? Did we design a process suited to the agent's limitations?

The model's limitations don't remove our accountability for how we use it. We can put a deliberate, repeatable process around a non-deterministic component and require evidence that it creates useful outputs and outcomes.

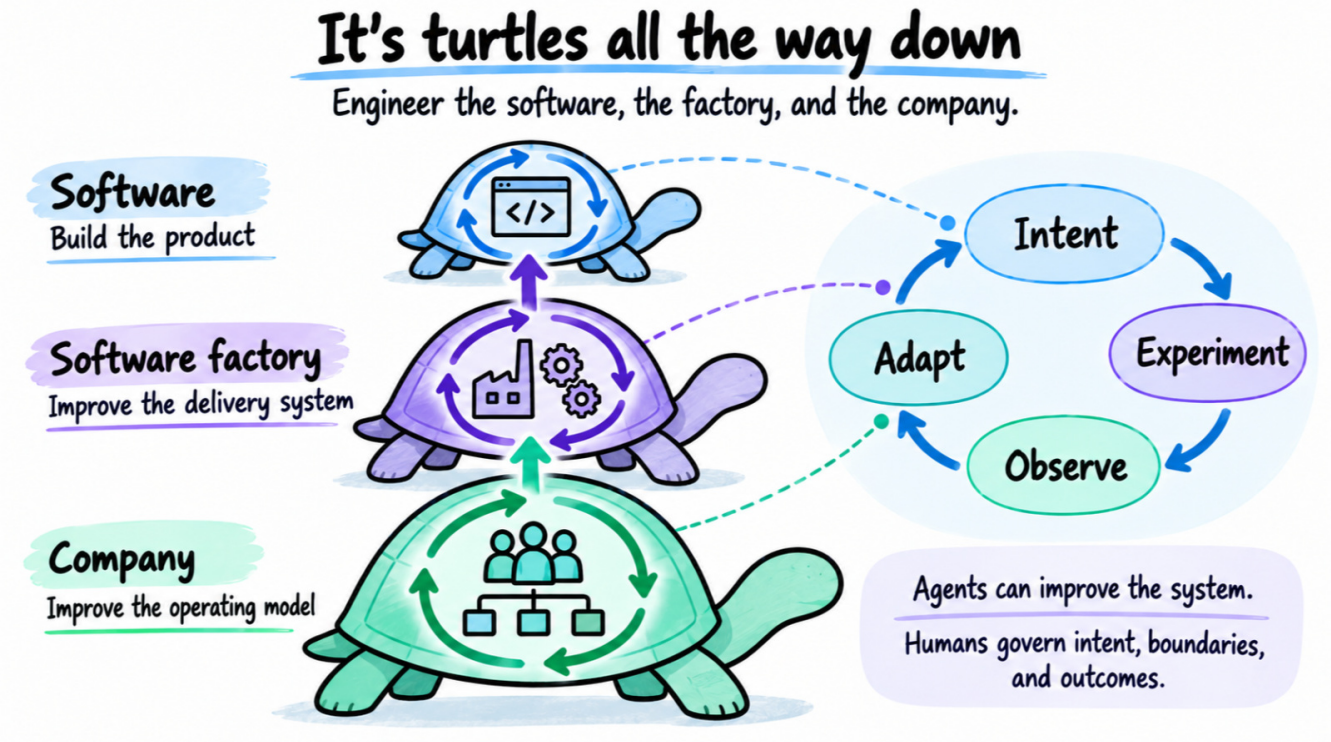
There is also a question about incentives. If the organization rewards output while claiming to care about outcomes, the factory can become very effective at pursuing the wrong thing. Better execution doesn't resolve a mistaken objective. That is another reason to retain human responsibility for purpose, boundaries, and consequential decisions.

12. It's turtles all the way down

The same pattern can be applied at more than one level.

We develop the software toward an intended outcome. We develop the software factory that runs that product loop. We can also develop the company's operating model: how it sets priorities, makes decisions, coordinates work, and learns.

Each system supports another system, and each can have its own improvement loop.



Software, the software factory, and the company can each be developed through intent, experiment, observation, and adaptation

The increased capability of the product and engineering organization also creates room to evolve the product for the AI age. That might mean positioning it for an agent ecosystem, rethinking the experience for both people and agents, or changing how customers experience the product. Faster implementation is useful when we can connect it to that product thinking.

Engineering and product thinking also become relevant beyond the product and technology organization. The capabilities we build for software development can help the wider business build and improve its own AI-enabled capabilities.

The discipline remains recognizable: establish intent, choose an experiment, observe what happens, and adapt. At the factory level, that may mean changing a skill or a verification step. At the company level, it may mean revisiting a decision rule, an organizational boundary, or the assumptions behind an objective.

Agents can participate in these improvement loops. People still need to govern what the system is trying to achieve and the boundaries within which it can change itself. We shouldn't assume that a company becomes deterministic just because more of its operating model is explicit and maintained as code.

As engineering and software leaders, I think we have a role in spreading this loop engineering and continuous improvement across the whole company.

Sources

1. Faros, AI Engineering Report Q3 2026; DX, State of AI Impact in Engineering: Q2 Report; LinearB, 2026 Software Engineering Benchmarks Report. Faros's comparisons are observational within-team higher- versus lower-adoption quarters; agentic-review adoption is a separate comparison. DX's innovation ratio measures effort allocation.

2. Shay Mandel-Yuval Yeret interview, especially skills and context around 9-12 minutes, human intervention around 16-20 minutes, and factory-development objectives around 41-42 minutes. This is a practitioner account of the approach and its intentions, not evidence of a measured or guaranteed future return.

About the author

Yuval Yeret works with product and engineering leaders on scaling AI from activity to impact. He helps organizations develop a trustable software factory, give people and agents effective agency, and connect delivery to adoption and useful outcomes. His advisory and coaching work combines hands-on experimentation with changes to team ownership and ways of working.

Work with Yuval on your AI product development lifecycle.