Sonar

The Agent Centric Development Cycle

Executive summary

Software development is becoming a hybrid human-machine activity, and the shift is accelerating. AI agents have moved beyond simple code completion to become full collaborators in the development process, writing substantial volumes of production code across business-critical systems. But this transformation has introduced a fundamental tension: the speed of code generation has far outpaced the ability of organizations to verify what is being produced.

The data makes the scale of this tension clear. According to Sonar's 2026 State of Code Developer Survey, 96% of software developers say they do not fully trust AI-generated code to be functionally correct. That skepticism is well founded. Researchers at Carnegie Mellon found that early velocity gains from AI tools typically fade within three months, while a 30% rise in static-analysis warnings and a 41% rise in code complexity persist long after. The result is code verification debt: code arrives faster than humans can meaningfully review it, and technical risk compounds quietly in the background.

Adopting agentic development without absorbing the accompanying risk requires a new operating model. This ebook introduces the Agent Centric Development Cycle (AC/DC), a methodology from Sonar built specifically for the realities of AI-driven code generation. AC/DC structures the relationship between engineers and AI agents through a three-pillar loop of Guide, Verify, and Solve, with zero-trust, multilayered code verification at its core, executed by an independent trust engine rather than by the same models that produced the code. Organizations that anchor code verification with SonarQube are already 44% less likely to report AI-linked production incidents, evidence that trust, when engineered deliberately, is what converts AI speed into durable business value.

The sections that follow examine why the trust gap exists, how verification debt accumulates when generation outpaces review, and what the Agent Centric Development Cycle looks like in practice. They close with concrete steps engineering leaders can take to begin implementing the methodology today.

Navigating code trust in the era of AI coding

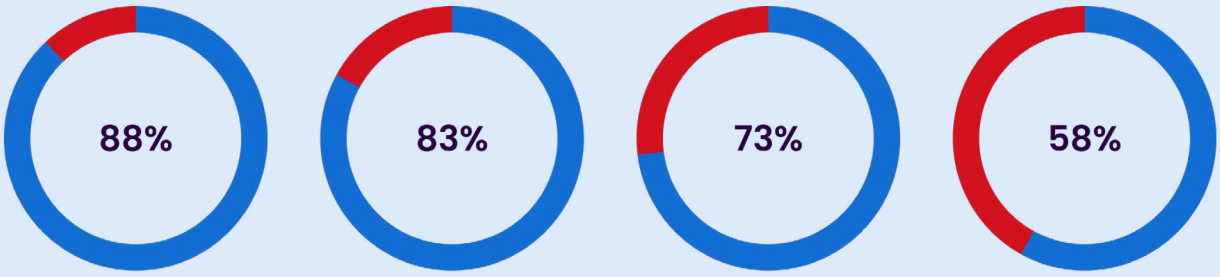
Low trust in AI-generated code

Software engineering is in the middle of a fundamental transformation that is changing how code is written, reviewed, and deployed. The era of typing code directly into an editor is over, giving way to a new paradigm in which developers manage and review the work of autonomous agents operating in parallel. Programming is increasingly about giving tasks in natural language to AI agents and supervising their execution. What were recently simple code-completion tools have become agentic coding platforms that act as full collaborators in the development process.

The data backs this up. According to the Pragmatic Engineer Survey of March 2026, a majority of software developers now regularly use AI agents in their daily work. And Sonar's State of Code survey found that this usage is not limited to side projects or experiments. AI-assisted coding has become a standard part of the software development lifecycle, with the vast majority of developers who have tried these tools reporting daily usage, and 58% of them reporting using AI to write code for business-critical or mission-critical services. It's no surprise that the volume of code being produced through these methods is substantial and climbing rapidly.

Developers are using AI across the gamut of software projects

Thinking about your team / company, which of the following types of work involves the use or assistance of AI?



n=1,149 | source: State of Code Developer Survey

In short, software development is quickly becoming a hybrid human-machine activity. But the resulting explosive growth in volume has not carried a matching level of reliability. Organizations are grappling with systems that can produce sophisticated results, but are just as capable of hallucinating APIs, generating bloated structures, and silently introducing technical debt and security vulnerabilities.

Realities of tool adoption versus functional trust

The central paradox of current development workflows is the gap between how widely AI tools are adopted and how much developers actually trust the output. The State of Code survey showed that while developers aggressively use AI for tasks like writing documentation and understanding existing code, they remain deeply skeptical about its effectiveness when it comes to modifying or maintaining complex legacy systems. The result is that the most common use cases are not necessarily the ones where the tools deliver the most reliable value.

The following data shows the percentage of developers using AI for specific tasks alongside the percentage who rate the tools as extremely or very effective for that same task.

	Use case	Effectiveness ↓	Adoption
A	Writing documentation	74%	74%
B	Explaining or understanding existing code	66%	78%
C	Vibe coding / creating new projects with mostly AI-generated code	62%	48%
D	Generating tests	59%	75%
E	Researching technical solutions or exploring APIs/libraries	59%	74%
F	Translating code from one language to another	58%	50%
G	Assisting development of new code	55%	90%
H	Code review	47%	55%
I	Debugging code	44%	65%
J	Refactoring or optimizing existing code	43%	72%
K	Adding or updating functionality in existing code	42%	76%

n=1,149 | source: State of Code Developer Survey

What this reveals is that software developers are highly pragmatic. They use AI because it eases the cognitive load of getting started (or because management demands speed) but they remain acutely aware that its output often lacks contextual awareness. This awareness shows up as a severe trust deficit in the survey: 96% of developers do not fully trust that AI-generated code is functionally correct. That lack of trust creates a persistent cognitive burden, as developers must now serve as continuous auditors for a flood of machine-generated contributions.

According to academic research, their skepticism is well founded. Multiple studies show that increasing a model's raw capability does not automatically produce proportional improvements in reliability. Researchers at Princeton tested 14 coding models across 18 months of releases and found that capability gains translated into only minor reliability improvements. Meanwhile, evaluations by METR showed that while modern models can handle complex, multi-hour software engineering tasks as long as you set the quality bar at a 50% accuracy level, performance collapses when the accuracy bar is raised to 80%. At that higher threshold, the complexity of tasks the models can handle drops significantly, with the time horizon of tasks that frontier models can complete dropping to roughly one hour. This demonstrates that accuracy remains a hard bottleneck for autonomous code generation.

To add further context, Sonar maintains a leaderboard that evaluates frontier large language models on real-world software tasks. The metrics include pass rates, verbosity (measured in lines of code), cyclomatic complexity, and defect rates per million lines of code.

RELEASED	MODEL	PASS RATE	VERBOSITY	COMPLEXITY	BUGS	VULNERABILITIES	RMI
Apr 2026	GPT 5.5 Medium	78.67%	703K	251	521	75	31.10
Apr 2026	Opus 4.7 Thinking	82.52%	336K	240	815	260	64.11
Mar 2026	GPT 5.4 Medium	81.09%	1778K	263	572	50	28.80
Feb 2026	Opus 4.6 Thinking	82.38%	563K	239	740	170	52.19
Nov 2025	Gemini 3 Pro	81.72%	289K	216	777	190	54.99

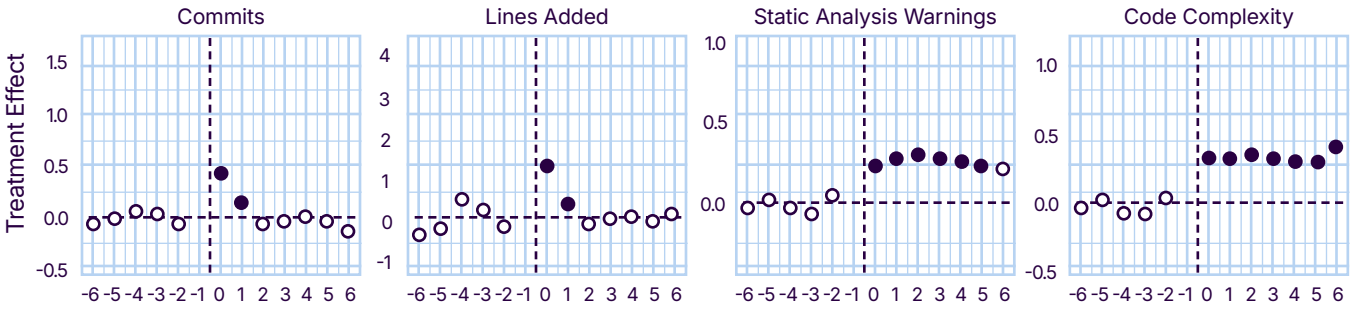
source: Sonar Leaderboard

In addition to things like code verbosity and rate of bugs created, Sonar also evaluates a metric called the Risk Mitigation Index, or RMI. This index merges the other metrics to create a mitigation burden score: an estimation factor of the cleanup a team takes on once the model has written the code.

The implication is clear: as organizations push developers to use AI coding models to increase output, they simultaneously expand the surface area for vulnerabilities, logic errors, and architectural drift. The models are highly capable, but they lack the deep institutional context of the specific repository they are modifying. This makes them prone to creating technically valid solutions that are architecturally wrong.

Verification debt and the risks of unchecked generation

One of the most striking findings from research on AI-assisted development is that productivity gains are often illusory, or at best short-lived, when it comes to long-term maintainability. A study by Carnegie Mellon University researchers analyzed projects that adopted Cursor and measured the impact on code quality using SonarQube. The data showed a dramatic three-to-five-times velocity spike during the first few months. But that spike disappeared entirely by the third month. Worse, adopting the tool correlated with a 30% increase in static-analysis warnings and a 41% increase in code complexity. These quality regressions ultimately created enough drag to slow overall development velocity after the initial excitement wore off.



source: Carnegie Mellon

This pattern points to what many now call “verification debt.” Because AI can generate code far faster than humans can comprehend it, a dangerous gap opens. Code arrives at a pull request instantly, but understanding what it actually does takes time and effort. If organizations do not actively scale their verification processes to match generation speed, unverified code inevitably creeps into production, causing outages and compounding technical debt.

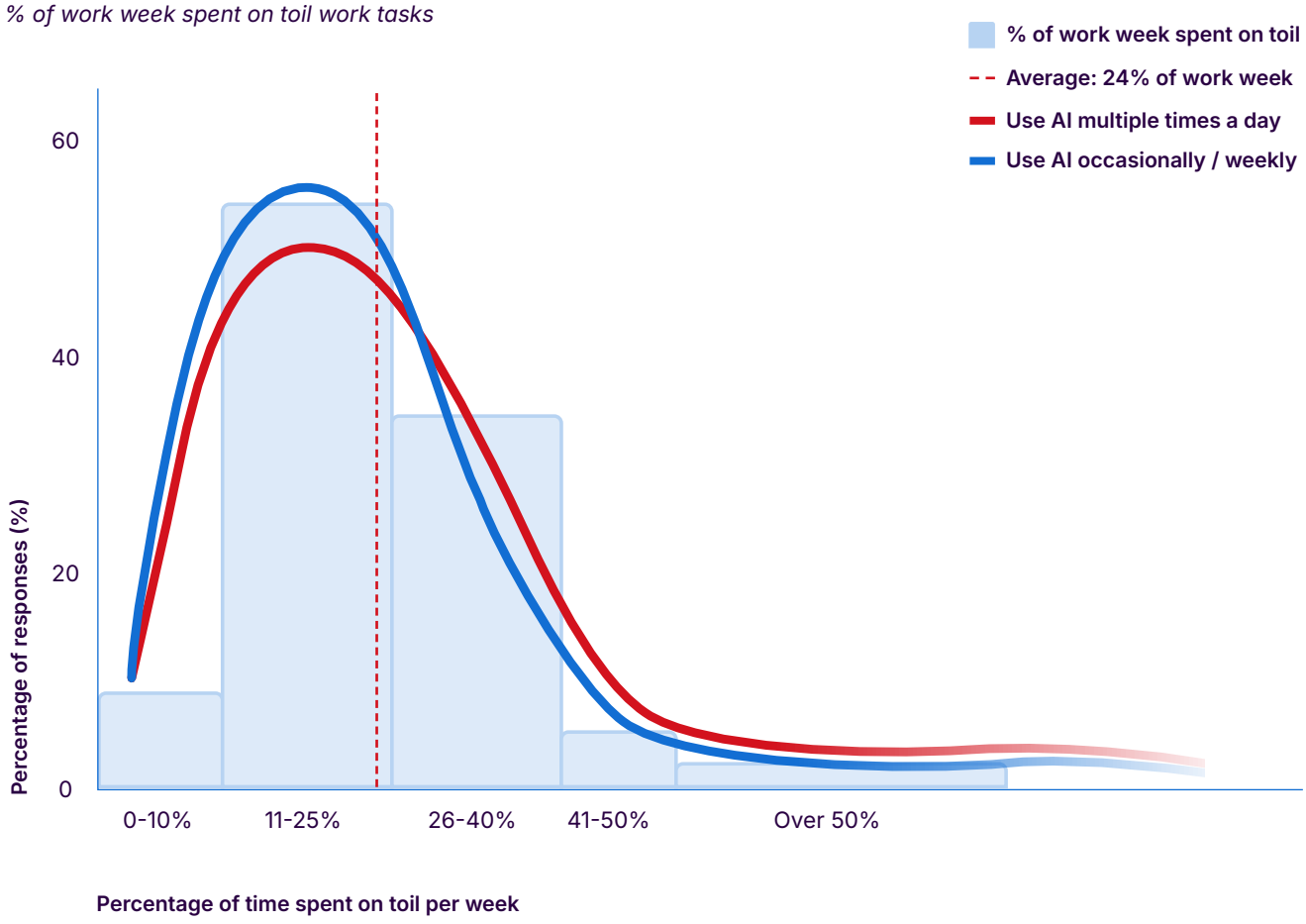
The problem is made worse by model hallucinations. Large language models are built to produce results rather than admit uncertainty, which leads to outputs that are highly confident but could be completely wrong. They frequently propose solutions that violate architectural constraints or invent APIs that do not exist. Verification debt therefore represents a growing gap in the ability of human reviewers to ensure the safety of the software being deployed.

How AI changes the nature of developer toil (but doesn't reduce it)

The State of Code survey shows that developers are feeling the pressure from verification debt directly. Although 75% say AI reduces the time they spend on toil, objective measurements of their work weeks reveal that the actual proportion of time spent on toil stays virtually unchanged—between 23% and 25%—regardless of how frequently they use AI tools.

Time spent on toil

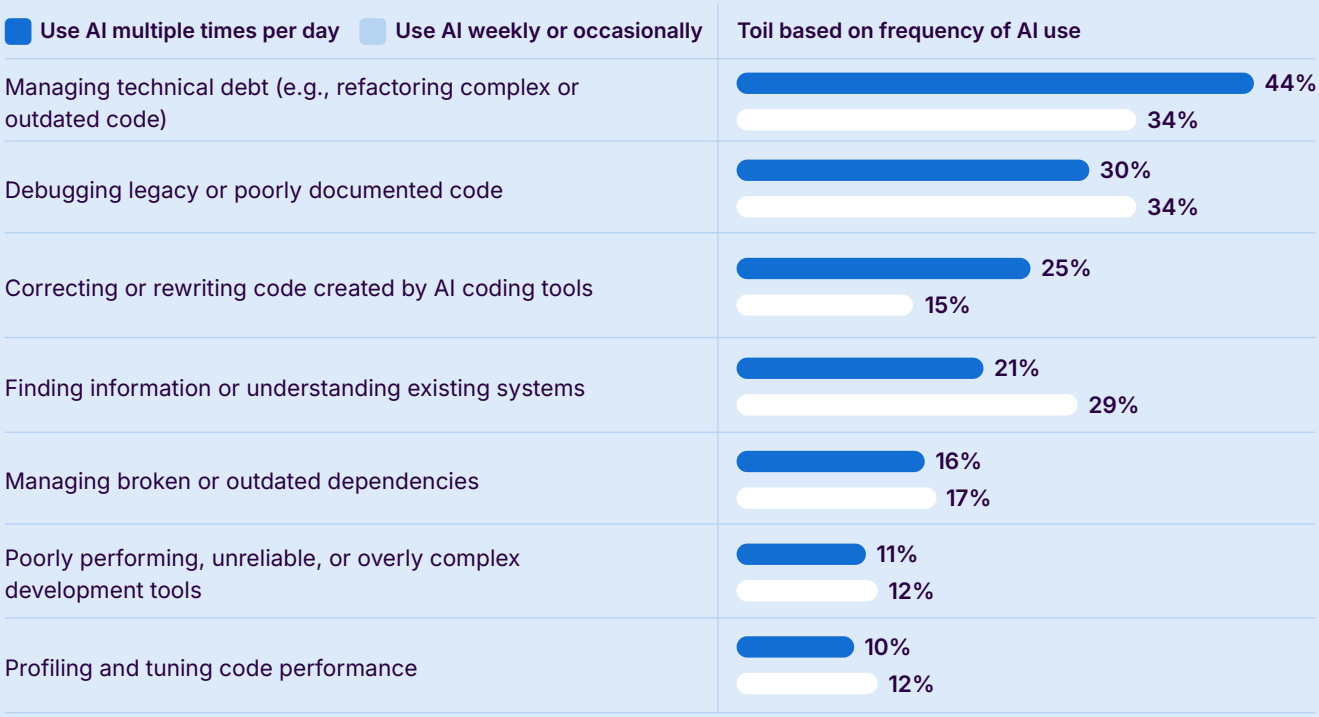
% of work week spent on toil work tasks



n=1,149 | source: State of Code Developer Survey

The nature of the toil, however, has shifted. As software developers lean on AI coding tools to eliminate the toil of searching for information or deciphering legacy code, they create a new category of toil focused on correcting machine-generated output and managing the resulting technical debt. The time saved upstream by having an agent write code gets consumed downstream during review and correction. The table below maps sources of friction for frequent AI users versus occasional users.

Thinking about your current workflows, what are your biggest sources of "toil work" that hinder / sap productivity or increase frustration in your current role?



n=626 | source: State of Code Developer Survey

This shift calls into question the return on investment of many enterprise AI initiatives. If a developer saves four hours generating boilerplate but spends four hours debugging subtle logic errors that an automated check could have caught, the net productivity gain is zero. The industry is catching on: engineering leaders are increasingly shifting focus from code generation speed to delivery quality and merge confidence. But in order to support these goals at the speed of code generation by agentic AI, a new approach is needed.

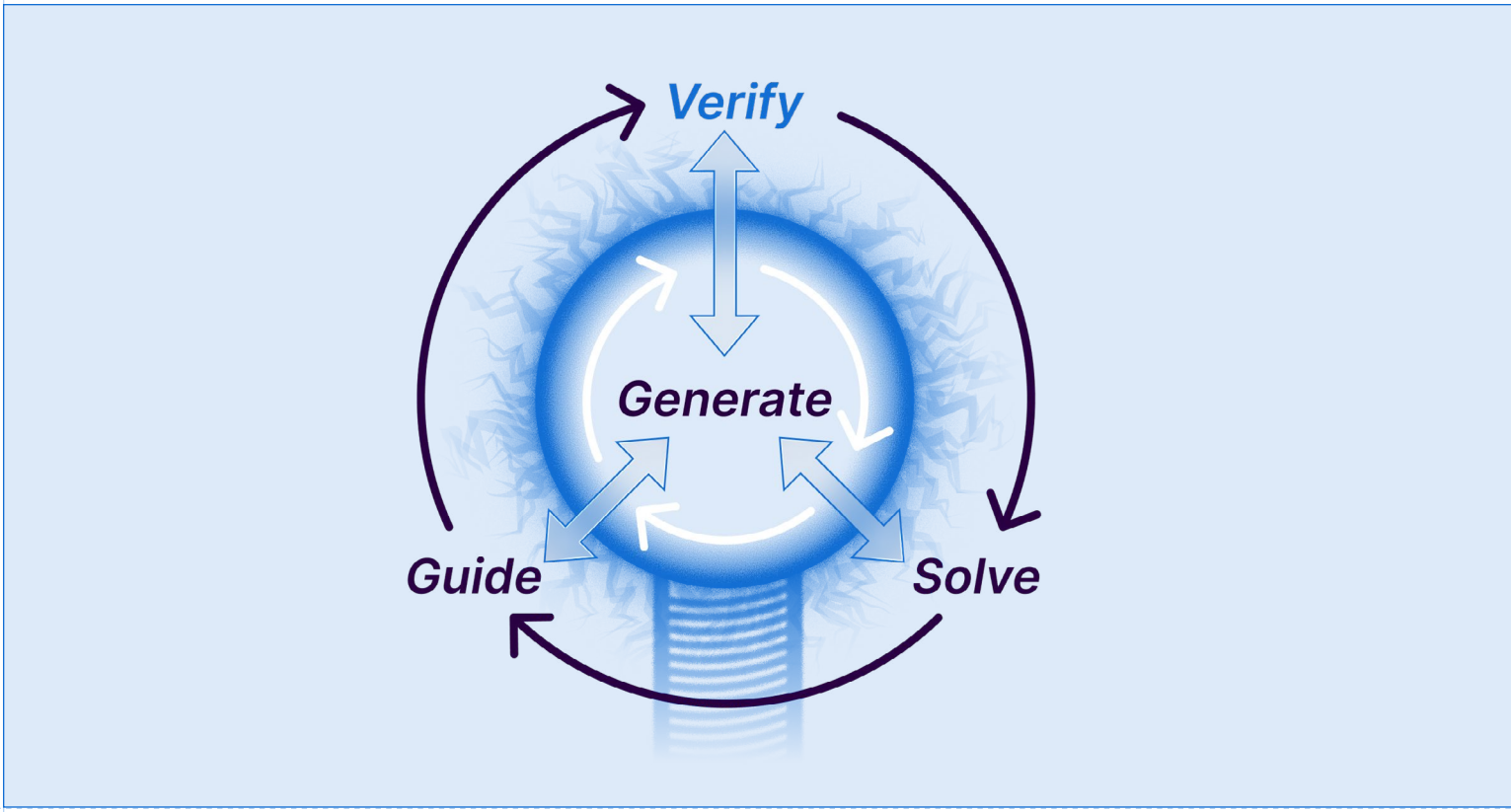
What is the the Agent Centric Development Cycle, and how does it work?

Addressing compounding risks and restoring real productivity requires a new methodology that treats AI agents as a core part of the workforce rather than a bolt-on tool. The industry consensus is that traditional workflows, designed for human typing speeds, are not equipped to handle agentic scale.

Sonar has introduced a methodology built specifically for this environment: the Agent Centric Development Cycle. It takes the best practices of conventional software development and reimagines them for the realities of autonomous code generation, in order to build trust in AI code. The goal is to ensure that AI agents operate in a trustworthy, consistent, and transparent way.

The cycle has three pillars that execute in a continuous loop, centered around the central act of code generation that's completed by AI agents.

What are the three stages of the Agent Centric Development Cycle?



Guide. In this first pillar, engineering teams define the context and constraints within which an AI agent must operate. That means providing guardrails, such as organizational standards, architectural intent, and data-flow expectations. Without this context, agents fill in the blanks with probabilistic guesses, producing code that is syntactically valid but contextually wrong: violating security postures, ignoring style guidelines, or misunderstanding data flows. Too little context leads to code that ignores required standards; too much context degrades model performance. The Guide pillar ensures that agents receive dynamically selected, authoritative context so they can produce appropriate outputs quickly and efficiently.

Verify. This is the most critical step, and the starting point for most organizations. AI is non-deterministic and can produce results that look correct but fail under specific conditions. Therefore, verification must be mandatory, comprehensive, and automated so that it's fast enough to keep up with agents. This pillar must operate on a zero-trust, multilayered approach. It requires a mix of logical reasoning, deterministic evaluation, and consistent analysis that provides auditability and complete repeatability. Organizations cannot rely on AI models to review other AI models exclusively; doing so leads to inconsistent results that endanger security posture and fail compliance audits. Effective verification separates duties, applying precise code intelligence to validate reliability, maintainability and security before code moves toward production.

Solve. Once issues surface in the Verify pillar, the cycle closes the loop through automated remediation. Precise, actionable reports from code verification are handed to specialized code repair agents. Armed with a deep understanding of application structure and the specific context provided during the Guide pillar, these agents generate targeted fixes for any issues found. Every issue identified and solved feeds back into the Guide pillar for the next iteration, making the entire system progressively smarter and more resilient.

This cycle plays out across two layers: an inner loop and an outer loop. The inner loop runs within each individual agentic reasoning cycle, ensuring that an agent validates its own logic as it works through a task. The outer loop runs after the agent completes the broader task, providing a comprehensive, centralized review before the pull request is merged. This dual structure catches errors early and prevents them from compounding into large structural defects.

Multilayered zero-trust verification with SonarQube

To make the Agent Centric Development Cycle work, engineering teams need a neutral, rigorous platform that holds all code to the same standard regardless of whether a developer or an agent wrote it. SonarQube serves as the independent trust and code verification engine for this cycle. It delivers consistent, repeatable findings with a low false-positive rate, giving organizations the assurance they need to ship code confidently. Its deterministic-first methodology ensures every finding is explainable and fully auditable.

The properties that make large language models effective at generating code are precisely the properties that disqualify them from verifying it. AI coding agents are probabilistic, meaning the same prompt can yield different outputs on successive runs. This makes their assessments inherently non-reproducible. They are also constrained by context windows that cap how much of a codebase they can process in a single pass, presenting a hard limitation when enterprise monorepos routinely contain tens of millions of lines of code. And their outputs are opaque: modern compliance regimes require a documented, reproducible audit trail where every finding traces to a specific rule applied against a specific code path, a standard that black-box AI outputs cannot meet.

SonarQube's zero-trust architecture avoids these issues by allowing a strict separation of duties between AI for code generation and deterministic review for verification. This ensures that the review process is consistent, repeatable, auditable, and fully explainable (often a requirement for meeting modern regulatory and compliance standards), irrespective of the chosen AI solutions used to write it. SonarQube also maintains a benchmarked false-positive rate of just 3.2%, eliminating the friction of unreliable results as code volume grows, and letting engineering teams ship AI-assisted code with the same confidence they have in developer-written contributions.

SonarQube implements multilayered code verification by orchestrating a diverse set of analysis engines, each targeting a distinct category of risk. The process begins with deep, deterministic static analysis that maps data flows, semantic syntax, and architectural dependencies across more than 40 languages to identify structural weaknesses and security hotspots. These high-precision checks are then layered with additional capabilities such as AI-driven code reviews for logical consistency. By combining these techniques, SonarQube ensures there is no single point of failure in the verification process.

Sonar provides a complete suite of solutions to support all three pillars of the cycle:

Guide pillar capabilities. SonarQube's architecture and context augmentation capabilities dynamically captures organizational guardrails and architectural standards and feeds them into any agentic framework.

Verify pillar capabilities. SonarQube serves as the core execution engine for multilayered verification. SonarQube Advanced Security adds dependency-aware application security testing and software composition analysis to protect from broader software supply chain security risks. For teams working with highly agile agents, SonarQube Agentic Analysis provides optimized self-verification directly inside the agentic inner loop.

Solve pillar capabilities. SonarQube AI CodeFix provides targeted remediation suggestions for issues surfaced by the analysis engine. The SonarQube Remediation Agent uses LLMs to solve issues at scale, and can be used to tackle backlog issues as well as problems found during automated PR review.

SonarQube bridges the gap between speed and quality by keeping analysis embedded in the workflows that developers and agents actually use. Solutions like the SonarQube MCP Server and SonarQube CLI let software developers and coding agents test code quality, security, and maintainability directly within their AI-native environments.

Sonar customers already see the benefits. The State of Code survey found that organizations using SonarQube are 44% less likely to report that AI adoption led to a higher frequency of production outages and incidents.

Conclusion

The transition to agentic software engineering is an irreversible shift that is fundamentally changing how organizations create value. But the first wave of adoption has exposed a critical reality: generating code faster does not automatically mean shipping better software. Organizations that optimize purely for volume find themselves trapped by a verification bottleneck, spending enormous time auditing machine-generated output and clearing complex technical debt.

To capture the promise of AI without exposing operations to serious risk, engineering leaders must put trust mechanisms in place that match the speed of generation. A multilayered, zero-trust approach to code verification is essential for enterprise-safe development.

If you're looking for ways to get started, here's what we recommend:

1. **Establish bounded autonomy.** Give AI agents the freedom to generate code, but enforce centralized approval gates and deterministic analysis before any machine-generated code reaches production.
2. **Implement the Agent Centric Development Cycle.** Stop treating AI tools as simple chat companions. Move to a structured framework where agents are systematically guided by context, verified by independent metrics, and used to solve their own mistakes.
3. **Equip developers with the tools to orchestrate.** Recognize that reviewing AI code and designing context frameworks are the most valuable focus areas for developers in the AI era. Shift training and tools toward systems thinking and prompt design rather than raw code syntax.
4. **Standardize on a single verification platform.** Eliminate the security blind spots created by disconnected tools. Apply the same rigorous quality gates to developer-written and machine-generated code across the organization.

Ultimately, AI makes an excellent collaborator. But the organizations that will successfully navigate this revolution are the ones that empower their human software engineers to direct strategic intent, while relying on robust, independent platforms to verify every line of code produced.

Use the AC/DC framework today

The Agent Centric Development Cycle is designed to be model and toolchain agnostic, serving as a universal trust layer across a fragmented AI landscape. So transitioning to AC/DC does not require an immediate overhaul of your entire infrastructure. You can begin today by establishing your verification platform as a mandatory gateway for all machine-generated payloads, ensuring that every AI contribution is measured against defined quality profiles and organizational guardrails.

If you have SonarQube Cloud, you already have access to everything you need to get started with the AC/DC framework. By integrating SonarQube into your current environment through integrated plugins, our MCP server, SonarQube CLI, and more, you empower your team to automate these checks at the speed of agentic reasoning rather than being limited by human manual review.

To help you map out this transition, we have developed a set of practical implementation guides that provide a clear path for integrating Guide, Verify, and Solve stages into your workflows. You can find these at www.sonarsource.com/acdc.

Interested in trying SonarQube Cloud?

Ask for a free trial!