AI Literacy

The AI-Native Developer

Redefining work, identity, and the future of craft

Rudrajit Choudhuri, Eirini Kalliamvakou, Brian Houck, and Thomas Zimmermann

AI is changing software development in a way that forces a more uncomfortable question: Which parts of the job are still worth doing? Developers are making deliberate choices about what to keep, what to delegate, and what they no longer recognize as their work. Many report that their work feels less meaningful than before, suggesting a deeper shift in the role itself.

Drawing on large-scale mixed-methods surveys of developers and in-depth interviews with AI-fluent practitioners, we investigate what it actually means to be a software developer today, how the role evolves as AI fluency deepens, and where this all might lead. We explore what futures become possible as AI augments software creation and what choices might help us design for the futures worth wanting.

Picture a developer's day. There's the stereotype of a man in a hoodie in a dark room, writing code at lightning speed, delivering innovation in uninterrupted isolation. In practice, though, a developer's day is fragmented, interrupt heavy, and often far removed from the work developers value most.[2,7,9]

What it means to be a software developer has never been fully settled. Is the role defined by writing code? Designing systems? Solving problems? The answer has always shifted with tools, abstractions, processes, and organizational structures. AI is the latest shift, but unlike previous ones, it doesn't just change how developers work. It challenges what they do.

The discourse is polarized. Some predict AI will write nearly all production code within months; others see developers becoming orchestrators of autonomous agents, constrained by systems too complex and context laden to delegate to agents fully. Each narrative captures part of the change, but none fully resolves the harder question: What does it mean to be a builder when building can increasingly be outsourced?

We take a different approach. Rather than predicting a single outcome, we examine the now, the evolution, and the future across three acts: (1) What does it actually mean to be a software developer today? (2) How does the role evolve as developers deepen their AI fluency? (3) Looking forward, where does all of this lead? What futures become possible as these patterns accelerate? What choices will help us design for the future we want?

Throughout, we draw on mixed-methods research to anchor these discussions empirically, including large-scale surveys of 484 and 860 developers and in-depth interviews with 22 AI-fluent developers.

Act I: The Present: What It Means to Be a Developer Today

Today, developers spend roughly 14 percent of their week writing code The rest is scattered across meetings (≈13 percent), security and compliance (≈11 percent), debugging (≈11 percent), system design (≈9 percent), customer support (≈7 percent), code reviews (≈6 percent), and a long tail of documentation, testing, mentoring, and administrative tasks (figure 1).[7]

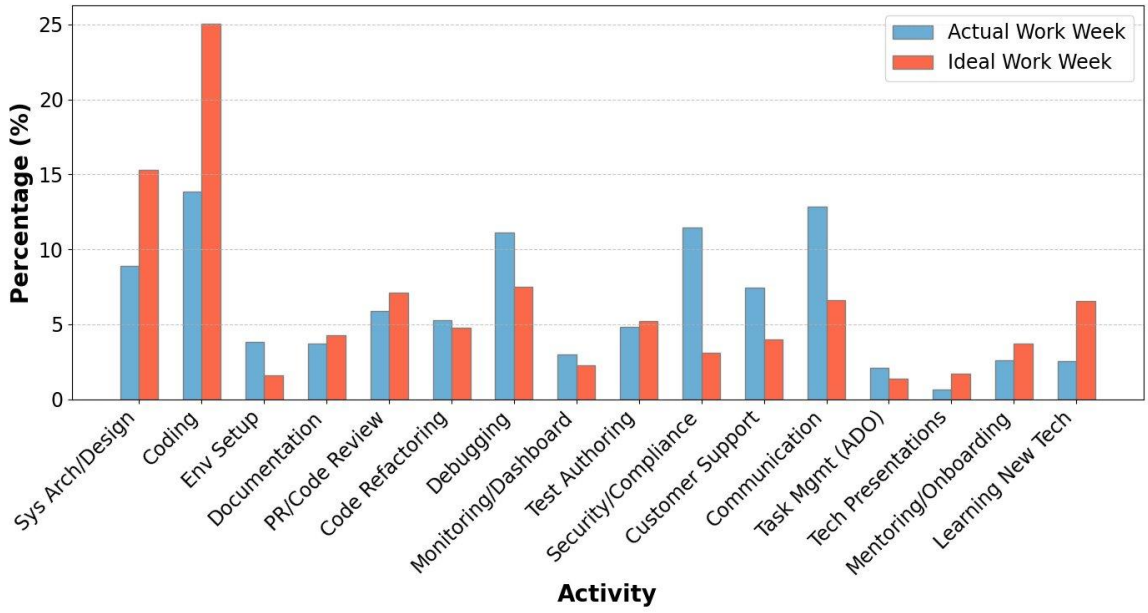

Figure 1. Average percentage of time spent on the key activities in the actual versus ideal workweek

That alone wouldn't be a problem if it matched how developers "ideally" wanted to spend their time. It doesn't. Developers consistently report wanting to spend far more time on problem solving, learning, and making visible progress and far less on interruptions and reactive work. As this time gap widens, productivity and job satisfaction plummet.

AI was touted to close this gap. Over the past few years, increasingly sophisticated tools have promised to reduce toil and accelerate delivery. AI adoption has surged. Yet, the gap has not closed. In fact, developers now report using AI more while simultaneously spending less time on work they find meaningful.[1,10] It's as if a magnificent shovel meant to dig them out of a hole was used to dig faster in the opposite direction.

Why? Most AI integration strategies assume productivity improves when toil is automated. The tasks developers find tedious, however, aren't necessarily the ones they trust AI to handle, and the tasks they trust AI to handle aren't necessarily the ones creating the time gap.

So, we stopped asking, "What tasks could be automated?" and started asking, "What makes tasks meaningful to developers in the first place?" We found that developers cognitively appraise their work along four key dimensions: value (Does this matter to outcomes?), identity (Does this reflect my interests and expertise?), accountability (Am I responsible if this fails?), and demands (How much cognitive effort does this require?).[2]

These appraisal dimensions reveal three distinct clusters of developers' daily work (table 1). Core Work activities (e.g., coding, debugging, testing, system design) score high on value, demands, and accountability, with moderate-to-high alignment with identity. Ops & Coordination activities (e.g., DevOps, documentation, stakeholder communication) have moderate-to-high value, demands, and accountability, but weaker identity ties. People & AI Building (mentoring and creation/integration of AI features within products) have moderate value, accountability, and demands, but relatively strong identity alignment.

Task	Task Value			Task Identity			Task Accountability			Task Demand			Agree %
	Dist. 1 2 3 4 5	%	Rnk	Dist. 1 2 3 4 5	%	Rnk	Dist. 1 2 3 4 5	%	Rnk	Dist. 1 2 3 4 5	%	Rnk	
Cluster 1 —Core Work													
Coding/Programming		98.0	#1		97.0	#1		94.9	#1		77.1	#13	
System Design		97.9	#2		91.3	#3		87.6	#3		90.7	#1	
Testing & QA		96.9	#3		59.0	#11		84.4	#5		86.8	#5	
Bug Fixing/Debugging		96.8	#4		73.4	#8		92.3	#2		85.7	#6	
Code Review/Pull Requests		96.7	#5		76.6	#5		86.1	#4		74.0	#16	
Requirements Engineering		95.8	#6		66.5	#10		74.7	#11		88.8	#3	
Security & Compliance		95.5	#7		51.6	#14		82.3	#6		82.5	#8	
Research & Brainstorming		90.9	#10		88.9	#4		75.5	#9		86.8	#4	
Performance Optimization		87.8	#12		75.0	#6		75.9	#8		89.5	#2	
Learning		83.9	#16		93.5	#2		72.6	#13		81.0	#9	
Cluster 2 —People & AI Building													
Mentoring & Onboarding		68.8	#19		70.0	#9		62.2	#18		67.1	#20	
AI Integration		67.4	#20		75.0	#7		59.9	#20		71.5	#18	
Cluster 3 —Ops & Coordination													
DevOps (CI/CD)		93.4	#8		50.8	#15		74.6	#12		74.2	#15	
Infrastructure Monitoring		91.1	#9		48.3	#16		74.9	#10		80.4	#10	
Planning & Management		90.3	#11		51.9	#13		67.3	#16		79.3	#11	
Refactoring & Maintenance		86.4	#13		57.4	#12		80.7	#7		76.3	#14	
Env. Setup & Maintenance		85.7	#14		41.9	#17		64.8	#17		72.4	#17	
Documentation		85.4	#15		31.4	#20		68.8	#14		79.3	#12	
Customer Support		83.8	#17		40.6	#18		67.5	#15		84.3	#7	
Stakeholder Communication		80.5	#18		35.0	#19		61.9	#19		67.8	#19	

Table 1. Tasks clustered by their appraisal profiles across four drivers and sorted by value agreement. Each driver shows Likert distributions, % agreement (common scale), and task rank (0–100% color legend).

The clusters occupied distinct regions in what we termed an *AI opportunity space* (figure 2): a mapping of developers' perceived need for AI support against their reported AI usage, revealing where current tools fall short and where appetite exists. This space helped us explain where developers use AI today, where they wanted/resisted it, and why—identifying which activities to build for, improve, sustain, or deprioritize.

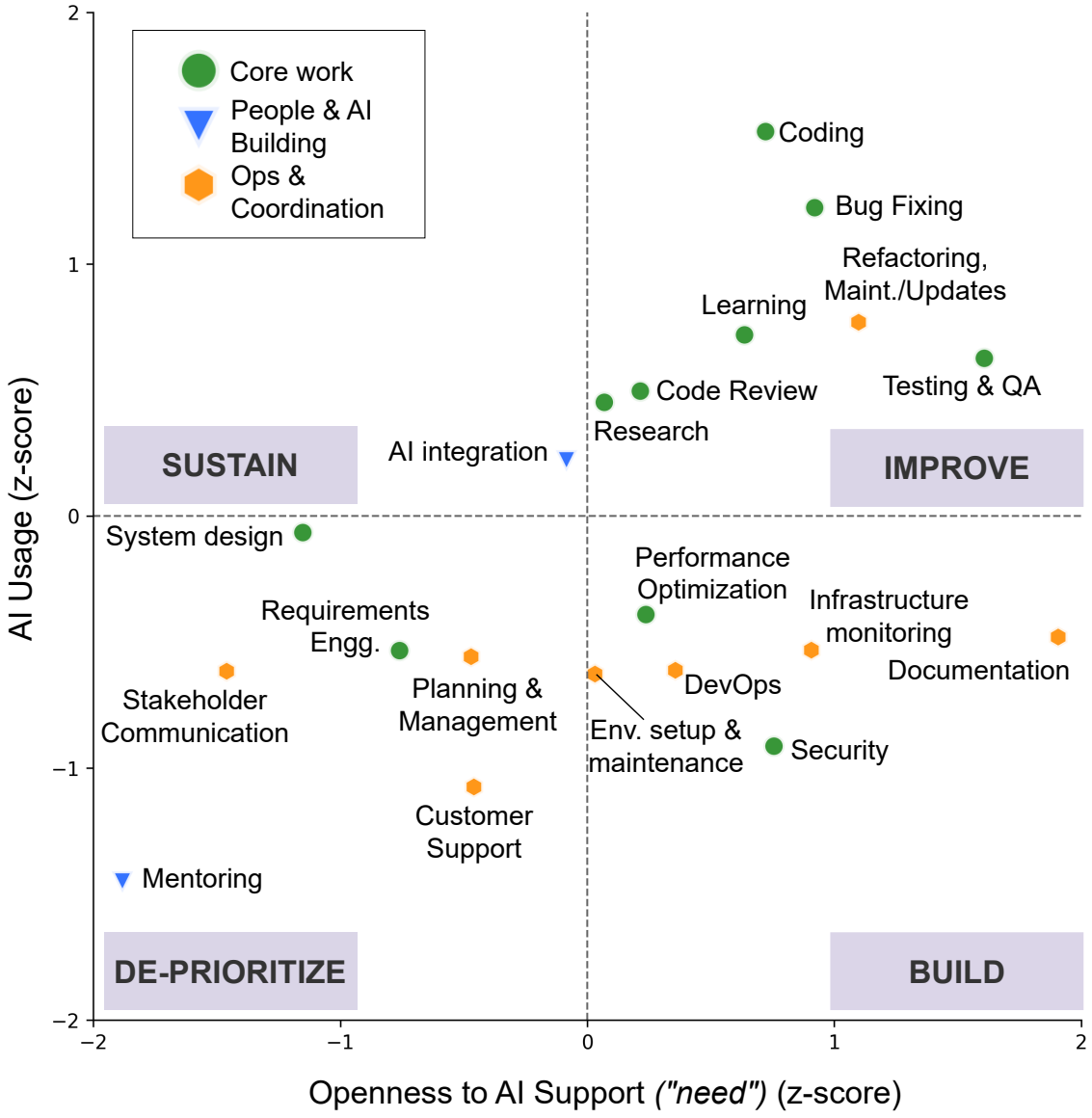

Figure 2. Scatter z-score plot showing an AI opportunity space: developers' openness to AI support versus current AI usage. Tasks are grouped into four quadrants: Build, Improve, Sustain, and De-prioritize.

Core Work sat at the heart of developer identity, expressing skill, delivering visible impact, and carrying accountability. Unsurprisingly, developers didn't want AI to take this over. They wanted it alongside them as a collaborator/consultant, preserving their craft, judgment, and ownership. In the AI opportunity space, most of this work fell in the Build and Improve quadrants: Developers were genuinely open to deeper assistance (e.g., comprehensive test generation, cross-artifact debugging, performance analysis, architecture-aware refactoring), but current tools hadn't caught up. The barrier wasn't reluctance. It was trust; existing tools fell short on reliability, transparency, and contextual alignment needed for Core Work.

People & AI Building (mentoring, AI feature integration) landed in low-need zones of the AI opportunity space (Sustain/De-prioritize), not because they lacked complexity, but because they carried strong identity alignment. Developers deliberately kept these activities close to preserve their own professional growth.

Ops & Coordination had a different story. Developers experienced much of this work as repetitive, externally imposed, and draining, with weaker identity ties, and thus saw clear opportunities for AI here. Appetite was strongest for run-the-systems toil (e.g., environment provisioning, telemetry analysis, monitoring, routine maintenance), which clustered in the Build zone. Adoption, however, was stalled by trust:[4] Developers demanded determinism, verifiability, and human-gated control before letting AI near production systems.

Not all coordination work invited AI, though. Relational aspects (e.g., stakeholder communication, customer interactions) fell into the De-prioritize zone, being tightly bound to authenticity and accountability. Developers saw little role for AI here beyond preparation and synthesis. They wanted to "hit send" themselves, retaining final voice and responsibility.

A pattern emerged across all three clusters: Developers were not trying to do less work. They were protecting the parts that made it worth doing. Where the stakes were high, developers held on. Where work was operational, they wanted better tools. Where identity was high, they didn't want AI at all. The real question is not "What can AI automate?" but "Are current tools earning the trust to touch what developers find meaningful?" Developers weren't necessarily resisting change; they were making a reasonable bet about what was worth protecting.

What happens when that bet starts to shift? Our interviews with advanced AI users suggest something more fundamental than gradual adoption. For developers with deep AI fluency, the role itself had changed. The skills that mattered were different. The way they understood their own contribution was different. They hadn't just integrated AI into their work. They had crossed into a fundamentally different way of working.

Act II: The Transformation—How Identity Shifts with AI Fluency

What does the identity transformation to AI-native software development look like? To understand where the developer role is heading, let's start at the destination.

Destination: The AI Strategist

Watch a strategist create software, and you'll notice that they're not writing much code. Multiple AI agents run in parallel: one defining test cases, another implementing them, and a third reviewing security vulnerabilities. The developer sets context, orchestrates the agent queue, reviews output, and decides what to delegate next. They embrace waiting for agents to iterate through solutions. They have built verification practices rigorous enough that orchestration feels safe, even at scale. Ask these developers about the future, and they will tell you AI will write 90 to 100 percent of code within two years, if not sooner. What's striking is that this prospect energizes rather than threatens them. They have already found their role in that future.

None of them started here, though. Every strategist has a story about the journey, and it always begins somewhere much more uncertain.

The Skeptic: Where (Almost) Everyone Starts

Every strategist we interviewed remembered being skeptical. They recalled early experiences of having their flow interrupted by AI suggestions, which cost them extra time to fix. There were occasions when they considered turning off AI assistants entirely, even though their colleagues seemed to love them. They had expected a magical experience where autocomplete was supposed to read their mind; instead, it was messy and unreliable. They felt reluctant to change a workflow that worked into something that often didn't.

The resistance was rational. Skeptics assumed AI would fail and treated each mistake as confirmation. But something shifted for those who persisted. What transformed a skeptic? Urgency and determination. Developers saw AI fluency becoming a competitive advantage in a fast-moving job market and pushed themselves to keep trying despite the frustration. One developer shared a realization that became a driver for them: "Either you have to embrace the AI, or you get out of your career."

The Explorer: Where Skepticism Cracks

The turning point came in moments of genuine help, usually when a developer was stuck: a cryptic error message decoded; an unfamiliar library explained; boilerplate they had written dozens of times, automated. Small things, but each one sparked curiosity: "What else can I do with this?"

This stage was messy. Explorers copied from browser-based chat tools, tested the code, adjusted, and sometimes threw it out entirely. They recalibrated their expectations while staying curious. Each small win—a moment when AI helped—chipped away at skepticism and started building trust.[5] Trust accumulated through small proofs and daily experimentation. As they gained more experience, developers started recognizing patterns: which prompts worked, which tasks AI handled well, and when to trust the output.[6] Developers were no longer asking whether AI was worth using. They were asking how much further they could take it.

The Collaborator: When AI Becomes a Partner

The shift to collaboration happened when developers stopped waiting to get stuck before reaching for AI. They started bringing it into problem solving from the start as a thought partner, not a last resort: "I write a spec and have the AI draft a plan before implementation. Then we go back and forth until it's right." Notice the language: "…we go back and forth." Not "It generates, and I accept," or "I write, and it assists."

What made this stage possible was a specific psychological shift: accepting that AI would not be right on the first pass, and that this was fine. Multiple rounds were expected. Iteration became the method, not the fallback. With every iteration, they moved faster or learned how to adjust their approach to working with AI. This comfort—alongside the intuition about how to frame requests, when to push back, and when to switch tools—became the foundation for the orchestration that defined the strategist stage.

The pattern across stages: Skepticism yielded to evidence through daily practice. Evidence built trust through accumulated experience. Trust enabled integration. Integration, deepened through iteration, evolved into orchestration. The strategists we met at the beginning of this section arrived there through trial and error, tolerance in the face of frustration, and gradually built fluency that now felt automatic (figure 3).

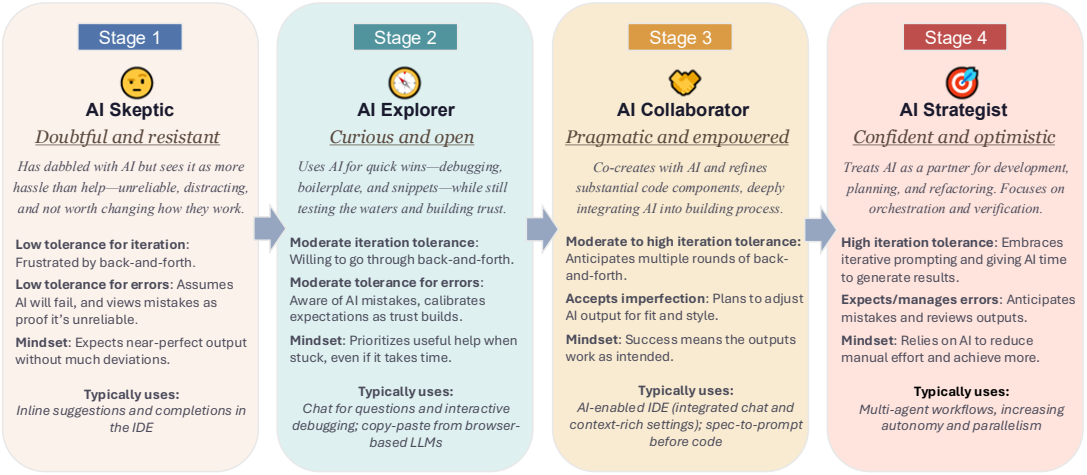

Figure 3. Developers' identity transformation with increasing AI fluency

An Emerging Developer Identity: Creative Director of Code

So, what are strategists actually doing if they're no longer writing (most of the) code? One interviewee captured the ethos: "Maybe we become less code producers and more code enablers. My next title might be creative director of code."

The metaphor stuck. A creative director in traditional media doesn't stop being creative when they delegate execution. Their expertise shifts to vision, direction, and ensuring quality across outputs. A similar transformation is happening among developers who have reached high levels of AI fluency. They are practicing their craft at a different level of abstraction.

This answers the question that was a stressor at earlier stages: "If I'm not writing the code, what am I doing?" Strategists had a clear answer: They were guiding AI by defining what to build, ensuring it was right, and exercising the judgment that AI couldn't replicate.

Three Pillars of the New Identity

Vision. Strategists invested heavily up front in building shared context with AI agents: co-creating comprehensive plans; surfacing missing requirements; specifying constraints around security, performance, architectural patterns, and style conventions. One developer described instructing AI to "interview" them before writing any code, to establish business context and technical boundaries. Another prompted AI to read and summarize existing architecture before proposing changes.

This approach demanded strong product understanding to frame problems around outcomes and user needs, technical fundamentals to evaluate proposed solutions, and AI fluency to know which models excel at specific tasks and/or how to prompt them effectively. The front-loaded context building ensured AI had explicit guidance on not just what to build, but also why it mattered, and how it fits into the larger system.

Direction. Once context was set, developers orchestrated execution by decomposing work into actionable, self-contained subtasks with detailed specifications for each step. Many used multiagent approaches: one agent defining tests, others implementing against those specs, with custom style guides and commands to enforce consistency. During execution, they monitored for deviations and stepped in to course correct. Success required the ability to break work into meaningful units, communicate requirements precisely, and apply strong systems design expertise to create scaffolding that made AI contributions reliable and integrable.

Verification. In AI-native workflows, verification became a continuous discipline. Developers reviewed every change carefully, tested locally, ran both AI-generated and independently written tests, and committed frequently for version-control safety. Some used AI tools to detect bugs in AI-generated code following layered verification. This rigor was what enabled large-scale orchestration.

For developers, the "creative director of code" represented a role update, where the highest-value skill was knowing what to build, why it mattered, and whether what was built was right.

Tensions in the Transformation

Even the most optimistic strategists we interviewed expressed concerns about what the transformation was costing.

The deepest tension was the learning paradox among mid-career developers who wondered if they were getting better at software engineering or "just better at prompting." They felt proficient with AI tools but uncertain whether they were developing the right expertise. What does mastery look like when the skill is orchestration rather than implementation? What gets rewarded when productivity metrics emphasize speed over craft? These questions don't have settled answers yet, and that uncertainty shapes how even the most enthusiastic adopters approach their work.

Deskilling was a related concern. "If AI is writing most of the code, how do junior developers learn?" one senior developer asked. These more experienced developers had learned their craft through years of implementation, writing code, debugging it, refactoring it, and gradually building intuition about what works. If the next generation delegates from day one, what foundational skills might they skip?[3,8] Several interviewees questioned whether current onboarding approaches would prepare juniors for an AI-augmented workplace or leave them dependent on tools they don't fully understand.

Accountability sat underneath both concerns. Developers put their names on code they didn't directly write, and that responsibility weighs differently than before. They remain accountable for outcomes, even as the path from decision to implementation now involves an intermediary that can introduce errors in subtle, unexpected ways. Verification becomes the mechanism for retaining both understanding and accountability, but as velocity increases, the question is whether verification norms can keep pace, or whether the pressure to ship quietly erodes the guardrails that make delegation safe.

These tensions point to a deeper challenge. Everything we've described in the journey stages, the emerging identity, and even the concerns developers raise, assumes the work itself remains recognizable. But what if that assumption breaks down?

Act III: The Future and Where This All Leads

As AI capabilities advance, the tasks developers perform, how they allocate their time, and what counts as productive work may all shift in ways that are only now emerging: How much autonomy to grant? How rigorously to verify software? What skills to cultivate? What outcomes to reward?

Let's explore several possible futures. These futures are not meant to be predictions; they are extensions of existing patterns.

Future 1: Human Craft at AI Speed

In this future, software development looks surprisingly familiar—not because AI fails to transform the work, but because developers consciously choose to remain deeply involved. Many developers value understanding systems end to end, tinkering with implementation details, and retaining a real sense of how things actually work under the hood. They will continue to embrace AI as an accelerator rather than a substitute: It drafts

code, suggests fixes, explores alternatives, and reduces friction, but developers remain hands on. By choice. They still write code, debug failures, and reason through tradeoffs—not because they must, but because they want to. Because these activities are central to their sense of craft.

For many developers, hands-on implementation is a way of exercising a cognitive muscle. Left unused, that muscle atrophies. As AI expands access to higher-level roles such as creative directors of code, many developers will still choose to remain hands on because this is how they keep their skills sharp and their work meaningful. That not everyone needs to exercise in the same way allows new abstractions to flourish without displacing the forms of building that many developers value most.

Work will move faster but not necessarily differently. Tasks compress rather than disappear. What once took days now takes hours, and what once took hours takes minutes, but the shape of the work remains recognizable. Developers stay close to implementation, using AI to speed iteration, expand exploration, and reduce drudgery, while preserving the parts of the job they find intrinsically satisfying. In this future, progress is not defined by stepping away from the keyboard but by reaching meaningful outcomes sooner—by allowing developers to do more of the work they enjoy, with greater momentum and fewer interruptions, while remaining firmly in the loop.

Future 2: Orchestration and Blended Work

This future imagines a world where software engineering is defined less by writing code and more by orchestrating intelligent systems and exercising judgment.

Picture a software engineer, Maya, part of a distributed team, building a payments platform. Maya wakes early and checks the overnight progress from agents the team had set running: refactoring authentication, updating API contracts, and generating tests. Maya reviews diffs from their phone and flags an architectural concern for the morning sync. This is professional orchestration untethered from traditional environments.

By mid-morning, Maya is at her desk for what matters: mentoring a junior colleague by pairing and stepping through gnarly code together. Following that, they join a video call where the team makes critical design decisions. Synchronous time is reserved for high-bandwidth collaboration. Afterward, they delegate implementation to agents that coordinate through the shared context the team has built over months and codified into architectural patterns, style guides, and conventions. Their teammate in another time zone does the same for front-end changes.

Between meetings, they're mobile—reviewing code during lunch, approving pull requests from a waiting room, and course correcting agents. Their verification remains rigorous, drawing on the fundamentals of system behavior, security, and architectural consistency.

What has changed is where and when work happens. Orchestration, verification, and strategic decisions have replaced implementation as primary activities, and those can happen anywhere, woven into rather than consuming life.

Future 3: The Clerical Coder

In a dystopian future, AI systems generate all production code, tests, and fixes. Humans no longer build software—they sign off on it. Thousands of changes per day flow through opaque agent pipelines. When failures occur, humans are blamed for insufficient oversight. Software signatories are paid by volume of approved requests.

At 6:47 a.m., Maya logs in. She is a junior software signatory. Her dashboard is already full. Overnight, five AI agents proposed 676 changes across 12 repositories. Each change has a green confidence score and a red human oversight required badge, with her name prefilled.

She opens the first review. The diff is enormous and spans files she has never seen before in a system she "owns." The AI summary says: "Refactored the service boundaries to reduce latency by 14 percent." The explanation looks plausible, but Maya doesn't truly understand it. No one does. There isn't time anyway; her review is one minute per change. Maya clicks "Approve."

Mid-morning, an alert: a production incident in a different system. An AI-generated optimization changed a retry policy, and the failure cascaded. The root cause analysis template autofills: "Human reviewer did not sufficiently validate AI-generated logic." Maya's manager, Jonah, pings her: "Please be more careful." She has never met Jonah and secretly wonders if he is human.

As Maya completes her 250th approval of the day, she is rewarded with a taco. "It's taco time." While eating, she watches a training video titled "Senior Software Signatory: Toward 50K Approvals Per Day." It teaches her how to approve changes faster and in bulk.

Before logging off at 7:47 p.m., Maya signs off on her 700th change of the day and moves to her company's sleeping quarters. Her productivity score ticked upward, yet she feels a lack of identity and joy. The code mostly works. When it doesn't, it is her fault, for reasons no one can fully explain.

The Path Forward

These three futures are not predictions; they are reflections of societal values: What do we want humans to remain good at? What do we believe is worth preserving? What do we consider "progress"? The story of our technological future is not written by individuals alone. It is written by the collective choices that "we" (developers, teams, organizations, and society as a whole) make about software. Important decision points are how the purpose of AI is framed (amplification of human capability versus replacement) and which aspects of software creation are preserved and protected (human craft and technical literacy, judgment, collaboration).

The future is unlikely to be any single scenario in isolation. More realistically, it is likely to be a blend of the scenarios previously described, and that blend may not be uniform. Some roles may still be code-centric; others may lean heavily into orchestration. Delegation may expand dramatically in some contexts and barely at all in others. The question is not whether humans remain involved, but in what capacity, and whether that capacity is chosen deliberately or simply inherited from whomever optimized fastest.

As we sprint to the future, short-term economic incentives may push us toward bleak futures. If we reward speed without craft, we will get volume without depth. If we treat verification as clerical work, we will hollow out accountability. If we choose our future intentionally, however, and design systems that preserve judgment, reward learning, and keep ownership meaningful, we can have both productivity and craft, not as a tradeoff, but as a reinforcing pair.

Tools are changing. Abstractions are shifting. But responsibility remains the same: to create software in a sustainable way that benefits humanity. AI is already transforming software engineering. The question is how we will shape that transformation and toward what end.

Related Material

AI Where It Matters: Where, Why, and How Developers Want AI Support in Daily Work
Rudrajit Choudhuri, et al.
https://arxiv.org/abs/2510.00762

What Needs Attention? Prioritizing Drivers of Developers' Trust and Adoption of Generative AI
Rudrajit Choudhuri, et al.
https://arxiv.org/abs/2505.17418

Time Warp: The Gap Between Developers' Ideal vs Actual Workweeks in an AI-Driven Era
Sukrit Kumar; et al.
https://ieeexplore.ieee.org/abstract/document/11121727

"Maybe We Need Some More Examples:" Individual and Team Drivers of Developer GenAI Tool Use.
Courtney Miller, et al.
https://arxiv.org/abs/2507.21280

Thinking Less, Trusting More: GenAI's Impacts on Students' Cognitive Habits
Rudrajit Choudhuri, Christopher Sanchez, Margaret Burnett, Anita Sarma
https://arxiv.org/abs/2601.22430

References

1. Butler, J., Jaffe, S., Janßen, R., Baym, N., Hecht, B., Hofman, J., Rintel, S., Sarrafzadeh, B., Sellen, A., Vorvoreanu, M., Teevan, J. (Eds.). 2025. Microsoft new future of work report 2025. Microsoft Research Tech Report MSR-TR-2025-58; https://aka.ms/nfw2025.

2. Choudhuri, R., Badea, C., Bird, C., Butler, J., DeLine, R., Houck, B. 2025. AI where it matters: where, why, and how developers want AI support in daily work. In IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).

3. Choudhuri, R., Sanchez, C., Burnett, M., Sarma, A. 2026. Why Johnny can't think: GenAI's impacts on cognitive engagement. arXiv preprint 2601.22430; https://arxiv.org/html/2601.22430v1.

4. Choudhuri, R., Trinkenreich, B., Pandita, R., Kalliamvakou, E., Steinmacher, I., Gerosa, M., Sanchez, C., Sarma, A. 2025. What needs attention? Prioritizing drivers of developers' trust and adoption of generative AI. arXiv preprint 2505.17418; https://arxiv.org/abs/2505.17418.

5. Choudhuri, R., Trinkenreich, B., Pandita, R., Kalliamvakou, E., Steinmacher, I., Gerosa, M., Sanchez,

C., Sarma, A. 2025. What guides our choices? Modeling developers' trust and behavioral intentions towards GenAI. In Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE), 1691-1703; https://dl.acm.org/doi/10.1109/ICSE55347.2025.00087.

6. Dell'Acqua. F., McFowland III, E., Mollick, E. R., Lifshitz-Assaf, H., Kellogg, K., Rajendran, S., Krayer, L., Candelon, F., Lakhani, K. R. 2023. Navigating the jagged technological frontier: field experimental evidence of the effects of AI on knowledge worker productivity and quality. Harvard Business School Working Paper 24-013; https://www.hbs.edu/faculty/Pages/item.aspx?num=64700.

7. Kumar, S., Goel, D., Zimmermann, T., Houck, B., Ashok, B., Bansal, C. 2025. Time warp: the gap between developers' ideal vs actual workweeks in an AI-driven era. In IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP),12-22; https://ieeexplore.ieee.org/document/11121727.

8. Lee, H.-P., Sarkar, A., Tankelevitch, L., Drosos, I., Rintel, S., Banks, R., Wilson, N. 2025. The impact of generative AI on critical thinking: self-reported reductions in cognitive effort and confidence effects from a survey of knowledge workers. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. Article no. 1121, 1-22; https://dl.acm.org/doi/10.1145/3706598.3713778.

9. Obi, I., Butler, J., Haniyur, S., Hassan, B., Storey, M.A. Murphy, B. 2025. Identifying factors contributing to "bad days" for software developers: a mixed-methods study. In IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 1-11; https://ieeexplore.ieee.org/document/11121682.

10. Storer, K. M. 2024. How gen AI affects the value of development work. DORA (DevOps Research and Assessment); https://dora.dev/research/ai/value-of-development-work.

Rudrajit Choudhuri *is a Ph.D. student at Oregon State University focusing on human-centric AI for software engineering. He uses large-scale empirical studies, causal experiments, and behavioral analyses to understand the cognitive processes that shape how developers collaborate with AI and to design interfaces that foster appropriate reliance on AI tools. His work treats human cognition as a first-class design concern, generating insights for building and integrating AI responsibly. Contact him at choudhru@oregonstate.edu.*

Eirini Kalliamvakou *is a research advisor at GitHub and leads research that deciphers developers' motivations, needs, behavior, and how tools support them. Her insights shape product thinking, strategic storytelling, and leadership decisions across the company. She holds a Ph.D. in computer science from the University of Victoria and speaks about developer productivity, happiness, and the impact of AI on individuals, teams, and organizations. Contact her at ikaliam@github.com.*

Brian Houck *is an applied scientist on Microsoft's Engineering Thrive team. His work combines large-scale telemetry analysis, field experiments, surveys, and qualitative research to uncover the technical, cultural, environmental, and organizational factors that shape developer productivity and well-being. He is best known for his work on the SPACE framework for developer productivity and for measuring and improving developer experience in large engineering organizations. Contact him at brian.houck@microsoft.com.*

Thomas Zimmermann *is a Chancellor's Professor and Bren Chair at the University of California, Irvine. His research focuses on empowering people and organizations to create better software more efficiently with AI. He is best known for his pioneering work on systematic mining of software repositories and his empirical studies of software development in industry. He is a Fellow of the ACM, AAAS, and IEEE. Contact him at tzimmer@uci.edu.*