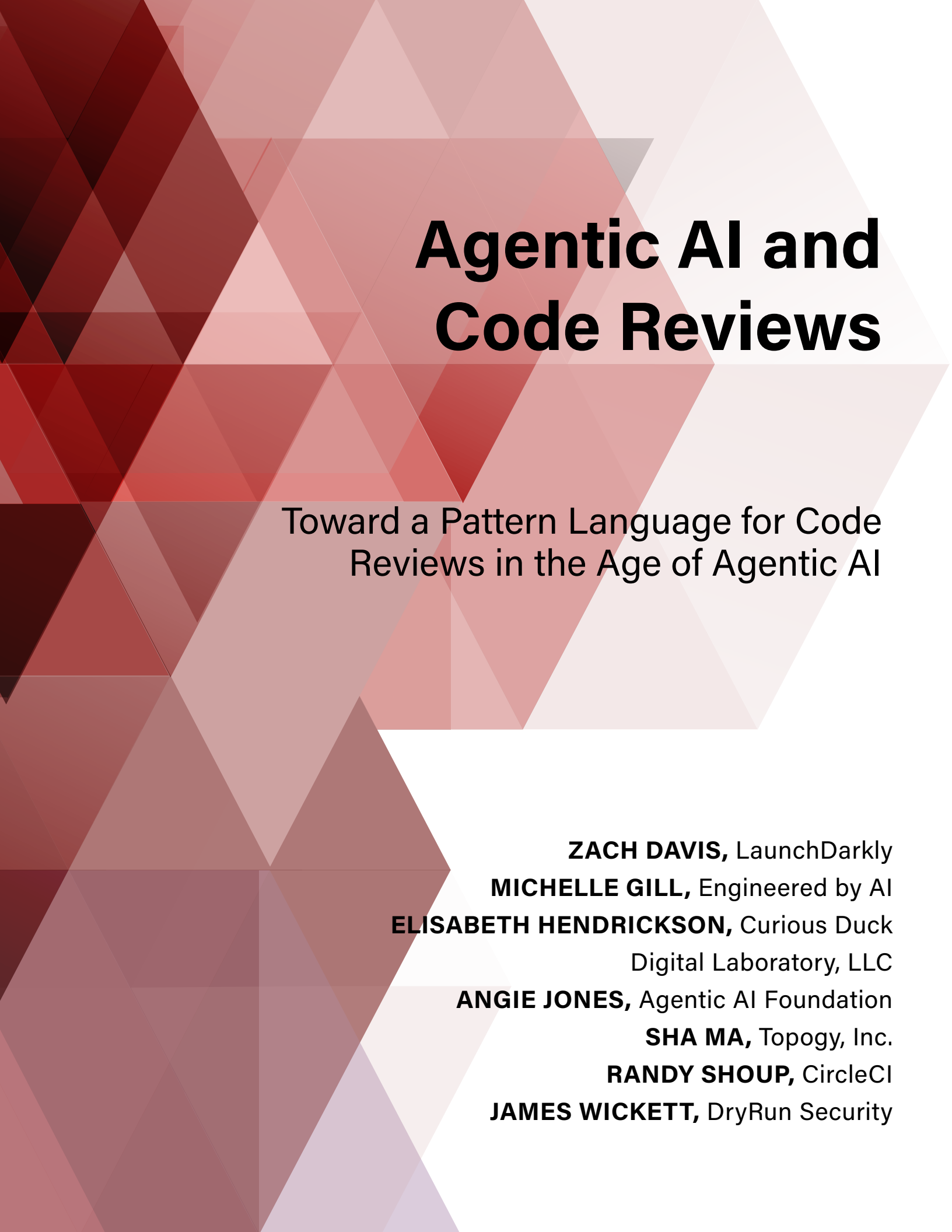




# Agentic AI and Code Reviews

Toward a Pattern Language for Code  
Reviews in the Age of Agentic AI

**ZACH DAVIS**, LaunchDarkly

**MICHELLE GILL**, Engineered by AI

**ELISABETH HENDRICKSON**, Curious Duck

Digital Laboratory, LLC

**ANGIE JONES**, Agentic AI Foundation

**SHA MA**, Topogy, Inc.

**RANDY SHOUP**, CircleCI

**JAMES WICKETT**, DryRun Security



25 NW 23rd Pl  
Suite 6314  
Portland, OR 97210

Enterprise Technology Leadership Journal  
Fall 2026  
Volume 3, Issue 2  
Copyright © IT Revolution 2026

ISSN: APPL100010486

All works herein are licensed under the Creative Commons Attribution-Non  
Commercial-ShareAlike 4.0 International License. To view a copy of this license,  
visit <http://creativecommons.org/licenses/by-nc-sa/4.0/>.

When sharing this content, please notify  
IT Revolution Press, LLC, 25 NW 23rd Pl, Suite 6314, Portland, OR 97210

Produced in the United States of America

AI tools may have been used to assist with language and editorial refinement  
in the papers in this journal under the authors' direction. All ideas, frame-  
works, and content originate from the authors' expertise and have been  
reviewed by IT Revolution's editorial team prior to publication.

For further information about IT Revolution, these and other publications, special  
discounts for bulk book purchases, or for information on booking authors for an  
event, please visit our website at [ITRevolution.com](http://ITRevolution.com).

To view past issues, please visit  
[ITRevolution.com/Resources](http://ITRevolution.com/Resources).

The Fall 2026 *Enterprise Technology Leadership Journal* is sponsored by



# Agentic AI and Code Reviews

## Toward a Pattern Language for Code Reviews in the Age of Agentic AI

*Zach Davis, Michelle Gill, Elisabeth Hendrickson, Angie Jones, Sha Ma, Randy Shoup, James Wickett*

### INTRODUCTION

A typical software delivery process involves developers committing changes to a branch and then submitting a PR<sup>1</sup> that triggers one or more code reviews. These code reviews act as a quality gate, preventing substandard code from being merged to the main branch. They also support knowledge sharing, enabling team members to stay up to date with the contents of a rapidly evolving codebase.

Although developers value code reviews, an in-depth review takes time, time that comes at the expense of other development activities. Even before AI, the PR process was a common bottleneck. With AI accelerating both the rate of change and sheer volume of code, manual code reviews for all change sets have become unsustainable. As one technical leader told us, “We were already struggling to keep up, and now our senior engineers receive dozens of merge review requests per day.”

Additionally, there is also a concern that without human supervision, agents are too likely to produce code that is overly complex or subtly wrong. In response, organizations are seeking better approaches to code review, asking questions like: How do we keep up? How do we leverage AI to streamline the process without increasing risk? And what does all of this mean for human-in-the-loop review strategies?

In this paper, we propose a set of patterns to reduce the manual burden of code reviews and apply AI as part of a layered verification strategy that includes deterministic checks, agentic reasoning, contextual policy evaluation, and selective human judgment.

---

<sup>1</sup> In this paper, “change set” refers to a set of code changes, whether or not those changes are ready to be merged into the main, or trunk, branch. Change set is synonymous with “diff,” “commit,” etc. “PR,” or pull request, is a proposal to merge a change set into the main, or trunk, branch. PR is synonymous with “merge request.”

## DEFINING CODE REVIEW

We define *code review* as an examination of source code by an entity that is not the author, with the intention of making a judgment that leads to an action. All three elements of this definition are important:

- **“An examination of source code”** allows the reviewer to judge the implementation, both what it does and how it is structured. A reviewer might refer to a specification or other artifact that addresses intentions, but intentions aren’t enough. They may also execute the code to verify its behavior, but simply exercising it is insufficient to judge how it is expressed and organized.
- **“An entity that is not the author”** refers to two ideas. First, the purpose of a code review is to have external perspectives on the implementation other than that of the author(s). Second, in the age of AI, those perspectives may come from an agent rather than a human.
- **“Making a judgment that leads to an action”** refers to the outcome of a review. Where a code review is part of a PR process, the action may be to perform the merge, reject the change, or provide feedback that must be incorporated before the change can be merged. This notion of making a judgment and taking action distinguishes a code review from a casual reading.

## THE VALUE OF CODE REVIEWS

Code reviews are not free. They introduce significant costs, including increased lead time due to delays from waiting for reviews and from context switching; added effort as reviewers spend hours on the process, reducing their time for deep work; and the need for specialized expertise, as effective reviewers need system, domain, and social knowledge. Before we consider how to adapt code review practices in the age of AI, we must first ask whether the value these reviews offer outweighs the friction they introduce. Fortunately, there are empirical studies that point to the actual value delivered by code review.

A Microsoft study of modern code review—one of the foundational empirical works in the field—determined that while finding bugs is a key reason people cited for performing

code reviews, there were multiple other benefits, including identifying code improvements not linked to bug fixes, finding alternate solutions, and knowledge sharing.<sup>2</sup>

Further, code reviews may be over credited as a defect-finding mechanism but under credited as a risk triage mechanism. Most review activity is devoted to maintainability, style, and knowledge sharing—subjectively increasing the code quality to decrease defects—with only a minority of comments or review-induced changes directly addressing functional, potentially blocking defects. In particular, a 2009 IEEE paper found that the majority of findings in code reviews related to “evolvability” risks.<sup>3</sup> Another large study in 2014 of reviews in open-source projects came to a very similar conclusion.<sup>4</sup>

It is clear that code reviews have value, and that value extends beyond identifying functional flaws in a given code submission. As agentic AI reshapes how code is written, making infeasible the traditional approach that forces all changes to be reviewed by humans, how do we preserve the value of code reviews?

## HOW DOES AGENTIC CODE REVIEW WORK?

The idea of agentic code review inevitably raises a question: *Can we trust an agent’s judgment?* We propose reframing the question as: *How do we structure our processes so trust is earned rather than assumed, confidence is delivered reliably, and evidence is provided to ensure it?* Human code review persists not solely because of its effectiveness but because of three interlocking forces that agentic systems have not yet fully displaced: trust, compliance, and accountability.

Agentic code review belongs inside the same generation → verification → release → learn loop we use today, but its placement and responsibilities change. Instead of acting as a late, human-only gate at merge time, agentic review should run continuously and proportionately to risk. It should be close to the author during development (IDE, pre-commit, pre-push) as a structured gate before merging to protected branches. Additionally, it should act as an asynchronous post-merge analysis to identify issues, regressions, or policy drift

---

2 Alberto Bacchelli and Christian Bird, “Expectations, Outcomes, and Challenges of Modern Code Review,” in Proceedings of the 35th International Conference on Software Engineering (ICSE2013) (Piscataway, NJ: IEEE, 2013), 712–721. <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/ICSE202013-codereview.pdf>.

3 Mika V. Mäntylä and Casper Lassenius, “What Types of Defects Are Really Discovered in Code Reviews?” *IEEE Transactions on Software Engineering* 35, no. 3 (May/June 2009): 430–448. <https://doi.org/10.1109/TSE.2008.71>.

4 Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens, “Modern Code Reviews in Open-Source Projects: Which Problems Do They Fix?” in Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014) (New York: ACM, 2014), 202–211. <https://doi.org/10.1145/2597073.2597082>.

not caught earlier in the pipeline. High-risk systems generally require strict pre-merge verification and should prevent undeployable code from accumulating in protected branches. This better distributes review signals across the delivery life cycle.

Different stages of the life cycle demand different agentic review behaviors. During authoring, agents should optimize for fast feedback and rapid developer iteration. Before merging, agents should enforce repository policy, architectural constraints, regression detection, and organizational standards. After merging, agents should monitor for system-level effects that were not visible in isolation, especially in distributed or high-risk environments. Before production promotion, agentic review serves as a release governance function, evaluating operational risk, rollout safety, compliance requirements, and unresolved uncertainties.

To safely remove humans from the critical path, three conditions must hold:

- First, as much validation as possible must be shifted into deterministic checks—such as tests, static analysis, policy-as-code, and reproducible evaluation pipelines—so agents are focused on the residual subjective space rather than rediscovering objective failures that could have been proactively identified.
- Second, the review stack must be risk-tiered along a spectrum, where the degree of human involvement is calibrated to the risk profile of the change and the system it affects, ranging from none through lightweight spot checks to explicit authorization.
- Third, every agentic decision must produce a durable, inspectable record of what was evaluated, under which policies, with what confidence, and by whom, so organizations can preserve accountability, audit decisions, debug failures, and continuously improve the behavior of their review systems.

In practice, this means agentic reviews should run early and often, in small loops with the author, escalating only when confidence is insufficient for the specific change, system, or contributor context. “No human in the loop” should be treated as a configuration choice justified by risk and evidence, not as a default. That is the North Star to drive toward.

## PATTERNS FOR AGENTIC CODE REVIEW

Organizations vary significantly in system criticality, regulatory constraints, engineering maturity, release practices, and tolerance for automation risk. There is no single “correct”

implementation model for agentic code review. Legendary architect Christopher Alexander's books *The Timeless Way of Building* and *A Pattern Language: Towns, Buildings, Construction* were written for those who design physical spaces, and his work also directly led to the "patterns movement" within the software community. Alexander and his coauthors described the purpose of patterns this way:

*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*<sup>5</sup>

Inspired by Christopher Alexander's words and the larger software patterns community, we've framed our recommendations as modular and independently composable approaches rather than a single set of prescriptive solutions. These patterns should, therefore, be viewed less as a maturity ladder and more as a toolbox. Teams should evaluate them individually, adopt them incrementally, combine them selectively, and evolve them over time to suit their context and as their confidence in automated verification grows.

Many of these patterns build on practices that predate generative AI. In our experience, the organizations most successful with agentic review are those that have already heavily invested in deterministic engineering practices, such as automated testing, static analysis, continuous integration, strong typing, architectural clarity, and small change sets. In agentic development environments, these practices are even more important because they directly affect the ability of humans and AI systems to reason safely and consistently about changes. Collectively, these patterns represent a shift from code review as a single, human-centered merge gate toward a layered verification system that leverages AI along with selective human judgment.

## Pattern: Small Incremental Changes

One of the oldest software engineering practices becomes even more important in an AI-assisted development environment: keeping changes small. Large changes are not only hard for humans to review but also challenging for AI reviewers to reason about. The smaller the

---

<sup>5</sup> Christopher Alexander, Sara Ishikawa, and Murray Silverstein, with Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel, *A Pattern Language: Towns, Buildings, Construction* (Oxford University Press, 1977), x.

surface area of a change, the easier it is to understand what it does, whether it is safe, and what went wrong if something breaks.

AI agents make the problem of large changes worse before they make it better. Left to their defaults, AI code generation tends to produce large, synthesized change sets that mix mechanical refactoring, behavioral changes, dependency updates, and formatting all at once. That combination overwhelms reviewers—human and machine—while burying the parts of the change that actually matter.

The **Small Incremental Changes** pattern introduces smaller units of risk by separating refactors from behavioral changes, isolating dependency updates and generated boilerplate from functional modifications, and prompting agents to produce reviewable intermediate commits rather than a single large change set at the end. Intermediate commits are particularly critical when following a dark software factory pattern, in which AI operates unsupervised in loops. Soft thresholds on change set size, file count, or subsystem breadth should trigger decomposition before review even begins. Feature flags and progressive delivery additionally decouple how much code ships from how much actually turns on.

The Small Incremental Changes pattern also makes it easier to roll back, to bisect, and to correlate with regressions when something goes wrong in production. The human's job here is mostly configuration and policy: defining the thresholds, setting the decomposition conventions, and reviewing the shape of a change set rather than manually splitting it.

This pattern makes the Shift Left pattern more effective. Less surface area means faster, more reliable pre-commit and pre-push checks. It also makes the Continuous Code Quality Loop pattern work better, since tightly scoped changes give the author and review agents a cleaner target to converge on.

## Pattern: Shift Left

As software pipelines mature, organizations increasingly move deterministic verification earlier in the software life cycle—from post-commit review and CI back toward the moment code is written. Without shifting left, human reviewers and advanced AI models spend expensive attention rediscovering problems that deterministic tools could already detect—formatting violations, type errors, dependency policy breaches, schema mismatches, and many classes of security and correctness issues. This wastes the scarcest resource in the review process: judgment applied to problems that actually require it.

This **Shift Left** pattern applies wherever deterministic checks exist but are run at merge time, in CI, or not at all. It does not apply to concerns that genuinely require judgment,

such as architectural trade-offs, ambiguous requirements, operational risk, and business intent. Those remain the domain of human and agentic reviewers.

Several forces shape the decision to shift verification earlier.

- First, human attention and advanced model reasoning are both expensive; every objective concern removed from review through automation frees capacity for “fuzzier” problems.
- Second, feedback is most effective while authors still have context. A formatting violation caught in the IDE costs seconds, but the same violation caught in review costs a context switch and an additional code-verify-merge loop.
- Third, deterministic checks become faster and more reliable when changes are small and focused, since large change sets make static analysis noisy and slow. At the same time, teams resist pre-commit gates that feel like friction, so the checks must be fast and their value immediately visible, or developers will route around them.

The approach here is to encode objective verification into the tools and pipelines closest to where code is written—IDE integrations, pre-commit hooks, local review agents, CI gates—and run them continuously during authoring rather than at review. In this scenario, PRs that fail these checks are rejected before a reviewer ever sees them. Repository-specific guidance (style rules, naming conventions, architectural constraints) lives in machine-readable artifacts for agents to access during authoring rather than discover at review.

The human’s role in the Shift Left pattern also shifts accordingly. Rather than catching mechanical issues during review, the human configures which checks run at which stage, sets policy for what blocks submission versus what warns, and maintains the machine-readable guidance artifacts. When a new class of recurring review feedback emerges, the human decides whether it can be encoded as a deterministic check and shifted left. Over time, the review surface gets quieter. What arrives at the review stage has already cleared the objective bar, and the reviewer focuses entirely on judgment calls.

The Shift Left pattern works best when the Small Incremental Changes pattern is already in place, as focused change sets make deterministic checks faster and more reliable. It is also a prerequisite for the Continuous Code Quality Loop pattern, which depends on mechanical concerns being handled automatically so the review agent can reason rather than rediscover violations.

## Pattern: Design for Verification

Agentic review systems are substantially more effective in environments where software behavior is already designed to be verifiable. Practices such as Test-Driven Development (TDD), Behavior-Driven Development (BDD), contract testing, executable specifications, and acceptance testing workflows all increase the amount of evidence available to both humans and agents during review.

The core advantage of these practices is the reduction of ambiguity and the ability to make intent executable. When expected behavior is expressed explicitly and executable validation is present alongside the implementation, review systems spend less effort inferring intent from code alone. This becomes particularly important in AI-assisted development environments, where implementation can be generated rapidly but intent is likely underspecified.

In practice, this means defining acceptance criteria before implementation begins, so that the generated code has explicit behavioral targets. Service expectations should be expressed as contract tests and schema validation rather than implicit assumptions. Production regressions should be captured as replayable test cases. Representative datasets, edge cases, and failure scenarios should live alongside the codebase so review agents can reason against real operational conditions rather than hypothetical ones. The generated code should produce or update verification artifacts as part of implementation, not as a separate follow-up task. Flaky tests undermine all of this, so test reliability becomes infrastructure, no longer a nice-to-have.

Over time, this shifts review away from asking “Does this code appear reasonable?” toward “Does the implementation satisfy the explicitly defined behavioral expectations?” Review moves upward from implementation inspection toward validating assumptions, edge cases, failure modes, and business outcomes. The distinction here matters because subjective review scales poorly, while executable validation scales comparatively well.

The Design for Verification pattern reinforces both the Shift Left and Continuous Code Quality Loop patterns. Deterministic checks run earlier and with more confidence when behavioral expectations are already encoded. The agent performing the review has more to work with and less to infer.

## Pattern: Deterministic Engineering Practices

Agentic review is fundamentally a reasoning problem. The more implicit the environment becomes, the more effort reviewers must spend reconstructing assumptions before evaluat-

ing whether a change is safe, correct, or aligned with organizational expectations. In highly dynamic systems, substantial review effort is often consumed in simply discovering how the system behaves before any judgment about the change itself can begin.

The organizations most likely to achieve high-confidence autonomous reviews are, therefore, often not those with the most advanced models, but those whose systems are easiest to reason about mechanically. Repositories with consistent conventions, explicit ownership, reproducible environments, and well-defined interfaces tend to produce significantly better outcomes for both human and agentic reviewers. The goal is to constrain the problem space so that fewer dangerous states are expressible in the first place.

In practice, **Deterministic Engineering Practices** mean preferring explicit schemas and typed interfaces over loosely structured data exchange; reducing hidden side effects and implicit framework behavior; and making builds, environments, and dependency resolution reproducible across developer and CI environments. Repository structures and architectural conventions should be standardized across teams. Service ownership, dependency relationships, and operational boundaries should be encoded in machine-readable formats. In distributed systems, idempotent operations and observable state transitions are easier to reason about than those with hidden or irreversible consequences. Abstractions that require deep institutional knowledge to use safely—the kind where a new team member can easily make a dangerous mistake—should be treated as liabilities.

All of the same characteristics that make systems easier for humans to maintain also make them easier for agents to review. Clear boundaries, explicit ownership, reproducible behavior, and constrained interfaces improve verification outcomes regardless of whether the reviewer is human or machine.

The Deterministic Engineering Practices pattern reinforces all other patterns. The less implicit the environment, the more effective Shift Left checks become, the faster the Continuous Code Quality Loop converges, and the more reliable Design for Verification practices are.

## Pattern: Context Engineering

Agentic code review systems are only as good as the context they can reliably access. Human reviewers compensate for missing information through institutional memory, tribal knowledge, hallway conversations, and intuition built over years of experience. Agents do not have that luxury. If the information required to evaluate a change is not explicitly available, structured, and retrievable, the agent will make poor assumptions or fail to reason about the change altogether.

This creates a new prerequisite for effective agentic review: Codebases need to become agent-ready. This means that engineering environments need to evolve from repositories of code into repositories of operational knowledge, architectural intent, and historical decision-making.

The **Context Engineering** pattern is the practice of making that happen deliberately, as part of the software life cycle itself. It is not optional overhead added after adopting agentic systems. Organizations that attempt agentic review without investing in context infrastructure often find that the failure lies in a lack of organizational memory rather than in model intelligence.

At a minimum, review agents need access to far more than the PR itself. They require surrounding context, including:

- Repository structure and dependency relationships.
- Architectural constraints and service boundaries.
- API contracts and backward compatibility expectations.
- Coding standards and organizational conventions.
- Historical incidents and postmortems.
- Previous code review comments and accepted trade-offs.
- Security policies and compliance requirements.
- Documentation explaining why systems behave the way they do.

The challenge is ensuring agents can retrieve the right context at the right level of abstraction during evaluation.

Consider a human reviewer who instinctively recognizes that changing a seemingly harmless API field could break backward compatibility for an undocumented mobile client still running an older version in production. An agent will not make that connection unless compatibility rules, client dependencies, and rollout expectations are encoded somewhere retrievable. The gap is in what the model has access to, not in its reasoning ability.

In addition to policy, agents also need access to precedent. Human reviewers rarely evaluate code against rules alone. They evaluate against decisions the team has already made. Why was a similar approach rejected six months ago? Why does this subsystem intentionally violate a normal convention? Why is performance prioritized over readability in this specific path? That institutional reasoning needs to be captured somewhere that an agent can actually reach it.

Organizations that succeed with agentic review treat context as foundational infrastructure. This often includes knowledge graphs, indexed architecture documents, service cata-

logs, structured repository metadata, and standardized context files such as **AGENTS.md**. These provide curated pathways to the information that humans already implicitly rely on, rather than forcing agents to rediscover it from raw source code.

One emerging approach is to allow agents to learn directly from prior reviews by indexing merged PRs, reviewer comments, architectural discussions, and incident reports. Over time, this creates a feedback loop where review agents become progressively better at modeling how the organization itself makes engineering decisions.

Repositories that are easier for humans to understand also tend to be easier for agents to reason about. Clear boundaries, explicit ownership, deterministic testing practices, strong typing, high-quality documentation, and consistent conventions all improve agentic review outcomes. In this sense, repo readiness strongly overlaps with general engineering maturity. The investment pays off in both directions.

Humans shift from manually inspecting every implementation detail to defining policies, maintaining architectural knowledge, encoding operational expectations, and improving the information substrate agents rely on. The role is less that of a reviewer and more of a curator: ensuring the system has what it needs to make good judgments on its own.

The Continuous Code Quality Loop, Adversarial LLM Review, and Multi-Agent Reviews patterns depend on Context Engineering being in place. An agent operating without a well-structured context will surface noisy, shallow findings regardless of how sophisticated the review logic is. Get the context right first, and the downstream patterns become substantially more effective.

## Pattern: Continuous Code Quality Loop

Typically, code review feedback arrives too late in the delivery cycle. Defects and violations discovered at merge time require context reconstruction, interrupt downstream work, and impose coordination overhead on both authors and reviewers. As AI agents generate code at increasing velocity, this latency compounds: The gap between when a problem is introduced and when it is caught grows larger, and the cost of late-stage rework increases proportionally.

The **Continuous Code Quality Loop** pattern addresses this by running a review agent continuously alongside the author agent while code is being written. The review agent catches issues, suggests fixes, and checks that those fixes actually worked, all before the code is ever submitted as a PR. Rather than a single gate at the end, review becomes an ongoing cycle embedded in the authoring process itself.

This pattern works wherever code is actively being written, and feedback can arrive during or right after the authoring step, not hours later. It's most critical in AI-assisted workflows, where an author agent produces code at a pace no human reviewer can match.

The loop runs at multiple points: inside the IDE while code is being written, at pre-commit before changes are staged, and at pre-push before they're shared upstream.

Whether it's a substitute or a complement depends on the risk tier and what's been configured to run in the loop. But in either case, the goal is the same: By the time code reaches the next stage, the obvious problems are already gone.

A few real tensions come up in the practice of this pattern. Feedback has to be fast. If the review agent noticeably slows things down during live coding, developers (or agents) will turn it off. But feedback also has to be substantive. An agentic loop that only catches formatting issues isn't pulling its weight; a linter could do that.

The loop also needs to know when it's done. That means verifying that fixes actually worked, not just suggesting a change and moving on. It means not going looking for new problems once old ones are resolved. Design for Verification upfront so the loop hands off rather than spins.

The Continuous Code Quality Loop pattern is implemented as an autonomous cycle between two cooperating agents: an author agent responsible for producing or modifying code and a review agent responsible for evaluating that output against a defined set of quality dimensions and returning structured, actionable feedback.

The loop operates as follows:

1. The author agent produces or modifies an implementation artifact: a function, a class, a change set, or a commit.
2. The review agent examines the artifact against applicable quality dimensions:
  - » Functional correctness: Does the implementation satisfy the stated intent or acceptance criteria?
  - » Security posture: Are there injection vectors, insecure defaults, or policy violations?
  - » Adherence to coding standards and architectural conventions.
  - » Performance and resource usage concerns.
  - » Test coverage or behavioral verification.
3. The review agent returns structured feedback: specific findings, the location and nature of each issue, and a concrete remediation suggestion or corrected version of the affected code.

4. The author agent applies the suggested remediation or regenerates the affected artifact.
5. The review agent re-evaluates the updated artifact to confirm that the issue has been resolved and that the fix has not introduced new concerns.
6. The loop repeats until a convergence criterion is met: a clean review pass, a maximum iteration count, or a residual risk level acceptable for handoff to the next verification stage.

The review agent should focus on what it's actually good at: reasoning about intent, context, and behavior. Mechanical checks for concerns, such as formatting, type errors, and known vulnerabilities, should run in parallel via linters and static analysis tools. They should not consume the review agent's cycles. Reserve the agent's attention for the things only reasoning can catch.

In practice, this loop can be instantiated at multiple points in the development workflow:

- **During live authoring in the IDE:** As the author writes, review agents integrated into development environments can surface findings in-line within the file being edited or as side-panel suggestions. At this stage, the loop is lightweight and conversational, focused on immediate improvements rather than comprehensive review.
- **At pre-commit:** Before changes are staged, a more thorough review agent pass can run across the full change set, evaluating the change as a coherent unit and applying the organization's documented standards, security baselines, and convention rules. This is the point at which the loop should catch most issues that would otherwise surface in a formal review.
- **At pre-push:** Before code is shared upstream, a final pass by the review agent validates that the commit set is internally consistent, that no regressions were introduced across the accumulated changes, and that the change is ready for downstream CI and formal review stages.

In fully agentic environments, the human may not observe individual loop iterations at all. The loop runs autonomously between author and review agents. The human encounters the code only after the loop has converged—at the PR, during plan review, or in a post-hoc audit. In hybrid environments, the human participates selectively: reviewing a summary of what the loop found and resolved, inspecting residual findings that the loop

flagged but could not remediate with sufficient confidence, and adjusting the loop's configuration based on observed outcomes over time.

Humans also play a critical role in calibrating the review agent's quality dimensions and convergence criteria. The loop can only evaluate what it has been instructed to evaluate. If an important concern—a class of architectural violations, a specific security strategy, a business-rule constraint—is not encoded in the review agent's configuration or accessible context, it will not be caught. Humans who observe recurring issues in post-merge incidents or production regressions should update the loop's configuration to catch those issues earlier in future cycles. This makes the loop a mechanism for organizational learning, not just real-time quality improvement.

The Continuous Code Quality Loop pattern is primarily a correctness and quality mechanism operating within the author's own development cycle. It is not designed to catch issues that require external adversarial reasoning. A review agent checking an author agent's output can share some of the same assumptions the author made. The Adversarial LLM Review pattern addresses this by bringing in a separate, independent model as a reviewer, one that isn't correlated with how the author approached the problem.

For higher-risk changes, the Multi-Agent Reviews pattern goes further, bringing in specialist agents for security, performance, compliance, or domain-specific concerns running in parallel. The Continuous Code Quality Loop pattern is the first tier in that stack. It provides the highest-frequency, lowest-cost check that raises the baseline before code reaches anything more expensive.

## Pattern: Adversarial LLM Review

If a review agent is given the same set of assumptions (similar context, closely related model, same prompt structure) as the author agent, then it should be no surprise when both agents find the same logic plausible, accept the same insecure defaults, and overlook the same edge cases. They agree not necessarily because the author produced correct output, but because the reviewer's and author's inputs were the same. Critical issues survive not because the review agent failed to look, but because the review agent was structurally incapable of seeing what the author agent couldn't.

The **Adversarial LLM Review** pattern addresses this by introducing structural independence between generation and evaluation. A separate model running in a separate session with a separate prompt reviews code with no access to the reasoning that produced it. The review agent evaluates the artifact as a senior engineer unfamiliar with the author's approach would, or as an end user encountering the production system would. By using

a different model than the author agent, the review agent will “think differently” because it was trained on different data, with a different architecture, different Reinforcement Learning from Human Feedback (RLHF) tuning, and different optimization objectives. The review agent internalizes different priors about what “correct code” looks like, thereby making its review findings more objective and valuable.

This pattern can be applied wherever AI agents generate and review code within the same workflow. It is most important when the code touches security-sensitive paths, business-critical logic, or systems where failure is expensive. It applies less when strong deterministic checks already sufficiently constrain the solution space, or when the change is small and trivial, so that a single well-configured review agent can reason about it safely and completely.

The core concept and principle of structural independence has deep roots in AI research. The generative adversarial network approach (GAN), introduced by Goodfellow et al. in 2014,<sup>6</sup> pits two neural networks against each other in a zero-sum environment. A generator produces outputs and a discriminator tries to distinguish generated outputs from real ones. The generator tries to trick the discriminator, while the discriminator gets better at catching the generator. Through this adversarial process, the generator improves because the discriminator provides an honest signal about where its outputs fail. The critical insight is that the discriminator is a separate network with its own parameters. It operates independently to signal the generator’s weaknesses.

In the Adversarial LLM Review pattern, the review agent uses a separate model and functions as the discriminator, which provides an honest signal about where the generated code falls short, uncontaminated by the reasoning that produced it.

The Adversarial LLM Review pattern also draws on the Zero Trust security architecture, whose core tenet is that no artifact, contributor, or system should be trusted by default, regardless of origin. There are direct parallels we can draw between Adversarial LLM Review and Zero Trust security:

- **Explicit verification:** Zero Trust means every single access request must be explicitly verified, authenticated, and authorized independently each time. In Adversarial LLM Review, the review agent is designed to scrutinize the author agent’s output without relying on assumptions about the author agent or the reasoning behind it.

---

<sup>6</sup> Ian J. Goodfellow et al., “Generative Adversarial Networks,” *arXiv* (June 10, 2014). <https://arxiv.org/abs/1406.2661>.

- **Least privilege:** Zero Trust systems use data classification and strict isolation. Users are granted only the absolute minimum access required and are blocked from the rest of the network. Similarly, in Adversarial LLM Review, the review agent doesn't have access to the same context and artifacts that the author agent used to produce the code output.

Adversarial LLM Review operationalizes these core principles of Zero Trust: It treats every change as potentially flawed and evaluates it based on evidence rather than provenance.

In practice, an Adversarial LLM Review solution can take several forms:

- **Cross-model review** uses one model for generation and another for review. Different model families tend to make different categories of errors: a logic error that one model finds plausible, another may flag as suspicious.
- **Cross-session review** uses the same model, but runs the author agent and review agent in completely separate sessions with no shared context, no shared conversation history, and a fresh system prompt oriented toward critical evaluation. Because the review agent reviews the code without the reasoning chain from the author agent, even using the same model, the two instances are completely separate sessions, which will catch more issues than one because context isolation removes the specific blind spots and assumptions introduced during generation.
- **Structured adversarial prompting** instructs the review agent to assume the code contains errors, to look for specific failure modes that AI-generated code is known to produce (logic errors, missing edge cases, incorrect assumptions about API contracts, etc.), and to evaluate the code against explicit acceptance criteria.

A few tensions to consider in adopting Adversarial LLM Review:

- **Signal versus noise:** If the review agent generates too much noise and false urgency over non-issues, developers will stop trusting its findings. The sensitivity should be calibrated over time based on positive and negative examples, of which findings correlated with actual production incidents and which ones were non-issues.
- **Cost and latency:** A second model pass is not free. Organizations should apply Adversarial LLM Review proportionally to risk. For example, reserving the use

of Adversarial LLM Review for complex logic updates, security-sensitive paths, but not for low-risk changes in well-tested systems.

Reviewer agents can run within the Continuous Code Quality Review Loop and produce structured findings with severity, location, and the reason why each finding represents a genuine risk. Findings can then be triaged and often automatically resolved by the author agent. Findings that are unresolved after several attempts may be escalated to a human. Humans calibrate the adversarial reviewer's sensitivity and focus, review findings that remain unresolved after the author/reviewer exchange, and decide whether the residual uncertainty is acceptable for the specific change and system.

When the Spec-Driven Development pattern runs upstream, the adversarial reviewer has a concrete specification to validate against. The question shifts from “Does this code look correct?” to “Does this implementation faithfully realize what was planned?” The Adversarial LLM Review pattern provides independent review and risk assessment without bias from the reasoning and prompt that led to the output. It applies a single generalist reviewer with a single evaluation lens trained to look at the system critically.

For specific risk considerations, the Multi-Agent Reviews pattern goes much further by decomposing the review into specialist concerns—security, performance, compliance, and domain correctness—each handled by a dedicated subject matter expert agent with a core focus and evaluation criteria. Where Adversarial LLM Review enables a structurally independent review agent to find problems the author agent missed, Multi-Agent Reviews ensures every dimension of risk is evaluated with sufficient depth.

## Pattern: Multi-Agent Reviews

A single reviewer brings a single perspective. It may be strong on security or thorough on correctness, but miss performance regressions or compliance implications. When a generalist agent is asked to review all dimensions simultaneously, it's prone to producing generic findings because its “attention” is not focused. The result is shallow coverage of many concerns rather than deep coverage of any specific concern.

This problem compounds at scale. A single PR may touch authentication logic, database queries, API contracts, front-end rendering, and infrastructure configurations. As AI-generated code increases the volume and complexity of changes, no single reviewer has sufficient depth of expertise across all dimensions.

The **Multi-Agent Reviews** pattern addresses this by decomposing reviews into specialist concerns, each handled by a dedicated agent with focused context, a specific evaluation

rubric, and a defined scope. A coordinator agent synthesizes the specialists' findings into a single, structured review that is triaged by severity and presented as a coherent assessment rather than a scattering of disconnected comments.

This pattern applies to situations where code changes routinely span multiple concern areas (security, performance, compliance, domain correctness, operational readiness, etc.) and where the consequences of missing an issue in any one area can be significant. It is most valuable in production systems at scale, regulated environments, and codebases with complex dependency structures. It applies less to small teams with a single codebase, where individual reviewers can hold the full system in their heads, to early-stage projects where architectural norms have not yet stabilized, and to environments where the overhead of coordinating multiple agents exceeds the risk of the changes being reviewed.

The core concept is similar in principle to ensemble methods in machine learning. Instead of relying on a single algorithm, ensembles combine the predictions of multiple models to create a single, highly accurate and robust model. In Multi-Agent Reviews, different models and configurations surface distinct categories of findings in parallel, producing richer results and deeper coverage than any single generalist reviewer.

The Multi-Agent Reviews pattern operates most effectively on code that has already passed baseline verification and is ready for deeper evaluation. This pattern assumes the Continuous Code Quality Loop has already run: mechanical issues have been resolved, the code compiles and passes tests, and the change is internally consistent. The architecture has four components:

- **Specialist agents:** Each is configured with a focused evaluation rubric, relevant context, and defined scope. Common specialists include a *security agent* (secrets handling, known vulnerabilities, injection vectors, authentication gaps, insecure defaults), a *performance agent* (caching behavior, latency thresholds, database query approaches, algorithmic complexity, resource utilization), a *compliance agent* (PII handling, SOX, PCI, HIPAA, GDPR, data retention, logging and audit trail), a *standards agent* (coding conventions, API contract compliance, architectural standards adherence), a *correctness agent* (race conditions, error handling, logic bugs, edge cases), and an *operational readiness agent* (observability, graceful degradation, failure modes, rollback safety, deployment configurations).
- **Triage agent:** Not every code change requires every specialist. The system should select which agents to invoke based on the change's risk profile, the organization's policy, and the files and subsystems it impacts. A documentation-only

change does not require a security specialist. The triage agent matches the depth of review to the actual risk surface of the change.

- **Coordinator agent:** The role of the coordinator agent is to receive findings from all the specialist agents and synthesize them into a coherent assessment. It deduplicates, assesses severity, resolves conflicts, and presents a single structured review with specific code locations and actionable remediation guidance. The coordinator agent also applies the organizational policy—which findings block the merge, which are advisory, and which should be suppressed.
- **Context routing:** This ensures each specialist agent receives the subset of context relevant to its concern. The security agent gets dependency information, authentication standards, and the organization's security policies. The performance agent gets database schemas and historical latency information. This helps reduce noise to each specialist agent.

The ensemble principle also extends to model selection, and organizations may benefit from running specialist agents on different model back-ends rather than routing all specialists through the same model, extending the Adversarial LLM Review pattern from a single independent reviewer to an ensemble of independent specialists.

Practical considerations include the tension between depth and coherence:

- Cost scales with the number of specialist agents. Running multiple agents in parallel on every PR is more expensive than a single pass. Risk-tiered invocation, by selecting which specialist agents to run based on the change's risk profile, keeps the costs proportional to the value of the review.
- Developer trust is earned through review quality, not quantity. Calibration is an ongoing process of tuning specialist agents' parameters and sensitivity based on the findings that led to real improvements and actions. Coordinator agent quality is also important; individual specialist agents can produce excellent results, but if the coordinator produces a noisy, contradictory, or poorly triaged summary, overall confidence can decline.
- Humans configure the specialist agents' roster and evaluation rubrics, review findings flagged by the coordinator agent, and make judgment calls on architectural trade-offs and business-context decisions that specialist agents are not equipped to evaluate. They also tune the system over time based on how findings correlate with bugs, production incidents, etc.

The Multi-Agent Reviews pattern decomposes reviews into separate concerns. Likely, the specialist agent roster, severity policies, and approval thresholds need to evolve as a system, and the broader Context Engineering pattern provides the infrastructure for encoding and maintaining the organizational knowledge that the specialist agents depend on.

## Pattern: Spec-Driven Development

Many of the hardest failures are not about a particular change set being wrong. They come from building the wrong thing or building it on the wrong assumptions, even if the code seems locally correct. By the time code is written and submitted for review, the most consequential decisions have already been made. Architecture, approach, scope, and assumptions are all baked in. Code review at this stage can catch implementation errors, but it is unlikely to reverse strategic mistakes, and the cost of trying is higher.

The shift in the **Spec-Driven Development** pattern is to move the primary review gate from the implementation to the plan: Review and harden intent before code exists, and treat code review as verification that the implementation faithfully realizes that plan.

The questions at this stage are different. Do you trust this plan? Is the implementation pipeline capable of executing it correctly? Is the plan unambiguous enough to execute at all? Success criteria must be defined as part of the initial plan, not discovered after the fact. This moves beyond code functionality to the more fundamental question of whether the change should be in the codebase at all, evaluated against adoption, reliability, performance, and business impact expectations. Edge cases, failure modes, and validation criteria are documented before any line of code is generated, making the resulting implementation a testable artifact of well-reasoned intent rather than something that must be reverse-engineered to derive meaning.

This does not eliminate code review. It changes its character from a gate that tries to catch what went wrong to a verification step that confirms a well-understood plan was faithfully executed.

In this pattern, the human is reviewing intent rather than implementation. Human judgment is applied where it has the most leverage: determining whether the right problem is being solved, whether the proposed approach is sound, and whether the system is being asked to do something it should not do at all.

In a fully agentic environment, humans may rarely review raw code; instead, they review intent and outcomes while agents handle the mechanics of writing and reviewing code. Getting there safely requires a rigorous notion of what we are approving when we approve a change, and Spec-Driven Development is the mechanism for that.

The Adversarial LLM Review pattern is a natural complement here. Once the coding agent has produced an implementation, an independent model can evaluate whether the result actually reflects the original intent, rather than merely whether the code is locally correct.

## CASE STUDIES

### Block's Approach to Scaling Agentic Code Review

Block began wide-scale adoption of AI-augmented development in early 2025, and by mid-2025, 90% of Block's 3,500 engineers were using AI tools daily in their workflows. As autonomous agents began producing more PRs, the organization encountered a new bottleneck: Human review capacity did not scale with agent output. Engineers were already managing multiple contexts and workflows. Reviewing another AI-generated migration, cleanup, or implementation PR often felt like low value and easy to defer.

Many agent-generated PRs were opened in draft mode under the assumption that a human author would refine them before requesting review. In practice, this often did not happen. PRs could sit untouched because the initiating engineer no longer felt the same ownership over the work they had delegated to an agent. This exposed a central reality of agentic development: Generating code is cheap, but organizational attention is not.

Block's first mitigation strategy was to introduce homegrown AI-powered code review. Early internal experiments allowed automated reviewers to comment on PRs once they left draft status. The results were mixed. The initial AI reviewers produced noisy or incorrect feedback, and engineers quickly lost trust in AI code reviews. However, abandoning AI review altogether was not a realistic option. As agent-generated PRs increased, human reviewers were already struggling to keep up with review volume. The challenge became finding a way to make AI review useful enough to absorb part of that load rather than adding even more work for engineers.

Later, Block piloted commercial code review workflows with a smaller set of teams before expanding them more broadly. These systems proved more useful. Engineers reported that the reviewers found issues humans missed and were increasingly capable of catching correctness and implementation problems earlier. Over time, AI review shifted from being another source of PR comments to serving as a first-pass verification layer.

Block also experimented with reducing review complexity by encouraging smaller, incremental changes. Rather than presenting reviewers with a single, large change set, engi-

neers increasingly used stacked PRs to break work into a sequence of focused, reviewable units. This made it easier for both human and AI reviewers to understand intent, isolate risk, and provide higher-quality feedback. This approach implemented the Small Incremental Changes pattern by reducing the size of each review decision rather than trying to make reviewers process more information at once.

One of the more effective changes was applying the Shift Left pattern—moving adversarial review earlier in the development life cycle. Instead of waiting for CI or human review, engineers could invoke a local adversarial agent before pushing code. This was Adversarial LLM Review in practice. The workflow resembled a review conversation more than a static linting pass, in which an authoring agent and a reviewing agent iterated together, often recording their discussion and reasoning in markdown artifacts attached to the work. The loop continued until the implementation reached a satisfactory state.

This changed the shape of the review. Rather than asking humans to review raw agent output, Block increasingly used agents to review other agents first, a Multi-Agent Reviews approach in which generation, critique, and remediation were distributed across collaborating agents before any human became involved. Human reviewers then focused on the remaining judgment calls: ambiguity, risk, system impact, product intent, and exceptions the automation could not confidently resolve.

The next evolution moved this pattern into CI, creating the Continuous Code Quality Loop. When automated review systems identified issues, agents were allowed to attempt fixes before a human reviewer became involved. An agent generated an implementation, an AI reviewer identified problems, another agent attempted remediation, and a human performed the final validation pass. The result was not fully autonomous delivery, but it significantly reduced the amount of mechanical review work humans had to perform.

A major lesson from the rollout was that the quality of reviews depended heavily on organizational context. Generic models could identify syntax problems or implementation errors, but they struggled with higher-level questions: whether the change aligned with product direction, whether it conflicted with customer requirements, whether a migration was necessary at all, and whether the behavior made sense for a particular service.

To address this, Block invested in Context Engineering by building internal knowledge systems that exposed service metadata, customer requests, and organizational decision history to agents during development and review. This allowed review agents to reason beyond code style and localized correctness toward intent, prioritization, and fit within the broader system.

The workflow also expanded beyond engineering teams. Work could begin from light-weight requests in systems like Slack, including requests initiated by non-engineers. A

request such as “solve this bug” could trigger an agentic workflow that moved from implementation through review and toward production readiness. This broadened who could initiate software changes, while also increasing the need for automated review systems to protect quality as the volume and sources of change grew.

Block’s experience directly illustrates this paper’s central argument. As AI accelerated code delivery faster than human attention could scale, the response was to design a verification system capable of producing sufficient confidence without demanding even more limited human attention. Each pattern the company adopted was a response to that constraint, and together they shifted review away from a single human inspection step toward a layered system where agents challenged each other, context-informed judgment, and humans were reserved for the decisions that genuinely required them.

## Cloudflare’s CI-Native AI Code Review System

Cloudflare’s internal AI code review system illustrates how agentic code review can be deployed as production engineering infrastructure instead of as a standalone assistant layered on top of PRs. The approach began from a familiar organizational problem: Code review is essential for catching defects and sharing knowledge, but it significantly slows engineering work when PRs wait hours for a human review. Cloudflare initially experimented with commercial AI review tools and with a simpler approach that passed a Git change set directly to a language model. Both approaches exposed limitations: insufficient customization for Cloudflare’s engineering environment, noisy recommendations, hallucinated errors, and generic advice that did not reliably help developers.

Cloudflare’s response was to build a CI-native orchestration system around OpenCode, an open-source coding agent. Rather than relying on a single general-purpose model to review an entire change, the system applies the Multi-Agent Reviews pattern: It launches multiple specialized reviewers for different concerns—security, performance, code quality, documentation, release readiness, and internal engineering standards. A coordinator agent then consolidates the reviewers’ outputs, removes duplicates, filters speculative or low-value findings, assigns severity, and posts a single structured review comment back to the PR.

This design reflects a shift from “AI as reviewer” to “AI as review workflow.” Embedding the system directly into CI so that verification happens automatically as part of normal software delivery, rather than as a downstream human step, applies the Shift Left pattern at the infrastructure level. The system does not merely summarize a change set or find possible bugs. It participates in the same CI process that governs normal engineering work, with

explicit policies for when a merge can be approved, when comments should be advisory, and when the system should request changes. Cloudflare’s approval policy is intentionally biased toward approval: isolated suggestions or non-blocking warnings can still result in “approval with comments,” while critical issues or production-safety risks will block a merge. A human “break glass” override remains available for cases where engineers need to bypass the AI reviewer, such as for urgent fixes.

A central feature of the system is “risk-tiered review.” Cloudflare does not use the same review depth for every PR. Small changes receive a lighter review, while larger changes or changes touching security-sensitive paths trigger the full set of reviewers. This reduces cost and latency for low-risk changes while preserving deeper scrutiny for changes more likely to introduce serious defects.

The system also applies Deterministic Engineering Practices by filtering noisy change sets before review: excluding lockfiles, minified assets, vendored dependencies, and generated artifacts, while preserving semantically important files such as database migrations. This reduced the amount of noise agents needed to reason about before any model judgment occurred.

The Cloudflare example also demonstrates that agentic review systems require operational controls beyond prompt design. Because the reviewer runs in CI, Cloudflare had to address failures common in production systems: model outages, rate limits, long-running sessions, context overflow, malformed outputs, and expensive, large change sets. The system, therefore, includes retry logic, model fallback, circuit breakers, timeouts, telemetry, and a separate control plane for changing model routing without editing the CI job itself. These features matter because once AI review is integrated into the software delivery pipeline, reliability and observability become part of the review system’s trustworthiness.

Cloudflare designed the reviewer to participate in iterative review through the Continuous Code Quality Loop, rather than relying solely on a single merge-time inspection. When developers push new commits to a previously reviewed PR, the system compares the new state against its earlier findings. Fixed findings are automatically resolved, unresolved findings are carried forward, and developer responses can affect whether a finding remains active. Rather than restarting from scratch after every commit, verification persists and evolves with the code.

Cloudflare’s review system operates at a significant organizational scale. Over a one-month period from March to April 2026, Cloudflare reported 131,246 completed review runs across 48,095 PRs in 5,169 repositories.

The system produced 159,103 findings during the period, averaging about 1.2 findings per review. Cloudflare interprets this low volume as intentional: The reviewer is designed

to prioritize signal over noise rather than maximize the number of comments, because developers' trust in automated review depends not only on whether the system catches issues, but also on whether it avoids wasting attention on speculative or trivial feedback.

Cloudflare does not present the reviewer as a replacement for human review. It highlights architectural judgment, cross-system impact, subtle concurrency bugs, and very large change sets as areas where AI reviewers remain weaker. These limitations are important. Cloudflare uses agentic code review as an augmentation layer—fast, consistent, and useful for many repeatable checks, but still dependent on human reviewers for architectural intent, system-level reasoning, and contextual trade-offs.

Cloudflare's system further demonstrates that agentic reviews only become trustworthy when the system around them is operationally sound. Multi-agent orchestration, continuous verification, deterministic filtering, and risk-tiered governance are the foundations upon which confidence at scale becomes achievable.

Every architectural choice Cloudflare made—from risk-tiered depth to deterministic filtering to the break-glass override—was an answer to this paper's central question: What level of confidence do we need, and what is the most efficient way to achieve it? And confidence at scale, as Cloudflare's production data suggests, is not only a model capability problem; it is an engineering problem.

## DryRun Security's Agentic Security Review

At DryRun Security, agentic code review began as a practical problem within the company's engineering workflow. DryRun started in 2023, using AI to write software from day one, and in classic startup fashion, dogfooded its own agentic contextual security analysis as it built the product. The team kept the standard GitHub PR workflow with enforced branch protection, so agent-written code couldn't be merged until it had been reviewed and approved.

When DryRun entered beta in late 2023, the team ran into the kind of problem that makes multi-tenant security very real, very quickly: Customer data could be accessed across tenant boundaries. The issue wasn't just a generic vulnerability; it exposed a product-specific security expectation that had to be enforced everywhere in the system. In a multi-tenant application, authorization and tenant isolation aren't optional implementation details; they're part of the product's security contract.

That incident changed how the company thought about AI code review. It wasn't enough for an agent to say whether a change set looked generally secure. The review system needed to understand the context of the application and the intent of the code—how authentica-

tion should work, where authorization checks must happen, how tenant context should be enforced, how customer data should be separated, and which SOC 2–related controls should apply. This illustrates the Context Engineering and Spec-Driven Development patterns in practice. The AI reviewer needs the code, but it also needs the context and intent: architecture, policies, ownership, environment details, and organizational expectations.

As the DryRun product matured, it naturally led the company to use the Multi-Agent Reviews pattern. After several iterations, DryRun’s review process now uses a general security analyzer alongside focused agents and sub-agents to examine specific concerns, including authentication behavior, authorization boundaries, tenant isolation, customer data handling, and internal policy conformance. The goal is not to produce more PR comments, but to return a structured assessment quickly enough to remain on the development path for agentic coding systems. In practice, this means useful feedback in thirty seconds.

DryRun also uses an exploitability test as part of the review process. Rather than only asking whether something appears suspicious, the system attempts to reason about whether the issue could actually be abused in context. This reflects the Adversarial LLM Review pattern: The authoring agent is trying to implement the requested change, while the review system approaches the result from the outside and asks which security assumptions might have been missed. This is especially important for bugs that don’t look dramatic in the change set but quietly weaken an authorization boundary or tenant separation rule.

By 2026, DryRun’s internal workflow will have become a generation, verification, and remediation loop. Dozens of coding agents contribute to implementation, while DryRun’s review system uses agents and sub-agents to review every PR against contextual security logic and custom policies. When an issue gets flagged, the finding must be useful and actionable for both the human reviewer and the authoring agent: The human needs to understand the evidence because they make the final judgment call, while the agent needs guidance specific enough to revise the code and rerun the review. This is the Continuous Code Quality Loop pattern applied to security review.

DryRun’s main lesson has been that the value of agentic review doesn’t come from asking a model to be a better version of a human reviewer. It comes from systematically encoding what the organization has already learned into an automated verification path. The tenant-isolation issue from beta became knowledge that couldn’t live only in memory, Slack threads, or tribal review habits. It had to become something the review system could consistently recognize as a code policy every time the code changed.

As models and coding agents change, the capability that needs to remain stable is model-agnostic security verification that understands the system’s own security contract,

explains risk in context, and sends the fix back into the development loop before unsafe code gets merged.

## CONCLUDING THOUGHTS: WHAT LEADERS SHOULD BE THINKING ABOUT

In some AI-assisted environments, engineers increasingly act as downstream validators of machine-generated code. This rapidly becomes unsustainable, and removing that human checkpoint only shifts the bottleneck. Upstream, decisions about what to build, how systems should behave, and how components interact carry more weight. Downstream, it's necessary to guard against increased risk and user impact. Thus, the transition to agentic code generation and review is best understood as a reconfiguration of how software organizations allocate responsibility, manage risk, and produce outcomes. Leaders who treat it as an organizational redesign will be better positioned to capture its benefits while maintaining control over risk, reliability, and outcomes.

Perhaps ironically, there are lessons to be learned from the heavyweight processes of the 1990s, although, fortunately, without the same heavy price paid for rigid, cumbersome practices.

- **Precise specifications are more critical than ever**, as ambiguity in intent is the dominant source of failure. However, because one person can span more of the development process, this does not have to turn into a slow relay race of handoffs. Rather, AI supports end-to-end ownership instead of fragmented, layer-based development models. Teams can reorganize around features or domains rather than front-end, back-end, and infrastructure silos without running headlong into the problem of having to know everything about everything. This reduces coordination overhead and aligns responsibility with outcomes.
- **Independent verification is necessary** to prevent circular validation, in which the generating agent checks that the code it generated does what it is supposed to do. However, unlike cumbersome, pre-Agile manual independent verification and validation practices, independent verification agents can provide continuous, rigorous feedback. Further, adversarial systems that detect drift and evaluate the performance of verification agents can provide an additional level of safeguards.

- **Audit records become even more important.** Fortunately, agentic systems can produce detailed records, including which policies were applied, how evaluations were conducted, and why decisions were made. These records can provide greater traceability and form a stronger audit foundation than traditional human review, provided they are treated as durable and queryable artifacts rather than transient logs.

These shifts in practice bring new questions and opportunities to evolve roles and responsibilities. The line between developers and testers has already been blurring. Now the line between architects, developers, and product managers blurs as the work shifts toward system design, prioritization, and orchestration across multiple initiatives.

The key question behind all of these practices is deceptively simple: If something were wrong, how would we know? Code reviews have historically been a key pillar in answering that question, but the traditional approach to code review simply doesn't scale. The current challenge for engineering leaders is organizing around and adopting a holistic set of practices designed to set clear intentions, assess changes incrementally, and provide visibility into the inner workings of software delivery.

## ABOUT THE AUTHORS

**Zach Davis** is a seasoned engineering leader with more than twenty years building products and scaling teams at high-growth companies. He is passionate about crafting high-quality product experiences, enabling engineering efficiency, and building tools that developers and product teams love. He was an early employee at both Atlassian and LaunchDarkly, and has recently shifted from engineering management to a more direct, hands-on Principal Engineer role leading developer efficiency and AI tooling.

**Michelle Gill** is an engineering executive and AI transformation leader with more than fourteen years of progressive leadership, including VP and Senior Director roles overseeing engineering organizations at hypergrowth SaaS companies and leading engineering organizations through the transition to an AI-native operating model. She brings a practitioner's perspective to the intersection of platform engineering and large-scale AI adoption, including enterprise AI, building GenAI teams, and autonomous AI agents. Her current work through Engineered by AI focuses on helping enterprise dev teams, founders, and CTOs build and scale AI-native engineering organizations.

**Elisabeth Hendrickson** is a technology leader with over thirty years of experience, having led geographically distributed teams of engineers, product managers, and designers as VP of R&D at a publicly traded company and VP of Engineering at a Series B startup. She currently works with technology leaders to improve collaboration, decision-making, and execution. She is the author of *Explore It!* from Pragmatic Bookshelf and coauthor of *Signals & Levers* from IT Revolution.

**Angie Jones** is Vice President of Developer Experience at Agentic AI Foundation where she works to advance the global adoption of open standards and best practices for agentic AI systems. A software engineer, executive, and inventor, she has spent her career helping organizations improve software quality, developer productivity, and engineering effectiveness through automation. Her current work focuses on AI-assisted development and the evolving relationship between human and machine collaboration in agentic software engineering.

**Sha Ma** is CEO and Founder of Topogy, Inc., an AI engineering acceleration platform that helps companies create leverage through shared context and intelligent workflows. A technology executive and engineering leader with more than two decades of experience, she has held CTO and VP of Engineering roles at category-defining technology companies from startups to post-IPO. Having led globally distributed engineering organizations through periods of rapid growth, platform evolution, and operational transformation, she currently focuses on improving AI effectiveness by connecting team collaboration, agent contributions, and evolving organizational knowledge to tangible business results.

**Randy Shoup** has spent more than thirty years building distributed systems and high-performing teams, and has worked as a senior technology leader at eBay, Google, and Stitch Fix. He coaches CTOs, advises companies, and generally makes a nuisance of himself wherever possible. As SVP of Engineering, he currently leads the Engineering organization at CircleCI.

**James Wickett** is CEO and cofounder of DryRun Security, where he and his team build an AI-powered security code review and verification product that is enforced pre-merge, applies policy, and detects vulnerabilities, including novel AI-made logic flaws and classic vulnerabilities. A software security leader, educator, and community builder, he has spent his career helping organizations improve application security and DevSecOps practices in fast-moving engineering environments. He has taught more than one million professionals through LinkedIn Learning and spoken at the RSA Conference, OWASP, and SXSW.