Editor: **Ciera Jaspan**
Google
ciera@google.com

Editor: **Collin Green**
Google
colling@google.com

Developer Productivity for Humans: Navigating the Tensions of AI in the Software Development Lifecycle

Jessica Baolin, Collin Green, Ciera Jaspan, and Francisco Gutiérrez, Google

Understanding Developer Experience With AI

AS GENERATIVE AI (GenAI) has expanded, our industry has focused increasingly on understanding how developers experience GenAI tools in their work. At a high level, the sentiment numbers look promising; internal survey data at Google show a consistent positive trend in AI sentiment. We've seen a 20% increase in AI sentiment over the past two years. But high-level sentiment scores alone don't capture how developers experience their day-to-day work, nor do they help us identify opportunities for further improvement. While we know that improvements in model quality and performance are helpful, we also wanted to understand how developers experience AI-powered development through the lenses of speed, ease, and quality and what significant, systemic tradeoffs they face. We wanted to examine how developers articulate these tradeoffs in their own words.

Fortunately, we collect data on developer experience quarterly, through our longitudinal survey program.[1] Our team thematically analyzed 1,110 open-ended responses from a random sample of Google developers who were asked: "Please share any thoughts or comments you have around how AI-powered tools have impacted your development workflow in the last three months." What we found is that AI introduces a series of systemic tradeoffs where productivity gains are in tension with points of friction.

Because our survey question was intentionally broad, the feedback we received was quite varied. Our analysis of these 1,110 narrative experiences proceeded in two sequential phases.

First, we mapped the data by specific use cases across the software development lifecycle (SDLC) to pinpoint where AI was most visible and impactful to developers. The vast majority of touchpoints clustered around four primary activities: code generation, information seeking, code review, and testing. Beyond these core pillars, developers also reported a notable footprint in secondary activities, including debugging, refactoring, writing documentation, and prototyping.

Second, we conducted a deeper thematic analysis within each use case to capture explicit developer sentiment. It was during this phase that we observed that a set of consistent themes cut across entirely different development activities.

Digital Object Identifier 10.1109/MS.2026.3703116
Date of current version: 8 September 2026

We found these themes interacted as four sociotechnical tensions associated with using AI. Understanding these overarching tensions is essential for any engineering organization trying to realize true operational return on investment with GenAI.

Four AI Tensions

When analyzing the qualitative feedback across these applications, we identified clear universal themes mapping to positive and negative sentiments. These tradeoffs reveal that individual efficiency gains often collide with systemic friction.

Work Saved Versus Work Shifted

Developers frequently observed that work saved from using GenAI in one part of their workflow resulted in work being shifted to other parts of their workflow or even to other people.

Developers reported that work saved during initial code drafting was reallocated to additional overhead from prompting and verification. This tradeoff was also seen in the July 2025 METR study, where they found developers spent less time coding but more time interacting with AI through prompting and reviewing AI output.[2] This happens because writing a prompt that leads to successful results is nontrivial; it requires careful and explicit articulation of constraints. Verification is also nontrivial; it requires high-scrutiny auditing of nondeterministic (and possibly hallucinated) output. Additionally, these tasks also require a different frame of mind than writing code, so developers expend effort shifting among "careful prompter," "programmer," and "code reviewer."

This work shifting dynamic extends beyond the individual to the entire team. Developers report work being shifted from one person to another in code review: the velocity gains for an author can translate into an increased time or cognitive load for their reviewer if the author did not do enough verification themselves.

Overall, GenAI provides time savings in some areas of development, with sometimes increases in others. Applying GenAI in development requires a thoughtful balance and awareness of these tradeoffs to achieve net savings.

> What we found is that AI introduces a series of systemic tradeoffs where productivity gains are in tension with points of friction.

Realizing Value Versus Incurring Debt

The drive for rapid output often stands in opposition to the long-term health of the code base and documentation. When developers leverage AI as a high-speed drafting assistant, they can inadvertently introduce semantic bugs, technical debt, and hollow documentation.

For instance, developers note that AI-generated code can be verbose or nonidiomatic. They perceive AI models to default to the most frequent patterns found in their training data, which may not produce the most optimal, modular abstractions for a specific architecture.

This tradeoff extends to documentation use cases, too. Developers note that AI tends to output generic documentation: polished prose that looks sophisticated but lacks the deep, precise "why" behind an architectural decision. Over time, the accumulation of these incomplete artifacts can make it harder for both human developers and future models to find high-quality reference examples. Industry research notes this pattern and suggests that it signals a shift from traditional technical debt toward "cognitive debt" and "intent debt," where the code itself works, but the underlying engineering rationale is not well understood, and team mental models quietly erode.[3] Technical, cognitive, and intent debt make it progressively harder for both human developers and future models to find high-quality reference examples. They can also contribute to the longer-term erosion of expertise (more on this topic below).

GenAI is a force multiplier, and it can be used both to increase or decrease debt. Engineers must utilize it appropriately to manage this tradeoff and ensure sustainable operations when needed.

The Quick Start Versus the Last Mile

Our developers report that AI tools are very effective at lowering the activation energy for development work. They reduce friction to start tasks, moving the developer more easily from a daunting blank page into an active state of editing. Yet developers find that they hit a wall as they get farther into making their code production-ready.

Developers comment that prototype or boilerplate infrastructure can be built almost instantly, but the engineering work needed to ready a system for production (e.g., handling edge cases, security, regulatory compliance, and integration with complex internal infrastructure) can require more manual troubleshooting than if the project had been started from scratch. This tension is particularly noticeable in the testing use case, where developers reported instances of AI tools lying (a specific type of hallucination) about tests passing when it hadn't even executed the tests.

This tension is related to the prior two: verifying the completeness and correctness of AI-generated code and assessing the extent to which it satisfies present requirements and incurs new technical debt are integral parts of production readiness. While it's always been necessary for engineers to be aware of where they are in the product development lifecycle and manage this tradeoff, it is even more important with GenAI-based tooling.

Capability Versus Expertise

AI tools excel at functioning as timely tutors, allowing developers to bridge knowledge gaps, enabling them to write code in unfamiliar languages or navigate legacy code bases without prior domain context. Developers reported that AI empowers them to work in unfamiliar territory but also that they may feel they lack the expertise to verify that AI-generated solutions are correct.

By bypassing the "productive struggle" traditionally required to master a system, developers risk falling into superficial learning habits and long-term skill degradation. AI may create a long-term risk to deep technical expertise in some cases.[4] Industry research has shown that as AI alters standard workflows, organizations must actively update on-the-job training to safeguard developers against the risk of deskilling.[5] This tension is further exacerbated by the default affirmation bias found in idea generation use cases, where models tend to validate and praise a user's ideas rather than identify flaws or challenge assumptions.

Without deep technical expertise and access to well-vetted design rationale, developers may struggle to evaluate accuracy. The result is sometimes an environment where they can generate code quickly but are unequipped to debug the complex, systemic failures that arise downstream. Using GenAI effectively means making conscious decisions of when to build long-term expertise in a topic versus when shallow knowledge is good enough.

Actionable Strategies for Practitioners

Navigating these four tensions requires shifting software development strategy away from a narrow focus on tool adoption and toward structural, day-to-day adaptations. Whether you are leading an organization or committing code yourself, the following strategies can help.

Optimize the End-to-End Workflow, Not Just Code Creation

The paradigm of focusing solely on individual developer speed and increasing AI adoption can create a massive auditing and management tax across the SDLC. To achieve net time savings, teams must shift from optimizing isolated tasks to reconsidering the broader workflow:

- *Automate continuous validation at the point of creation*: Prevent downstream bottlenecks by shifting the verification burden earlier and to automated systems. For example, instead of relying directly on the developer to do verification, there should be automated feedback loops between the coding agent and other feedback mechanisms, including testing and context-aware validation agents. This would reduce the validation burden on developers and their teammates.
- *Enforce micro-scope discipline*: Because AI makes it trivial to generate massive swathes of code and content instantly, strict adherence to small, modular goals is critical. Restricting tasks to tiny, isolated units shortens local debugging loops when a tool falters and keeps verification overhead from overwhelming the individual or their peers.
- *Consolidate the toolchain*: Minimize tool sprawl to preserve flow state. Constantly jumping among disconnected AI assistants adds decision-making toil; focus on cohesive, unified developer journeys. Directly combat decision-making toil by minimizing tool fragmentation and the cognitive load that comes with agent proliferation. This can help eliminate the hidden tax developers pay when constantly choosing among, configuring, and jumping among disconnected AI assistants.

Establish Antidebt Countermeasures

Because AI can rapidly proliferate verbose, nonidiomatic code or low-value documentation, teams must explicitly reward code health over raw output volume:

- *Protect code quality*: Conduct regular "refactoring sprints" or use specialized context-aware tools to actively consolidate, inline, and abstract repetitive, AI-generated boilerplate patterns. AI tools focused on readability or adherence to documented coding standards may lighten the load for teams on this front. Thoughtfully scoping the work that agents do and do not take on autonomously (e.g., specifying what kinds of bugs agents may fix without human intervention) is also helpful.
- *Protect documentation quality*: Establish strict technical density standards for AI-written documentation. Enforce peer review checks specifically designed to strip out generic prose, ensuring every document contains the critical, high-context "why" behind architectural choices before it is merged into the ecosystem knowledge base.
- *Establish and evolve technical debt management processes*: Fundamental principles underlying effective technical debt management still apply to AI-powered software development.[6] Making intentional decisions about technical debt tradeoffs, tracking and measuring technical debt, and sharing responsibility for technical debt across an engineering organization can help mitigate the reckless, inadvertent accumulation[a] of technical debt.

Close the "Last Mile" Production Readiness Gap

AI excels at lowering the activation energy to start a task, but it does not simplify the rigorous requirements of production-grade development:

- *Calibrate estimates:* Project timelines must explicitly account for the discrepancy between rapid prototyping and the effort required to achieve production-grade quality. Planning for several iterations of coding, testing, and experimentation will occur ahead of production-level hardening. Resource and timeline estimates should explicitly account for the production readiness work.
- *Ground AI in proprietary data*: Generic models struggle with internal nuances. Link development tools directly to internal application programming interfaces, proprietary code bases, and architectural documentation to drastically reduce hallucinations.

Safeguard Learning and Expertise

Lowering entry barriers to new domains is powerful, but automated shortcuts risk bypassing the "productive struggle" required to build and maintain deep technical mastery:

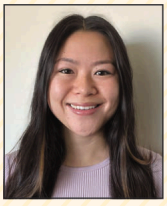
- *Design for active learning*: When using AI to learn, prompt it to act as a conversational coach that explains structural logic rather than an agent that simply outputs a copy–paste snippet.
- *Focus on structural mentorship*: Protect time for junior-to-senior pair programming specifically to deconstruct AI-generated architectural paths, preventing blind tool affirmation.
- *Preserve manual coding intervals*: For highly novel, critical, or complex system components, deliberately choose to code manually. Forcing the brain to construct the mental model from scratch prevents skill degradation and guarantees the team can debug complex failures. Consider deliberate manual coding for specific contexts, too: early career developers, new hires, or developers new to a code base can benefit from deeper engagement.
- *Build sociotechnical immunity*: Recent evidence suggests that organizations can protect against skill and expertise erosion by sensing it early (e.g., setting alerts around low- or no-time or zero-comment code reviews and watching for failure to test edge cases), actively containing it (e.g., requiring some AI-off code changes or reviews and requiring meaningful human engagement at key decision points), and implementing structural mitigations (e.g., using some saved time for skill strengthening activities and evolving trainings).[4]

Beyond AI Adoption: Optimize Team Dynamics

When we look at the bigger picture, AI tool deployment and adoption does not happen in a vacuum. The tensions we uncovered in developer comments and the strategies we suggest to strike a productive balance are important. However, we also know from internal research that merely increasing AI usage intensity does not, by itself, improve productivity or the overall developer experience. Rather, the benefits of AI appear to depend heavily on the quality of underlying team dynamics.[7] To leverage AI usage for maximum return, engineering organizations must actively invest in core team dynamics across four distinct dimensions:

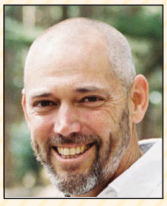
[a]https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

ABOUT THE AUTHORS



JESSICA BAOLIN (LIN) is a user experience researcher for the Developer Intelligence Team at Google, Seattle, WA 98109 USA. Contact her at jsclin@google.com.

CIERA JASPAN is a senior staff software engineer on the Developer Intelligence Team at Google, San Francisco, CA 94105 USA. Contact her at https://research.google/people/cierajaspan/ or ciera@google.com.



COLLIN GREEN is a senior research manager lead for the Developer Intelligence Team at Google, San Jose, CA 95134 USA. Contact him at https://research.google/people/107023/ or colling@google.com.

FRANCISCO GUTIÉRREZ is a user experience researcher for the Developer Intelligence Team at Google, 11000 Ciudad de México, Mexico. Contact him at franciscogh@google.com.

- *Process clarity*: Prioritize establishing guidelines regarding which tools should be deployed for specific tasks to eliminate cognitive tool fragmentation.
- *Strategic alignment*: Keep team practices aligned with long-term business and user values to counteract the drift caused by generic, AI-generated code and documentation.
- *Team culture*: Build high psychological safety so that developers feel comfortable raising concerns about verification overhead, auditing fatigue, or model hallucinations without fear of being penalized for speed.
- *Balanced workload*: Explicitly track and adjust team capacity to absorb the newly shifted cognitive burdens of prompting and code auditing, protecting developers from burnout.

Ultimately, GenAI does not rewrite the laws of software development rigor; it balances them on a tighter scale. Managing these systemic tensions is essential to ensuring that gains in individual speed do not come at the expense of engineering health and ecosystem quality.

References

1. S. D'Angelo et al., "Measuring developer experience with a longitudinal survey," *IEEE Softw.*, vol. 41, no. 4, pp. 19–24, Jul./Aug. 2024, doi: 10.1109/MS.2024.3386027.
2. J. Becker, N. Rush, E. Barnes, and D. Rein, "Measuring the impact of early-2025 AI on experienced open-source developer productivity," 2025, *arXiv:2507.09089*.
3. M.-A. Storey, "From technical debt to cognitive and intent debt: Rethinking software health in the age of AI," 2026, *arXiv:2603.22106*.
4. U. Ehsan et al., "From future of work to future of workers: Addressing asymptomatic AI harms to foster dignified human-AI interaction," in *Proc. CHI Conf. Human Factors Comput. Syst.*, Barcelona, Spain, 2026, pp. 1–21, doi: 10.1145/3772318.3791081.
5. M. Kam et al., "What do professional software developers need to know to succeed in an age of artificial intelligence?" in *Proc. 33rd ACM Int. Conf. Found. Softw. Eng.*, 2025, pp. 947–958, doi: 10.1145/3696630.3727251.
6. P. Avgeriou et al., "Technical debt in the AI era," *IEEE Softw.*, vol. 43, no. 4, 2026, doi: 10.1109/MS.2026.3683173.
7. C. Taylor et al., "Software development is a team sport," *IEEE Softw.*, vol. 42, no. 3, pp. 13–17, May/Jun. 2025, doi: 10.1109/MS.2025.3539403.