NOBL9

INDUSTRY REPORT

NOBL9 INDUSTRY REPORT · 2026

The Velocity Asymmetry

Software at Machine Speed, Reliability at Human Speed

An industry report on what happens when AI writes the code and humans still carry the judgment.

Published by Nobl9 | 2026

Executive Summary

Two changes are occurring simultaneously at software companies (which, these days, is almost every company). Teams use AI to write code, and ship AI-powered features into production.

Industry discussions often treat them as separate trends. Yet these are two fronts of the same battle for operators: a growing share of what runs in production is software that humans do not completely understand. An AI agent can write a working feature without a human ever reading the code it sits on top of, the services it touches, or how a real user will move through the journey it just built. The code works, but nobody knows how it behaves under different constraints. Meanwhile, the services underneath lean on models and external APIs that act differently from one run to the next. In both cases, the working knowledge that every reliability practice depends on never gets ingrained.

IN THIS REPORT

The impact down the road is potentially disastrous. Teams adopting AI coding assistants see 3.4 times more production incidents per code change (Faros AI Engineering Report, 2026). If you chain ten AI workflow steps together at 85% reliability each, the full journey succeeds one time in five (Temporal, 2026). Confidence is thinnest where systems are largest: only 19% of teams running AI in production are very confident their stack can handle two to three times current scale. At organizations with over 500 engineers, none of them are confident their stack can hold up (Inngest, 2026).

This does not reflect poorly on the teams involved. The practices that kept software reliable for two decades assume change arrives at a pace people can keep up with. This is because, even with advancements in automation and tooling, the process was still driven by humans. That assumption quietly failed, and it's resulting in some very loud chaos for teams that have embraced AI the most. We'll take a look at the evidence of the trends, explain why first instincts at a fix might not solve the problem, and go through what closing the AI reliability gap actually takes.

The Constraint Moved

> AI multiplied the rate at which software gets written. The processes that keep it reliable still assume a person read every line. This section measures the gap and shows where it first turns into incidents.

“

People are forgetting that the hardest part about software engineering is accounting for edge cases. This was always a problem, but how much more software is being built, thanks to AI, has made it orders of magnitude more difficult to solve. The surface area has exploded.

ENGINEER, 11-50 PERSON ENGINEERING ORG

QUOTED IN INNGEST’S “AI IN PRODUCTION: THE 2026 BENCHMARK REPORT”

For most of software's history, the limiting factor on how fast an organization could ship was how quickly its people could write code and test it. The practices that keep software reliable were built around that limit. Code review was efficient because changes arrive at a pace a reviewer can actually read. On-call worked because each service had someone who knew it well. Postmortems were effective because incidents are rare enough to study individually.

As a side effect, each of those practices does more than catch defects. They represent how a team builds its working knowledge of a system: a sense of what normal looks like, which dependencies are fragile, and which failure modes repeat. Because every step of building software was done by humans, the whole process ran at human speed, and this knowledge kept pace with the change.

Teams made decisions on the speed tradeoff based on their unique business constraints. Some shipped fast and often paid for it in incidents. Others slowed down to protect stability. Either way, the systemic knowledge accumulated as a natural side effect of doing the work. Teams became better at catching mistakes because they knew where to look. They inherently understood the systems they built because of the work they did to manage them.

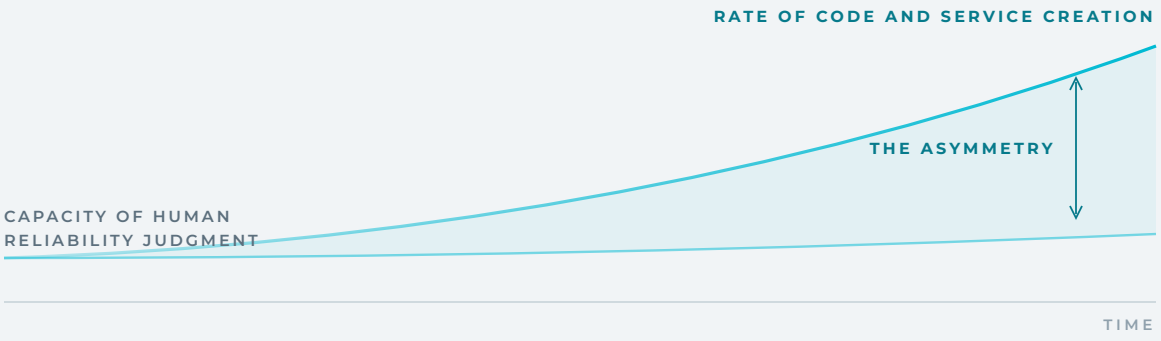
AI has effectively removed that safeguard. Teams now risk failing on two fronts. The first front is code written by AI. 86% of organizations already use AI coding agents in their development lifecycle, and 80% report measurable returns, so adoption will continue to accelerate (Anthropic, The 2026 State of AI Agents Report). An agent writes code to fulfill its context. It's inherently different from a human engineer who builds their own context as they work. While it can produce a correct-looking feature, it has no sense of the architecture it lands on, the load patterns of the services it calls, or the way users will actually move through the journey it built. Now multiply that across several times the old shipping volume. Review cycles fall behind, and the shared understanding that debugging and deployment decisions depend on never forms.

The second front is the behavior of AI features themselves. A traditional program does the same thing every time it runs, so a team can learn its habits. A feature built on a language model works differently. The same request can produce different responses, a dependency can degrade without ever returning an error, and an answer can arrive successfully but be wrong. Even a team that reads every line of its own code cannot fully learn the habits of a component that changes its behavior from one run to the next. These features and their accompanying reliability are inherently unpredictable.

While these are two frequent symptoms of the shift to AI-enabled development, both are adequately described as a common condition: unabsorbed change. Put plainly, software is entering production faster than the people responsible for it can build an understanding of how it behaves. On the contrary, absorbing a change means reading it, running it, and watching it in production long enough to know what normal looks like. Both AI fronts interfere with a team's ability to process and "absorb" change.

It is clear from the data that it has never been harder to manage software reliability at scale. As a result, incidents and outages have been on the upswing throughout the last year. This, however, is not a failure of reliability teams or the telemetry systems they have in place. Teams already run advanced monitoring, alerting, tracing, and incident management, designed to show them system performance in an incredibly detailed way. However, they carry an assumption that a human has time to interpret what they show. The volume of judgment calls now arriving was never part of their design, and no team is staffed for it.

The impact shows up in three places (and maybe more that we aren't considering yet): riskier changes, engineering time draining into reactive work, and confidence that falls as organizations grow.



The Measurable Cost

> Reliability work is claiming a larger share of engineering time exactly where AI adoption is deepest. This section follows the cost through time spent, incident rates, and confidence, and ends with the compound math that makes multi-step agent workflows fragile.

FINDINGS DRAWN FROM INNGEST'S SURVEY OF 130 BACKEND, FULL-STACK, AND AI ENGINEERS

The clearest single measurement of disruption comes from Faros AI, whose 2026 engineering report found that teams adopting AI coding assistants experience 3.4 times more production incidents **per code change**. If teams were simply shipping more changes, total incidents would rise even if each change stayed just as safe. However, the data shows clearly that each individual change became more likely to cause an incident regardless of any increase in volume.

3.4×

more production incidents per code change as teams adopt AI coding assistants.

FAROS AI ENGINEERING REPORT, 2026

This makes sense: engineers have, in many cases, lost their mental model of how their software is configured. Throughout the development process, the mental models engineers hold guide decisions about how to manage reliability. Even teams with the richest dashboards and tracing rely on mental models: the dashboards locate the symptom, and the model in a human's head supplies what the symptom means, which dependency to suspect, and what changed recently that could matter. Code that no one fully absorbed or built doesn't allow for the creation of a mental model, so problems surface later and take longer to trace.

As a result, and in part because of how much faster the coding process is, engineers are spending far more time making sure systems are reliable. In Inngest's 2026 survey of 130 backend, full-stack, and AI engineers, 20% of teams building AI in production reported spending between a quarter and half of all engineering time on reliability work. That is twice the rate of comparable teams that are not utilizing AI. It would be a mistake to read this as engineers falling behind on their planned work. AI sped up code writing, so planned work finishes faster than before. Instead of getting time back in their day, engineers now spend a disproportionate amount of time on reactive reliability work, either firefighting incidents or debugging generated code. Reactive work is not wasted work; fighting a fire in a service is one way to finally learn how it behaves. But it is the most expensive way to learn, arriving unscheduled and priced in customer impact, and the more of the recovered time it consumes, the less of the AI speedup an organization actually keeps. Coupled with incidents becoming more frequent in organizations relying on AI to code, the ROI for AI coding can even turn negative.

The same survey found 74% of AI teams reporting a customer-visible incident in the past 90 days against 62% of teams without AI features (Inngest, 2026). As AI-generated coding gets more entrenched within organizations, and the mental model engineers rely on gets foggier, that could signify more risk ahead.

The Green-Dashboards Problem

Multi-step agentic systems add a mathematical problem on top. Temporal illustrated it in April 2026: if each step of a ten-step workflow succeeds 85% of the time, the workflow as a whole succeeds roughly 20% of the time. The math is unforgiving. Every one of the ten components can look green on its own dashboard while the journey fails four times out of five. Call it the green-dashboards problem, and it predates AI. Component monitoring without journey-level measurement was always a partial view because users experience complete journeys while dashboards observe only individual parts. AI made the problem urgent by increasing the number of steps and shipping them faster than anyone can learn how they are composed.

Compound failure math for a 10-step agentic workflow. Source: Temporal, April 2026.

All of these factors lead to a general lack of confidence in the organization's reliability posture. 19% of teams running AI workflows in production report being very confident at two to three times current scale, and in organizations above 500 engineers the figure falls to zero (Inngest, 2026). The pattern fits the rest of the picture: the larger the organization, the more services carry unabsorbed change, and the more thinly the remaining working knowledge is spread.

Why the Obvious Responses Fall Short

> Teams are drowning in telemetry and still struggle to explain failures. This section examines the two most common responses, more instrumentation and AI-assisted review, and why neither closes the gap.

VERDICT

Organizations are already trying to sort out these problems. Two responses are the most common: instrument the systems more heavily, and point AI at the review bottleneck the same way it was pointed at the writing bottleneck. Much of the tooling industry is currently writing the first prescription. Neither solves the fundamental problem.

The first response is to invest in observability. When Inngest asked engineers an open question about the biggest unsolved problem in building reliable systems, observability led every other theme (Inngest, 2026). But a lack of instrumentation does not explain the gap. In the same survey, the most widely used tool category was application performance monitoring platforms such as Sentry, Datadog, and New Relic, and only 7 of 130 respondents reported running no structured observability at all (Inngest, 2026). These are the most heavily instrumented teams. If more data alone solved the problem, we would know it.

The green-dashboards problem is informative here. Standard observability tooling describes what a system is doing without concluding whether that behavior is acceptable. A dashboard returns time series and leaves the judgment to whoever is looking. A static threshold fires on transient noise and stays silent through slow degradation. Some teams have written part of this judgment down; many organizations maintain service level objectives for at least some of their services, and those objectives capture exactly the knowledge in question. In practice, that coverage tends to be partial and static. It exists for the oldest and best-understood services; it goes stale as systems change, and it is rarely in a form that software can read at the moment of a decision, whether that software is a deploy pipeline or an AI agent. The rest of the judgment still lives with the people who built each service.

That arrangement worked when the people holding the knowledge also reviewed every change. It degrades when change arrives unabsorbed, and it breaks down completely for AI agents, which cannot (or don’t know to) ask a colleague what normal looks like. An engineer paged at 2 a.m. and an AI agent about to deploy need the same two pieces of information: what counts as acceptable for this service, and how much risk remains. In most organizations, there is no single place where either of them can look up those answers.

The second response sounds like it dissolves the whole problem. If review is the bottleneck, let agents review. AI code review is already everywhere, it catches real defects, and it will only get better.

It still runs into two walls. The first is what reviewers can see. Review, human or AI, checks a change against its intent: whether the code does what it set out to do. But look at what actually takes AI teams down: external model and API failures, infrastructure crashes, and concurrency spikes (Inngest, 2026). None of that is visible in a pull request, no matter who reads it. The Faros numbers make the same point from the other side. The 3.4x spike was measured in a period when AI review tools were already in wide use, and the same report found the share of changes merged with no review at all went up as adoption deepened.

The second wall is the reference point. To judge whether a change is acceptable for production, a reviewer has to know what the service owes its users. That knowledge is exactly what is missing. An AI reviewer without a written target inherits the same blindness as an AI writer. Which is the deeper answer to the objection: as agent review improves, it becomes one more consumer of the missing contract. Review judges the change before it ships. A reliability target judges the outcome in production, where these failures actually happen. Teams need both. Only one of them exists anywhere by default.

More instrumentation

describes what a system is doing

AI-assisted review

checks a change against its intent

A reliability target

judges the outcome in production, where these failures actually happen

The practical question, then, is what it would take for a system of record to hold that judgment.

What Judgment at Machine Speed Requires

> Four requirements let reliability move at the same speed as the code. Each is a practice any team can adopt today. Together they give humans and AI agents one shared definition of acceptable.

These findings considered together read like a recipe for disaster. Changes are riskier because nobody understands them (like, deeply, deeply understands what's going on). User journeys fail while component dashboards stay green. The teams in trouble are already drowning in telemetry. And a growing share of the actors touching production are not people. Fortunately, we are not doomed to a dystopian future of failure. Addressing the shortcomings of AI-driven development requires a solution that addresses four primary areas:

1. It should measure complete customer journeys, because component health alone misses the failures users actually see.

2. Its definition of acceptable must be easy to revisit and revise as the service changes, because a threshold set once will be wrong within months.

3. It must work across whatever telemetry a team already runs, because well-instrumented teams will not rebuild their stack to adopt it.

4. Finally, it must be readable by machines, so that an AI agent can consult it at the moment it is building and deploying software.

Fortunately, we have a tradition from existing SRE practices that addresses all these areas: Service Level Objectives. An SLO is a short, written statement of what users can count on from a service, covering things like availability, speed, and correctness, expressed so that both a person and a program can evaluate it against the journey a user actually experiences. What changes in an AI-heavy organization is what that written target does. It moves institutional knowledge out of individual minds and into a shared understanding between teams. More importantly, it also hands AI agents the same reference point people carry. A code reviewer, a deploy pipeline, and an AI agent looking at the same service reach the same conclusion. The target also travels upward: a journey-level objective reads as a statement about customer experience, which makes it one of the few reliability artifacts a business leader can act on without translation.

The error budget is what makes the target operational, for people and agents alike. A non-perfect objective, meaning anything below 100%, implies an allowance for some level of incidents or mistakes; the error budget is that allowance, measured continuously. For a team, it answers how worried they are right now. For an agent, it is a machine-readable signal for how much room there is to act: ship freely against a healthy budget, defer more decisions as it burns, and hand the wheel back to a human when it runs out.

BUDGET HEALTHY
ship freely against a healthy budget

BUDGET BURNING
defer more decisions as the budget burns

BUDGET EXHAUSTED
hand the wheel back to a human when it runs out

Coverage has to arrive with the code for any of this to keep pace. Supervising each agent action individually introduces so much friction that the agents become useless. Thus, the workable answer is a framework that defines how agents are meant to build and deploy: a paved road whose default path already includes the reliability definitions. On that road, the pull request that creates a service also carries its objectives. They are reviewed like code and exist from day one. The same automation that writes the service can draft its objectives from the change and from history. What's more, thirty days of production data is generally enough for a defensible starting target.

One caution still applies: an AI-drafted objective needs a methodology behind it. A target derived only from what the service currently does will ratify current behavior, whatever that happens to be. Useful drafts start from the user journey and arrive as proposals for a person to accept, tighten, or reject. With that discipline in place, an agent following the paved road produces fewer incidents because reliability was designed into its route, with no one standing over each deploy.

Objectives also age, and AI has shortened the timeline. A target that would have stayed accurate for a year at human pace will lose its meaning in a week when agents extend the service daily: dependencies accumulate, traffic patterns shift, the team that set the target moves on. Stale targets still look like coverage on a status page, even though they measure services that no longer exist in the same form. Keeping targets current requires a named owner and a review cadence for each objective, with the gap between a service's rate of change and its last review itself tracked. This maintenance is also work AI can carry: the same tooling that drafts an objective can flag when a service has outrun its target and propose a revision for the owner to review, so the review cadence keeps pace with the change it is meant to track.

Finally, the same information must be available to the software doing the work, which is where the requirements converge into a closed loop. Before acting, an agent queries what the service promises and how much error budget remains, the same check a careful engineer would run. During a rollout, it watches the budget's burn rate and slows or reverts when burn accelerates. Afterward, it records what it observed in the same system of record, where the next actor to touch the service, human or agent, will find it. Interfaces in the style of the Model Context Protocol, which give agents structured access to external systems, make this practical today: reliability state becomes one more tool an agent can call. An agent operating inside that loop applies the judgment the team encoded, at whatever speed it works.

Reliability state as a closed loop: every actor, human or agent, consults and updates the same system of record.

Where Does Your Team Stand

The seven questions below can be answered in a single meeting. Each “no” indicates an area where the asymmetry is currently unmanaged.

1. ☐ **Definition.** Can you state, in machine-readable form, what “reliable enough” means for your five most critical services?
2. ☐ **Verdict.** When a service degrades at 2 a.m., does the on-call engineer see a single healthy-or-not verdict in seconds, before consulting a dashboard?
3. ☐ **Budget.** Do release and rollback decisions rely on error budget as a matter of routine?
4. ☐ **Coverage.** Are SLOs defined for a new service in the same pull request that moves it to production?
5. ☐ **Time.** Has every objective been reviewed since the service it describes last materially changed?
6. ☐ **Ownership.** Does every objective have a named owner who would notice if it went stale?
7. ☐ **Machine access.** Can an AI agent query reliability state before acting on production, through an interface built for programs?

Scoring: zero to two yes answers means reliability is reactive and incidents are the discovery mechanism.

Three to five means the contract exists in places, and drift and coverage gaps are the risk.

Six or seven means judgment scales with the system and autonomy can widen safely.

Conclusion

AI has removed most of the cost of writing code. Knowledge of whether a change to the codebase is safe still belongs to humans. AI has not unlocked massive ROI gains because the bottleneck in the process has been moved, not solved. The problem is structural because reliability practices, evolved and honed over years of experience, were designed for an era when writing code was the tallest order. It's measurable: 3.4 times the incidents per change (Faros AI, 2026), twice the share of engineering time going to reliability work (Inngest, 2026), and zero large organizations very confident at the next level of scale (Inngest, 2026). And it is still widening as adoption climbs: 86% of organizations use coding agents, so the volume of change has room to run.

The response the evidence points to is specific and largely available today: reliability targets written down in a form machines can evaluate, budgets that turn remaining risk into a continuously updated number, coverage that ships in the same pull request as the code, review practices that keep targets current, and interfaces that let agents consult the same information people do. Good reliability practices ensure that investments in AI will increase velocity. AI has opened the floodgates to a deluge of software. The open question is whether the judgment layer keeps up. In the data so far, the teams reporting confidence are the ones already treating that layer as infrastructure.

Sources

- Inngest, “AI in Production: The 2026 Benchmark Report.” Survey of 130 backend, full-stack, and AI engineers. All Inngest figures cited throughout are drawn from this report. Available at inngest.com/blog/ai-in-production-report-2026.
- Anthropic, “The 2026 State of AI Agents Report.” Survey of 500+ technical leaders, conducted late 2025 with research firm Material. Available at resources.anthropic.com.
- Temporal, “AI reliability is a decade-old problem,” April 2026. Compound reliability math for multi-step agentic workflows. Available at temporal.io/blog/ai-reliability-is-a-decade-old-problem.
- Faros AI, “The AI Engineering Report 2026: The Acceleration Whiplash.” Telemetry from 22,000 developers and 4,000+ teams; incident rate per code change and review-coverage trends during AI coding assistant adoption. Available at faros.ai/research.

All statistics in this report are attributed to their source at point of use. Survey figures reflect the respondent populations and time frames of the original studies.

ABOUT NOBL9

Nobl9 has spent years helping engineering teams put the practices in this report to work: defining what reliable enough means for each service, measuring it continuously across the telemetry those teams already run, and keeping the definitions current as systems change. The newer work is teaching AI to carry more of that load. Teams can now draft SLOs through a guided discovery conversation, get an explanation of why an error budget is burning, and give their AI agents direct access to reliability state through Nobl9’s MCP server. The destination is what we call continuous automated reliability: reliability defined, measured, and kept current as a property of the system, at whatever speed the system changes. More on the platform at nobl9.com, and on the AI work at nobl9.com/ai.

NOBL9.COM · NOBL9.COM/AI · HELLO@NOBL9.COM