

Lessons from Gitar Co-Founder on Engineering Trust into AI-Generated Code

Lessons from Gautam Korlam on building the context, reliability and evaluation loops needed to review AI-generated code at machine speed.

30 SEPT 2026 · 8 MIN READ



Darren Sim

Senior technology & product leader

All content published here is shared in a personal capacity. Nothing on this site should be interpreted as an official statement, endorsement, or communication from my employer.

As AI coding agents become more capable, generating code is no longer the hardest part. The greater challenge is ensuring that software remains reliable as the volume and velocity of change increase.

I had the opportunity to attend [AI-Generated Code == Code You Can Trust?](#), an AI Singapore event held at the Singapore Economic Development Board. Gautam Korlam, Principal Engineer at [Sonar](#) and co-founder of [Gitar](#), shared how engineering teams can prepare for software delivery at machine speed. Drawing on his experience with developer infrastructure at Uber and AI-native code review at Gitar, he offered a practical look at the engineering required around the model.

The central lesson was clear: a capable model alone does not produce dependable software. Trust comes from the **harness surrounding it** — the **context** supplied, the **tools** available, the **checks** enforced and the **feedback** captured.



Image note: This image was enhanced using generative AI based on an original photograph taken at the event.

Human–Scale Processes Are Meeting Machine–Scale Change

Traditional continuous integration (CI) and code review systems were designed around human-scale pull request volumes. AI coding agents are changing that assumption. Gitar had reportedly observed an approximately 10x increase in pull request volume over two years, with background agents and software factories adding further pressure to existing delivery systems.

Across the industry, recent studies paint a varied picture of how AI-assisted development affects engineering throughput:

Faros AI: A study of teams with high AI adoption found that they merged 98% more pull requests, while review time increased by 91% — highlighting the growing pressure on review capacity as coding velocity increases. [Read the study](#)

GitHub & Accenture: Their enterprise study reported an 8.69% increase in pull requests, alongside a 15% increase in merge rate and an 84% increase in successful builds. [Read the study](#)

METR: A randomized controlled trial found that experienced open-source developers took 19% longer when using early-2025 AI tools on familiar repositories, illustrating how outcomes can differ by task, team and environment. [Read the study](#)

These trends suggest that AI can increase **code-production velocity**, although its impact varies considerably by context. As more code moves through the delivery pipeline, the pressure shifts downstream: **review, testing and integration** will also need to keep pace. In this environment, assessing an **isolated diff** will become increasingly insufficient.

Consider an API that changes its representation of a price from cents to a currency type. The implementation may be correct within one repository, while a driver or customer application elsewhere continues to consume the old field. A useful review harness will need to identify this **cross-repository blast radius**, determine which teams are affected and verify that the change satisfies its **linked requirements** without introducing work beyond the intended scope.

Code review is therefore evolving from the identification of local defects into a broader assessment of system-wide impact.

KEY LEARNINGS FROM GAUTAM KORLAM'S SESSION

From AI-generated code to code you can trust

Four lessons on building dependable software with AI, from Gautam Korlam (Principal Engineer, Sonar).



01

Context is paramount, but more isn't always better.

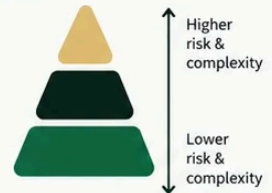
High-quality context (e.g. cross-repo usage, team rules, linked requirements) helps catch real issues. Too little misses problems; too much dilutes focus and increases cost.



02

Model selection should match the task.

Use smaller models for simple, focused changes, and reserve frontier models for high-risk work like contract or API changes. Think in terms of cost-per-task, not cost-per-token.



03

Reliability needs to be engineered.

Use provider fallbacks, detect retry loops, and surface uncertainty to users instead of ignoring unverifiable issues. A reliable harness anticipates failures and recovers gracefully.



04

Evals should form a continuous feedback loop.

When a developer flags an incorrect comment, it gets classified, added to the eval set, and used to improve prompts and context. Over time, evals compound, leading to fewer issues and higher quality.



KEY TAKEAWAY

Buy the harness, but own the context.

The underlying orchestration, routing and agent infrastructure will continue to evolve rapidly. What differentiates an organization is the knowledge it injects, the rules it enforces and how effectively it learns from developer feedback.

Context Is Paramount, but More Is Not Always Better

The quality of an agent's output is shaped by the quality of its context. A code diff is the minimum. Team rules, linked work items, cross-repository dependencies and previous developer feedback can all improve the agent's ability to reason about a change.

However, there are two sides of a coin. Too little context can lead to missed defects; too much can dilute the agent's focus, increase cost and create pressure on the context window. The answer is not to send everything.

A more deliberate approach is to preload the high-value information that almost every review needs — such as the diff and relevant engineering rules — while allowing the agent to retrieve supplemental context only when required. For a large CI failure, this may mean providing a representative log sample rather than every available log line.

Good context engineering is selective, not exhaustive.

Match the Model to the Risk

This leads to another useful design principle: avoid hardcoding model names into the harness. Abstract tiers such as light, medium and best allow teams to change providers without redesigning the workflow.

Model choice is only one part of the cost equation. Five factors influence agent cost: **token mix, model choice, caching, the number of turns and the context supplied.**

03 — KEEPING COST DOWN

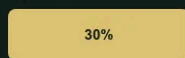
Five things drive agent cost.

01

Token mix

How much goes in, how much comes out and how much of the input was already cached.

NEW INPUT



30%

CACHED INPUT



50%

OUTPUT



20%

The mix matters because each token type is priced differently.

02

Model choice

Token rates vary by model.

03

Caching

Repeated input can be billed in full or reused.

04

Turns

Each turn resends everything that came before it.

05

Context

What we put in front of the model.

HARNESS ENGINEERING

COST 23 / 33

Retouched version of the “Five Things Drive Agent Cost” slide presented by Gautam Korlam.

These factors are closely connected. Larger contexts increase input-token usage, while additional turns may repeatedly resend the conversation history. Effective caching can reduce that burden, but its value depends on how much of the input remains reusable between calls.

Cost per token is therefore an incomplete measure. A less expensive model that needs many attempts, repeated tool calls and human intervention may cost more than a capable model that completes the task in fewer turns. What it comes down to is **cost per successful task**.

Gitar had reportedly achieved an approximately 50x reduction in cost per pull request since launch. The techniques were practical: reduce unnecessary turns, preserve prompt caching and prioritize the most important files instead of sending everything.

Reliability Needs to Be Engineered

AI systems will encounter unavailable providers, failing tools and uncertain evidence. A production harness should expect these conditions rather than treat them as exceptions.

Provider routing can enable a workflow to fall back across services when one becomes unavailable. Retry-loop detection can identify when a model repeatedly calls the same failing tool, inject guidance to try a different approach and stop or escalate when necessary.

Just as importantly, the system should surface uncertainty. If an agent cannot verify an issue, it is better to tell the developer than to silently ignore it or present an unverified conclusion with false confidence.

Reliability is not a property of the model. It is an outcome of the system design around it.

Specialized Agents Can Keep the Loop Focused

Gitar's architecture reportedly began with a monolithic agent responsible for the entire review. Over long-running pull requests, this created context bloat, higher cost and compaction.

The design evolved toward an event-driven architecture. Events from GitHub or GitLab — such as pushes, CI failures and labels — could be routed to specialized agents focused on areas including security, testing, performance and cross-repository impact. This ensemble of focused agents performed better than a single generalist carrying the entire problem in one context.

The lesson extends beyond code review. Agentic systems should decompose work around clear responsibilities rather than assume one large agent can reliably do everything.

Evals Turn Feedback into Compounded Learning

An effective harness does not stop when it produces a review comment. It learns from what happens next.

When a developer identifies an incorrect comment, the issue can be classified, traced back to its prompt or context and added to the evaluation set. Future changes to prompts, models and context can then be tested against these cases before release.

This turns individual mistakes into organizational learning. Over time, the evaluation set becomes a living record of where the system has failed and what dependable performance should look like.

Without evals, teams are left with demonstrations and anecdotes. With evals, they have a feedback loop.

Harness Can Be Bought; Context Cannot

The most thought-provoking takeaway was the recommendation: **buy the harness, but own the context**. The framework he shared draws a useful boundary between the capabilities an organization can source from a platform and those it should continue to shape for itself.

04 — BUILD OR BUY

Buy the harness, own your context.

Experience	Where people see and act on results	BUY
Orchestration	Which agent runs, and when	BUY
Control plane	Access and budgets	BUY
Agents	Specialist roles and their instructions	BUILD IF UNMET
Runtime	Context, tools, routing, tracing MODEL	BUY
Context	Business rules and what the team knows	KEEP
Tools and integrations	Code hosts and CI	BUY

● HARNESS ENGINEERING
OWNERSHIP
32 / 33

Retouched version of the “Buy the harness, own your context” slide presented by Gautam Korlam.

The baseline infrastructure — including orchestration, model routing, caching, agent loops and provider fallback — is evolving quickly and can be expensive to maintain. A platform can provide much of this control plane, allowing organizations to focus on the knowledge that differentiates their engineering practice: the context they inject, the rules they enforce, the agents they activate and the way they learn from developer feedback.

However, this still requires judgment. Teams will need to evaluate capabilities such as cross-pull-request conflict detection, functional validation against requirements, safe approval policies and configuration that developers can understand. There is no silver bullet. As AI accelerates software delivery, our engineering discipline around it will need to mature just as quickly. Trustworthy AI-generated code will depend not on access to a model alone, but on the context, controls and feedback loops engineered around it.



— THE AUTHOR

Darren Sim

Darren is a senior technology and product leader based in Singapore. He writes about the decisions, systems, and people behind meaningful transformation across Asia-Pacific.

[More about Darren](#) ↗

[Connect on LinkedIn](#) ↗