

Available online at [www.jpit.az](http://www.jpit.az)16 (2)  
2025

## Prospects of artificial software engineering

Zafar Jafarov<sup>a</sup>, Atif Namazov<sup>b</sup>, Javid Abbasli<sup>c</sup>, Khalid Nazarov<sup>d</sup>, Sevinj Aliyeva<sup>e</sup>

<sup>a,b,c,d,e</sup> Azerbaijan Technical University, H.Javid ave 25, AZ 1073 Baku, Azerbaijan

<sup>a</sup> [zafar.cafarov@aztu.edu.az](mailto:zafar.cafarov@aztu.edu.az); <sup>b</sup> [atif.namazov@aztu.edu.az](mailto:atif.namazov@aztu.edu.az); <sup>c</sup> [cavid.abbasli@aztu.edu.az](mailto:cavid.abbasli@aztu.edu.az); <sup>d</sup> [khalid.nazarov@aztu.edu.az](mailto:khalid.nazarov@aztu.edu.az);

<sup>e</sup> [sevinj.aliyeva@aztu.edu.az](mailto:sevinj.aliyeva@aztu.edu.az)



<sup>a</sup> <https://orcid.org/0009-0001-9327-1779>; <sup>b</sup> <https://orcid.org/0009-0008-2252-7328>; <sup>c</sup> <https://orcid.org/0009-0005-8982-0164>;

<sup>d</sup> <https://orcid.org/0009-0003-4010-2001>; <sup>e</sup> <https://orcid.org/0009-0005-9214-3723>;

### ARTICLE INFO

#### Keywords:

Artificial software engineering  
Software development automation  
Github copilot  
Human–AI collaboration  
AI ethics in programming

### ABSTRACT

The merging of AI and software engineering marks a defining moment: intelligent systems now move beyond simple code completion or test automation to become active partners in each phase of development. We term this “Artificial Software Engineering,” a collaborative framework where human ingenuity and machine intelligence co-author software—from early prototypes and code generation to debugging and architectural design (Menzies et al., 2019). By looking at platforms like Devin AI and GitHub Copilot, we see clear benefits—faster iterations, deeper error detection—but also face new challenges around trust, ownership, legal responsibility, and maintaining AI-influenced code over time. Rather than treating automation as an end goal, we argue that this emerging discipline demands fresh thinking about ethics, team dynamics, and design practices. Ultimately, the most successful software will blend human insight with algorithmic strength to drive responsible innovation.

## 1. Introduction

Software engineering is in the midst of a dramatic shift. What was once a craft performed entirely by human hands—writing code step by step, designing architectures on paper, and running countless tests—now involves another player: artificial intelligence. In the past few years, AI has moved beyond simple helpers (like code completion or automated tests) to join developers as a genuine partner in creating software (Ahmed et al., 2025). Today’s tools range from smart suggestion engines that predict your next line of code to autonomous agents that can spin up a working prototype and even deploy it. As these abilities improve, the line between “what engineers do” and “what AI does” grows ever thinner. We call this emerging way of working “Artificial Software Engineering”—an approach

where intelligent systems weave into every phase of development, from early planning to final rollout (Menzies, Williams, & Zimmermann, 2019). Yet, despite all the buzz, there’s surprisingly little careful study of this shift. Conversations often swing between overhyped sales pitches and narrowly focused performance reports, without tackling the bigger picture. How does AI really fit into a team? What new risks does it introduce? And how do developers change their daily routines when a machine shares their workspace? This paper aims to fill that gap. We look at real usage of tools like GitHub Copilot and Devin AI to see where they shine and where they stumble. We propose a simple framework for how AI joins each stage of the development cycle—ideation, coding, testing, deployment—and highlight the sticky questions of ownership, responsibility, teamwork, and maintenance that

Received 3 March 2025, Received in revised form 5 May 2025, Accepted 19 May 2025

<http://doi.org/10.25045/jpit.v16.i2.06>

2077-4001/© 2025 This is an open access article under the CC BY license (<https://creativecommons.org/licenses/by/4.0/>).

come along for the ride (Amershi et al., 2019). In the end, we believe software's future won't be a contest of humans versus machines, but a collaboration between the two. To make that partnership work, we need fresh mindsets, new processes, and software designs built for a world where people and algorithms build side by side.

## 2. Background and Related Work

The use of AI in software development isn't brand-new, but its pace and scale have surged lately. In the 1980s and '90s, we experimented with expert-system shells and rule-based engines to tackle small chores—code linting, simple static checks, or generating basic test cases (Islam M et al., 2023). Those early tools, however, buckled under the weight of growing codebases: their rigid rules and fixed knowledge quickly fell out of step with real-world complexity (Menzies, Williams, & Zimmermann, 2019).

When machine learning and neural networks entered the scene, everything shifted. Models trained on vast code repositories began to spot non-obvious patterns, surface hidden bugs, and even guess what a developer might type next. Natural language processing added another layer, allowing these assistants to interpret comments, draft documentation snippets, and “talk” with engineers in everyday language.

The real game-changer arrived with large language models spun into coding environments. GitHub Copilot—built on OpenAI's Codex—can instantly turn a plain-English prompt into working code, shortening the path from idea to implementation (Kogan & Palen, 2018). Other contenders, like DeepMind's AlphaCode and Devin AI, are pushing further—tackling end-to-end tasks, from sketching out an algorithm to running automated tests on the finished product.

Scholars have been quick to document these advances: graph-based networks for code reasoning (Allamanis et al., 2018), deep-learning bug detectors outpacing classic static analyzers (Pradel & Sen, 2018), and ML-driven refactoring techniques that clean up messy code. Surveys by Liu et al. (2020) and Yang et al. (2022) map AI's expanding footprint in test automation, defect prediction, and code synthesis.

Yet most of this work either zeroes in on narrow tool benchmarks or floats lofty predictions without tying them to day-to-day developer experiences. We still lack a grounded, lifecycle-

wide look at how AI reshapes every step of software creation—from gathering requirements and drafting prototypes through to deployment and maintenance (Ahmed et al., 2025). Questions around trusting AI suggestions, preserving code quality, and defining who “owns” machine-generated code often get relegated to the sidelines (Amershi et al., 2019). And the rise of hybrid teams—where humans and AI agents coauthor software—raises thorny issues about responsibility and collaboration that few have tackled head-on (Harman, M., & Jones, B. F. 2001).

This paper tackles these gaps. By combining real-world case studies of leading AI tools with a fresh framework for understanding their integration, we argue that “Artificial Software Engineering” isn't a distant dream—it's happening now. Making it work demands human-centered approaches, new collaboration models, and software designs built for a world where people and intelligent systems build side by side (Amershi et al., 2019; Ahmed et al., 2025).

## 3. Real-World Tool Analysis & Case Studies

The arrival of AI-driven development assistants has reshaped the software engineering process, moving it away from purely manual, rule-based routines toward environments where humans and algorithms work side by side. To see how this plays out in practice, we'll look at several prominent tools—grouped by their main roles: creating code, finding and fixing errors, producing documentation, and even taking on entire projects autonomously.

### 3.1. GitHub Copilot: Beyond Autocomplete

GitHub Copilot, powered by OpenAI's Codex, is now a fixture inside editors like VS Code and JetBrains IDEs. It does more than finish your current line—it can draft whole functions or propose boilerplate you didn't write yourself. Because it leans heavily on patterns common across public repositories, many developers find themselves writing in those same familiar styles—even when a different approach might fit better. In a 2022 survey, 88 percent of users said Copilot boosted their productivity, yet over 40 percent confessed to accepting suggestions without fully grasping them (Kogan & Palen, 2018). That trade-off—speed versus understanding—raises real questions about code quality and long-term

maintainability (Menziez, Williams, & Zimmermann, 2019).

### 3.2. Devin AI: A Self-Sufficient Engineer

Released by Cognition Labs in 2024, Devin AI aims to be more than a helper—it claims to be the first “AI software engineer.” Rather than waiting for prompts in your IDE, Devin lives in its own mini operating environment, complete with a command line, browser, and editor. In demos, it has built simple websites, wired up video-processing pipelines, and even closed bug tickets found on GitHub—all without a human typing a single line. But because access is still limited, we don’t yet know how Devin handles massive corporate codebases, ever-changing requirements, or the thorny issues of who legally owns its output (Ahmed et al., 2025)

### 3.3. Codium AI, Tabnine, and CodeWhisperer: Niche Specialists

Not every AI assistant aims to do everything. Codium AI zeroes in on test code—generating thoughtful unit tests and explaining coverage gaps, then slotting itself into existing CI/CD workflows.

Tabnine focuses on on-premises privacy, letting enterprises run completions against local models so proprietary code never leaves their servers.

Amazon CodeWhisperer ties deeply into AWS services, weaving in security scans and compliance checks as part of its suggestion engine. These tools show a shift toward lightweight, task-focused agents that can be composed together inside larger development ecosystems (Menziez, Williams, & Zimmermann, 2019).

### 3.4 Real-World Friction

When teams actually deploy these assistants, familiar challenges emerge:

- Skill atrophy in junior developers who lean too heavily on AI hints (Amershi et al., 2019).
- Context gaps when suggestions ignore an application’s architecture or legacy quirks (Allamanis et al., 2018).
- Hidden vulnerabilities introduced by generated snippets that no one fully audits (Shimmi et al., 2025).
- Inherited biases carried over from the public code the models learned on (Menziez, Williams, & Zimmermann, 2019).
- These issues remind us that sprinkling AI

into a workflow isn’t just an upgrade—it forces us to rethink how we design, review, and secure software.

### 3.5 Embracing a Hybrid Workflow

What works best in practice is not total automation, but a partnership: let AI handle repetitive, boilerplate work while humans steer the creative, architectural, and ethical decisions (Amershi et al., 2019). In that model, teams set the vision and guardrails, and intelligent tools accelerate execution—together forging a more agile, resilient path through the software lifecycle.

## 4. Theoretical Contribution: A Framework for AI Integration in Software Engineering

As AI tools become integral components across all phases of software development, developers often risk focusing excessively on individual functionalities while overlooking the broader context. To provide clarity, this concise, five-stage roadmap illustrates how extensively AI integrates into the development lifecycle—and highlights actionable steps moving forward (Ahmed et al., 2025).

Basic Editor Helpers is a approach of this type of app as spell-check for your code. The system fixes typos, auto-closes brackets, or re-formats lines—nothing more. It doesn’t “know” your project; it just follows simple rules (Pujiharto E et al., 2024).

- Examples: IntelliSense, static linters, syntax highlighters
- Who’s in charge: You write the logic; AI just tidies up.

Another extension like snippet prediction based is your editor that starts to guess your next move—whether that’s a token, a line, or a small block of code. It speeds up repetitive bits, but you still guide the overall design and review every suggestion.

- Examples: GitHub Copilot, Tabnine
- Who’s in charge: You steer the ship; AI handles the oars.

At this level, context-aware partner is one of the artificial tools which AI reads more of your code—perhaps entire files or modules—and offers suggestions that fit your architecture. It can propose refactors, generate tests (Xie & Zhang, 2018), or even learn your team’s style, acting a bit like a junior developer.

- Examples: CodiumAI (for tests), Amazon CodeWhisperer
- Who's in charge: You and AI make decisions together.

While working with team, task-level autonomy handles large scale of task procedures, when, AI can take on full tasks: write a feature from a spec, debug it, even deploy it. It keeps track of what it's doing and talks to other tools on its own, though you still give the final sign-off.

- Examples: Early demos of Devin AI
- Who's in charge: AI drives execution; you validate and approve.

At the highest level, AI based full co-engineering tools joins strategic planning and design discussions. It adapts as requirements change, learns from feedback, and truly collaborates in an ongoing development cycle (Jumper et al., 2021). Now questions of ownership, ethics, and governance become front and center.

- Examples: Cutting-edge research prototypes from top AI labs
- Who's in charge: You set vision and policies; AI and humans build side by side.

By placing your tools and practices on this ladder, you can:

- Pinpoint your current stage. Are you just fixing syntax errors, or already experimenting with autonomous agents?
- Anticipate new challenges. Each rise brings fresh risks—like maintaining code quality, managing ownership of generated code, or reviewing security (Shimmi et al., 2025; Menzies, Williams, & Zimmermann, 2019).

## 5. Challenges and Limitations

As AI tools become common in day-to-day software development, developers often find themselves drawn to appealing promises—quicker coding, improved testing, or simple deployments (Wang S et al., 2023). Yet these benefits also bring challenging considerations around technical reliability, teamwork dynamics, and ethical obligations. These aren't side issues; they go to the heart of making AI a reliable, sustainable partner in software engineering (Menzies, Williams, & Zimmermann, 2019).

From the building trust and clarity prospect, when an LLM spits out a function or a service stub, the reasoning behind its choices is often hidden (Chen et al., 2021). Developers might paste in that code without a second thought—until something

breaks and nobody truly knows why. Without clear explanations, tracing failures or performing security reviews becomes a guessing game, and the question “Who owns this bug?” quickly turns into “Who can even understand this code?”

For avoiding skill erosion, it is letting AI handle repetitive chores—boilerplate, unit tests, simple debugging—boosts output in the short term. But if junior engineers never write that code themselves, they lose out on essential learning moments. Over time, teams risk becoming dependent on AI for every problem, rather than building the deep problem-solving skills that resilient software projects require (Jordan & Mitchell, 2015).

Another hand, most AI assistants learn from public repos, so they're blind to your company's coding conventions, your bespoke frameworks, or that half-century-old legacy system in production. Aligned with your architecture by snippets that compile but don't fit your architecture, or worse, violate hidden business rules. In complex systems, even small mismatches can trigger costly knock-on effects (Kogan & Palen, 2018).

On the other hand, for taking account to ethical and legal grey areas, as AI generates more and more of our code, questions about authorship and IP naturally arise. Who holds the copyright on that routine CRUD endpoint it wrote? At the same time, AI models trained using public codebases may accidentally incorporate security issues or biased coding habits, spreading these problems widely. There's also a concern that broader availability of advanced AI tools might increase the gap between large tech companies and smaller developers or open-source communities. These matters require expertise in law, ethics, and public policy—not just engineering (Menzies, Williams, & Zimmermann, 2019).

When we need to avoid tool overload, the AI tool landscape is exploding: one plugin for docs, another to refactor, a third to generate tests—and that's before you add niche solutions for performance tuning or security scans (Polu, 2025). Juggling dozens of overlapping assistants can actually slow you down, not speed you up. Without shared interfaces or an orchestration layer, you risk replacing one set of headaches—manual tasks—with another: fractured workflows and tool fatigue (Kogan & Palen, 2018).

Addressing these challenges is not optional—it is essential. The future of artificial software engineering depends not only on advancing AI

capabilities, but also on building trustworthy systems, human-centered workflows, and inclusive governance models (Amershi et al., 2019; Ahmed et al., 2025). Researchers, developers, and organizations must actively shape this future—not as passive adopters of tools, but as co-designers of the systems and cultures that surround them.

## 6. Future Outlook & Open Questions

The integration of AI into software engineering isn't just a fad—it's upending how we dream up, build, test, and run our code. Right now we're only scratching the surface. As AI gets smarter and our problems get tougher, we'll wind up in a world where people and machines really work side by side—and even invent whole new ways to engineer software. But with that promise comes a heap of questions we haven't answered yet: technical puzzles, ethical tightropes, and social challenges we'll have to sort out if AI-powered development is going to earn our trust.

Picture an smarter, self-tweaking IDEs that really knows your project: it watches how your team codes, figures out your quirks, and then suggests tweaks—maybe a cleaner design here, a performance tweak there—without you having to ask. That's the dream of an adaptive environment. The catch? We need AI that can learn without accidentally breaking things in production, and do it in a way we can still understand. Otherwise, we'll end up with "helpful" suggestions we can't explain.

Today's AI feels more like a helpful intern than a true teammate. The next chapter is co-ownership: you and an AI agent splitting up tasks, talking through trade-offs, and genuinely sharing the end result. How do you even sketch out that relationship? Will we need roles like "AI coach" or "collaboration designer" to make sure both sides pull their weight?

When an AI tool sets up your deployment pipeline or auto-writes a critical module, who do you call if it goes sideways? And how do you prevent hidden biases or security holes from slipping through? These aren't just developer questions—they touch on law, ethics, and public policy. We need new ways to audit AI decisions, pin down accountability, and make sure these tools play by the rules we all agree on.

Right now the AI scene feels like a crowded bazaar: one plugin for refactoring, another for docs, a third for tests. Someday, it might make sense to weave these into a single, smooth platform that handles your entire lifecycle. But

that raises its own worries—will one company end up controlling everything? Can we keep things modular so you can swap in new tools without tearing down the whole house?

Sure, AI can pump out code faster. But is that code better? Does it help teams think more creatively, or does it make us lazy? Does working alongside an AI change how we solve problems five years down the road? We need fresh metrics, long-term studies, and honest conversations about how engineers and AI actually co-create value.

In short, building the future of software with AI won't be just a tech project—it'll be a team sport that pulls in experts from coding, design, ethics, law, and beyond. Only by mixing all those voices can we craft tools and processes that are powerful, transparent, and—most importantly—human-centered (Amershi et al., 2019).

## 7. Conclusion

Artificial Software Engineering isn't just about new tools—it's about reimagining the very way we build and care for software. As AI steps in to help with coding, testing, and architecture, the line between what humans create and what machines generate is shifting daily.

In this paper, we've surveyed prospects of today's AI-powered helpers—GitHub Copilot, Devin AI, and the like—then laid out a simple, five-stage view of their evolution, from basic suggestions all the way to true co-engineering. Along the way, we've highlighted both the big wins (faster prototyping, smarter debugging) and the tricky spots (trust, ownership, long-term upkeep).

What comes into focus is a blended future: software shaped not by people or machines alone, but by their partnership. Getting there means adopting new habits—peer reviews for AI output, clear ethical guidelines (Amershi et al., 2019), and a willingness to learn from each other. Success won't be about how much code we offload to algorithms; it'll be about how those algorithms help us write better, more reliable software—software that we understand, we trust, and we're proud to stand behind.

## References

- Ahmed, I., Aleti, A., Cai, H., Chatzigeorgiou, A., He, P., Hu, X., Pezzè, M., Poshyvanyk, D., & Xia, X. (2025). Artificial Intelligence for Software Engineering: The Journey so far and the Road ahead. *ACM Transactions on Software Engineering and Methodology*, 34(5), 1-27. <https://doi.org/10.1145/3719006>

- Allamanis, M., Brockschmidt, M., & Khademi, M. (2018). Learning to represent programs with graphs. In International Conference on Learning Representations (ICLR) (pp. 1-17). <https://doi.org/10.48550/arXiv.1711.00740>
- Amershi, S., Cakmak, M., Knox, W. B., & Kulesza, T. (2014). Power to the People: The role of humans in interactive machine learning. *AI Magazine*, 35(4), 105-120. <https://doi.org/10.1609/aimag.v35i4.2513>
- Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., et al. (2021). Evaluating large language models trained on code. ArXiv. <https://doi.org/10.48550/arXiv.2107.03374>
- Harman, M., & Jones, B. F. (2001). Search-based software engineering. *Information and Software Technology*, 43(14), 833–839. [https://doi.org/10.1016/S0950-5849\(01\)00189-6](https://doi.org/10.1016/S0950-5849(01)00189-6)
- Islam, M., Khan, F., Alam, S., & Hasan, M. (2023). Artificial intelligence in software testing: A systematic review. In Proceedings of the IEEE Region 10 Conference (TENCON) (pp. 524–529). IEEE. <https://doi.org/10.1109/TENCON58879.2023.10322349>
- Jordan, M. I., & Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245), 255–260. <https://doi.org/10.1126/science.aaa8415>
- Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., Tunyasuvunakool, K., Bates, R., Žídek, A., Potapenko, A., Bridgland, A., Meyer, C., Kohl, S. A., Ballard, A. J., Cowie, A., Nikolov, S., Jain, R., Adler, J., Back, T., et al. (2021). Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873), 583-589. <https://doi.org/10.1038/s41586-021-03819-2>
- Kogan, M., & Palen, L. (2018). Conversations in the eye of the storm: At-scale features of conversational structure in a high-tempo, high-stakes microblogging environment. In Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (pp. 1-13). <https://doi.org/10.1145/3173574.3173658>
- Polu, O. R. (2025). AI-Driven Automatic Code Refactoring for Performance Optimization. *International Journal of Science and Research*, 14(1), 1316-1320. <https://doi.org/10.21275/SR25011114610>
- Pradel, M., & Sen, K. (2018). Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 1-25. <https://doi.org/10.1145/3276517>
- Pujiharto, E. W., Tikasni, E., Lewu, R., Sudirman, S., & Utami, E. (2024). Systematic literature review on software requirement engineering in 5.0 Industry: Current practices and future challenges. *International Journal of Advanced Science Computing and Engineering*, 6(3), 104–108. <https://doi.org/10.62527/ijasce.6.3.152>
- Shimmi, S., Okhravi, H., & Rahimi, M. (2025). AI-based software vulnerability detection: A systematic literature review. <https://doi.org/10.48550/arXiv.2506.10280>
- Wang, S., Huang, L., Gao, A., Ge, J., Zhang, T., Feng, H., Satyarth, I., Chen, C., Liu, Z., & Wang, Q. (2023). Machine/deep learning for software engineering: A systematic literature review. *IEEE Transactions on Software Engineering*, 49(3), 1188–1231. <https://doi.org/10.1109/TSE.2022.3173346>
- Yang, Y., Xia, X., Lo, D., & Grundy, J. (2022). A survey on deep learning for software engineering. *ACM Computing Surveys (CSUR)*, 54(10s), 1-73. <https://doi.org/10.1145/3505243>