REVIEW ARTICLE

Is AI Code Generation Undermining Developers' Problem-Solving Skills?

Moomna Nazir[1] **and Yasir Arif**[2,*]

[1] Department of Computer Science, COMSATS University Islamabad, Sahiwal 57000, Pakistan
[2] Department of Computer Science, Global Institute, Lahore 54000, Pakistan

Abstract

The rise of AI tools such as GitHub Copilot and ChatGPT has reshaped software development by providing substantial support for coding and debugging tasks. Although these tools enhance productivity and reduce routine workload, existing research has largely emphasized short-term efficiency gains, leaving their long-term cognitive and pedagogical effects insufficiently explored. This study investigates the cognitive trade-offs associated with sustained reliance on generative AI, with particular attention to students and junior developers. Recent empirical findings indicate that excessive dependence on AI assistance may weaken deep debugging skills, impede conceptual understanding, and challenge established educational practices in software engineering. To address these concerns, we synthesize empirical studies published since 2020 and draw on contemporary pedagogical theories to propose a structured framework for balanced AI integration. The proposed hybrid model shifts emphasis from full automation to a learning-oriented process that foregrounds exploration, human reasoning, and critical evaluation. It comprises three iterative phases—Detect (AI-assisted exploration), Engage (manual problem-solving and algorithmic reasoning), and Verify (AI-supported refinement)—designed to preserve core cognitive competencies while effectively leveraging automation. The study underscores the importance of aligning AI tool usage with pedagogical objectives, ensuring that system design promotes understanding rather than output generation alone. These findings have implications for curriculum design in computer science education and for industrial strategies aimed at sustaining developer expertise in increasingly automated environments.

Keywords: AI code generation, developer cognition, gitHub copilot, code automation, programming pedagogy.

1 Introduction

The recent introduction of tools like GitHub Copilot, ChatGPT, and StarCoder has changed how software development is practiced and approached. These tools are expected to save coding time, help reduce developer workload, and help untangle sophisticated libraries or APIs, which enhances productivity and satisfaction [1].

Submitted: 02 August 2025
Accepted: 30 October 2025
Published: 27 January 2026

Vol. 2, **No.** 1, 2026.
10.62762/JSE.2025.847963

***Corresponding author:**
✉ Yasir Arif
yasirarif268@gmail.com

Citation
Nazir, M., & Arif, Y. (2026). Is AI Code Generation Undermining Developers' Problem-Solving Skills?. *ICCK Journal of Software Engineering*, 2(1), 1–10.

© 2026 by the Authors. Published by Institute of Central Computation and Knowledge. This is an open access article under the CC BY license (https://creativecommons.org/licenses/by/4.0/).

These assumptions have been validated with more research. In one study, Peng et al. [2] ran a randomized experiment, and in a group with access to GitHub Copilot, GitHub users completed an HTTP server programming task in an average of 55.8% less time compared to the group without access to Copilot. More interestingly, the rate of success on the given project did not vary significantly between the groups. In another study, Microsoft's enterprise data, which also comprised user surveys, showed that 73% of Copilot enterprise users reported agreement with the statement that the tool assists them in task acceleration, with 68% asserting improved task quality, often coupled with less effort [3].

Despite the undeniable productivity gains, an unresolved concern persists: Does reliance on AI-generated code compromise developers' problem-solving skills? Programming is fundamentally about logical reasoning, debugging, and deep understanding, not merely generating outputs. When AI assumes too many cognitive routines, particularly for less experienced developers, it risks eroding essential abilities such as debugging, logical reasoning, and algorithmic design [4]. This tension between efficiency and skill preservation constitutes the key problem addressed in this study.

The scope of this study is deliberately focused on early-career developers and students, as this group is most vulnerable to overreliance on AI tools. The discussion emphasizes widely adopted tools such as GitHub Copilot, ChatGPT, and StarCoder, which represent the leading edge of generative AI in software development.

The contributions of this paper are threefold, as given below:

1. It synthesizes recent empirical evidence (2020 onwards) to critically evaluate the cognitive trade-offs of AI-assisted programming, particularly for junior developers.

2. It integrates insights from pedagogical frameworks to highlight how AI tools can be aligned with skill development in software engineering education.

3. It proposes a structured three-phase framework Detect, Engage, Verify that balances automation with human reasoning, offering practical guidance for both educational curricula and industrial practice.

Emerging educational research supports this concern. For example, an investigation into student users of AI-based coding applications like ChatGPT and StarCoder revealed that, although they were more likely to complete tasks, users exhibited surface reasoning, a lack of deeper understanding, and creativity relative to manually coding peers [5]. Synthesizing across such studies reveals a consistent pattern: while AI accelerates task completion, it simultaneously risks undermining the deeper cognitive processes that are essential for long-term skill development. This synthesis strengthens the rationale for the current study.

This article critically explores the dual nature of AI code generation: its tangible productivity benefits and its subtle risks to essential cognitive capabilities. Drawing exclusively from empirical evidence dated 2020 onward, the study examines the interaction of developers with AI-generated code, the nature of cognitive trade-offs involved, and the broader implications for education and professional development. The study aims to evaluate whether AI assistance complements or undercuts the problem-solving mindset at the core of software engineering.

The remainder of this paper is organized as follows. Section 2 reviews the evolution of AI tools in software development. Section 3 analyzes the impact of these tools on developers' problem-solving skills. Section 4 examines the cognitive trade-offs between skill acquisition and tool dependence. Section 5 discusses pedagogical implications in software engineering education, while Section 6 outlines pedagogical foundations for structured AI use. Section 7 presents the proposed three-phase framework for balanced AI adoption. Section 8 explores industrial perspectives and developer responsibilities. Finally, Section 9 concludes the paper and highlights future research directions.

2 Evolution of AI Tools in Software Development

The landscape of software development has undergone a profound shift with the progressive integration of artificial intelligence. What started as AI-assisted coding capabilities in integrated development environments (IDEs) has evolved to fully automated systems that can generate entire code segments with little human input. One of the more prominent milestones was Microsoft's IntelliSense, which came out in the late 90s. It provided function and parameter

hints as well as real-time syntax suggestions. While IntelliSense was not driven by machine learning, its interface opened the doors to AI-assisted coding.

With the advent of AI, there was a paradigm shift in automation. By the early 2020s, new models like TabNine integrated deep learning and GPT-style transformers, improving code prediction across numerous languages. Unlike older systems that relied on keyword-triggered autocompletion, TabNine utilized immense code libraries to make contextual suggestions. This signified the move from static code completion to more dynamic probabilistic code suggestion.

The release of GitHub Copilot in 2021, powered by OpenAI's Codex model, represented a major turning point. Copilot not only completed lines of code but could also generate entire functions or classes based on plain-language comments. Its integration with editors like VS Code blurred the line between code suggestion and code generation. Developers were now engaging with AI as a pair programming partner, one that could offer real-time solutions informed by billions of lines of training data.

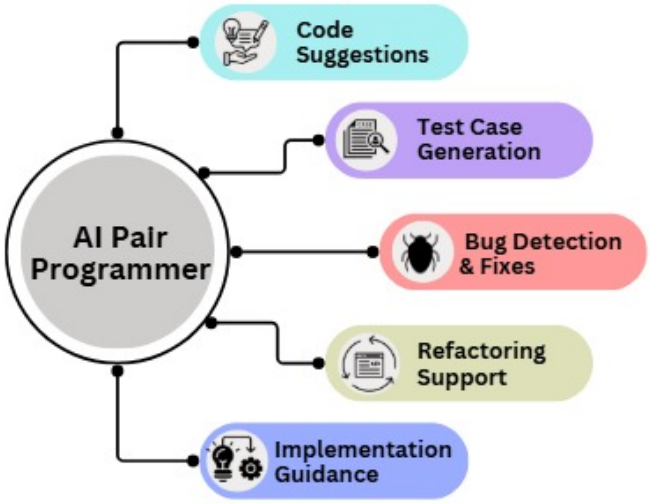


Figure 1. Capabilities of an AI pair programmer.

With the arrival of large-scale language models like ChatGPT and, later, StarCoder, the capacity for full code generation expanded further. These models can interpret complex queries, generate boilerplate, optimize algorithms, and even explain code behavior, all through conversational interaction. However, early user studies indicate a gap between developer expectations and the actual experience with such tools, highlighting the importance of optimizing human-AI interaction in real-world development contexts [7]. Unlike Copilot, which operates contextually within the editor, models like ChatGPT exist outside the IDE but offer broader, multi-turn interactions that support not just coding but reasoning and debugging assistance. Figure 1 shows the working of an AI Pair Programmer, including code suggestions, test case generation, bug detection, refactoring support, and implementation guidance.

AI tools today can be broadly categorized along a continuum. Code completion tools, such as IntelliSense and early versions of TabNine, offer syntactic assistance. Code suggestion systems, including Copilot, recommend multi-line logic within a developer's context. Full code generation systems, such as ChatGPT or StarCoder, can autonomously produce functioning code from natural language specifications, often with minimal human editing required. Figure 2 categorizes various AI coding tools based on their capabilities, such as full code generation, code suggestion, and code completion.

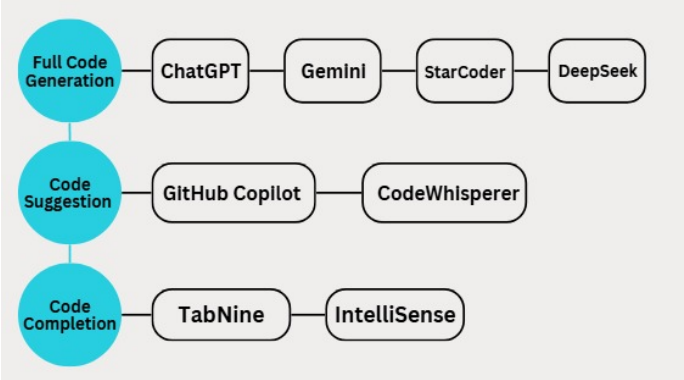


Figure 2. Categorization of AI code generation tools.

These tools are now also embedded within broader development workflows. In modern CI/CD pipelines, AI assists in test case generation, log analysis, and even code review. In collaborative environments, AI serves as a surrogate pair programmer available constantly, scalable across teams, and fluent in multiple programming paradigms. However, as this integration deepens, it becomes increasingly important to reflect on these capabilities reshaping not only coding tasks, but the skills developers are expected to cultivate. Figure 3 presents a timeline of key advancements in AI coding tools, from IntelliSense in 1998 to the emergence of automated agents in 2025.

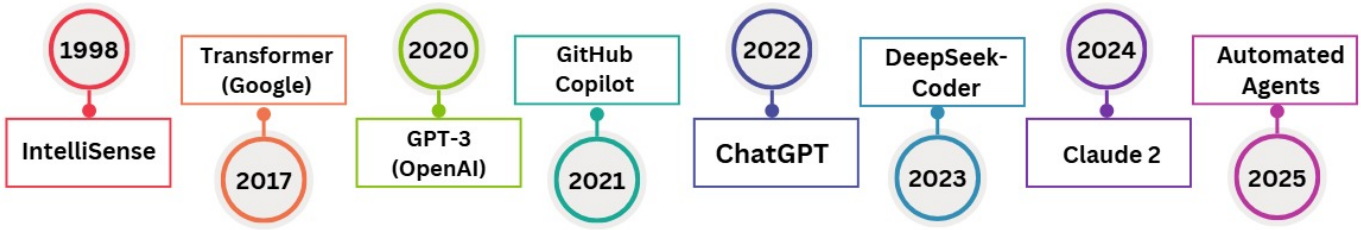


Figure 3. Timeline of key advancements in AI for code generation.

3 Impact on developers' problem-solving skills

AI-powered code generation tools such as GitHub Copilot and ChatGPT have yielded measurable productivity benefits. A peer-reviewed study published in 2024 surveyed developers using Copilot and matched subjective productivity reports to usage data. The findings indicated that developers accepting Copilot suggestions demonstrated improved satisfaction, performance, and reduced cognitive load, especially among junior developers, with these gains triangulated across multiple metrics [6]. In other software engineering domains, such as software defect prediction, machine learning models have achieved impressive accuracy [8].

Field observations within enterprise environments further supported these findings. The utilization of GitHub Copilot in the workplace has resulted in a tangible productivity boost, evidenced by the increase in the rate of acceptance of the suggested code edits in conjunction with the rise in the number of pull requests and overall developer happiness, indicating tangible efficiency improvements [9].

Despite the persuasive nature of productivity gains, educational research paints a different picture, especially regarding more profound cognitive engagement. Consider the 2025 article published in the International Journal of Artificial Intelligence in Education, which examined undergraduate programming students in the context of GenAI tools. The authors noted enhanced tool-driven efficiency and creativity perception, but students found AI code correction during assessments much more challenging than solving problems they had written themselves. This gap suggests reduced debugging and reasoning skills when relying on generated output [10].

Together, AI tools boost developer productivity and satisfaction, particularly for routine or boilerplate tasks. Similarly, in software defect prediction research, ensuring developer interpretability of model outputs has been emphasized to preserve engineering judgment. But foundational problem-solving skills, such as debugging, reconstructing logic, or verifying edge case behavior, appear to weaken if reliance on AI becomes normative [13]. Novice developers are most at risk, as they may accept AI-generated code without sufficient comprehension or critical scrutiny.

In summary, while there are benefits of AI-assisted development, it also cautions that unchecked dependence may undermine the cognitive practices essential to robust software engineering. Figure 4 presents a comparative overview of the positive and negative effects of AI coding tools on developers, highlighting benefits such as increased speed and exposure to new techniques, alongside drawbacks like reduced debugging depth and over-reliance on AI.

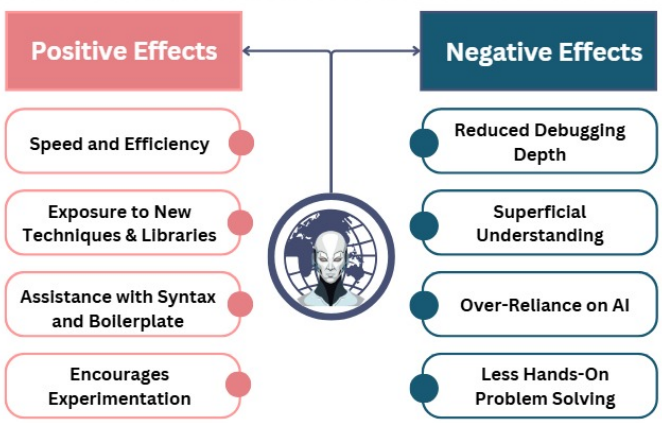


Figure 4. Positive and negative effects of AI on programming and solving skills.

4 Cognitive TradeOff: Skill Acquisition vs Tool Dependence

As AI-powered code generation becomes more prevalent in programming workflows, it is vital to understand the underlying cognitive trade-offs. On the one hand, developers benefit from the automation of routine tasks. On the other hand, reliance on AI may

reduce opportunities to cultivate essential reasoning and analytical skills.

A comprehensive meta-analysis published in 2025 evaluated 35 controlled studies between 2020 and 2024, comparing learning outcomes in programming courses with and without AI tools like ChatGPT or Copilot. The results showed significant improvements in task completion time and performance scores (SMD $\approx +0.86$) for AI-assisted learners. Notably, there was no statistically significant advantage in conceptual understanding or ease of comprehension (SMD $\approx +0.16$, $p \approx 0.41$) [14].

In another study published in 2024, researchers compared undergraduate programming students using ChatGPT versus those coding manually. While both groups achieved similar task completion levels, the AI-assisted group exhibited fewer self-initiated debugging strategies and tended to accept AI suggestions uncritically, pointing to reduced engagement with deeper problem-solving processes.

These findings can be interpreted through established cognitive models like Bloom's Taxonomy and the Dreyfus model of skill acquisition. Novice developers who perform tasks at the "remember" or "understand" level; may become overly dependent on AI for code generation, thereby bypassing progression toward the "apply" and "analyze" stages where logic formulation and debugging skills are honed [15]. Consequently, developers may excel at producing output but lack the scaffolding to independently reason about system behavior or design decisions.

5 Pedagogical implications in software engineering education

Integrating AI code-generation tools like Copilot and ChatGPT into programming education offers both opportunities and significant challenges. Educators must balance leveraging these tools' benefits with safeguarding students' development of core computational thinking and problem-solving skills.

A recent quasi-experimental study of 2025 evaluated AI-assisted pair programming using GPT-3.5 and Claude 3 Opus among 234 Java undergraduates. Compared with human–human pairing and individual programming, AI-assisted groups experienced increased intrinsic motivation, decreased programming anxiety, and improved task performance ($p < .001$) [16]. However, despite these affective benefits, interaction quality like perceived collaboration remained highest in traditional peer programming.

Complementing these findings, a systematic review and meta-analysis in 2025 synthesized 35 controlled studies on AI tools in programming education. While results showed improved task completion time and performance (e.g., SMD $\approx +0.86$), there was no significant gain in conceptual understanding or deeper problem-solving ability (SMD $\approx +0.16$, $p \approx 0.41$) [14]. This suggests AI tools assist with execution but may not promote cognitive development unless intentionally integrated.

Another peer-reviewed investigation in 2024 compared undergraduate students using ChatGPT-assisted coding to a control group coding independently. Although assignments were completed successfully, the AI group exhibited fewer self-initiated debugging strategies and showed more passive acceptance of AI-generated solutions [17].

Together, these sources point to critical pedagogical implications, such that using AI to boost motivation and reduce anxiety, but avoid presenting it as a replacement for human guidance and discussion. Moreover, introduce fundamental programming concepts and problem-solving exercises before permitting AI tools. Additionally, require students to test, explain, and critique AI-generated code to foster analytical engagement. Similarly, encourage peer collaboration and human feedback, as AI alone cannot substitute for the social and affective benefits of human-human interaction [16].

6 Pedagogical Foundations of Structured AI Use

The rapid integration of AI code generation tools into software engineering has raised important questions about the preservation of developers' problem-solving skills while benefiting from automation. Drawing on recent peer-reviewed scholarship, it is increasingly evident that a structured, intentional approach to using AI tools is necessary. Rather than allowing unregulated usage that could erode critical reasoning, several researchers advocate phased, cognitively aligned frameworks that retain the learner's active role in the development process. Figure 5 illustrates various AI coding tools categorized according to their primary functions in the software development lifecycle, such as understanding the problem, generating code, fixing bugs, and optimizing performance.

In a 2025 study, authors proposed the GROW-AI

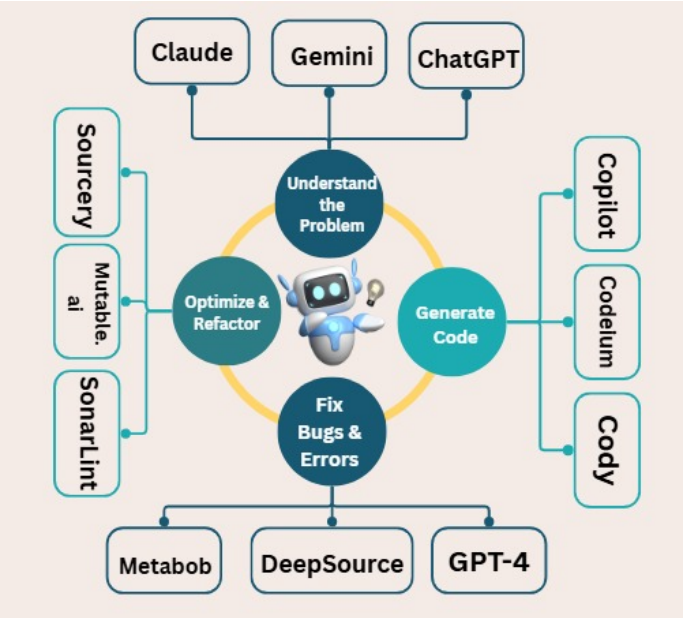


Figure 5. AI tools supporting the software development lifecycle.

framework, developed through thematic analysis of ICT students' use of generative AI tools in coursework. This framework emphasizes the importance of scaffolding AI adoption within educational environments. It encourages learners to engage with AI tools in ways that promote conceptual understanding rather than mechanical code reproduction. Specifically, the study warns that passive code consumption without reflective practice may inhibit the development of transferable problem-solving skills over time. This evidence reinforces the gradually evolving approach concerning student engagement, in which scholars are first instructed on how basic prompting of the AI is done, and progressively sophisticated verification and critical thinking tasks are incorporated in alignment with the scholar's evolving competencies [18].

The AI-Lab framework is another model aligned with constructivist pedagogy. This framework emphasizes that beginner programmers should grapple with mental workload to learn to code effectively. The framework suggests that AI can be utilized as a "learning assistant" during the initial stages of ideation and exploration of new syntactic patterns; however, AI should refrain from being used during the implementation of core logic. This approach ensures that learners remain actively involved with algorithmic design and do not outsource all reasoning to the model. The framework also suggests that learners should be taught to prompt AI critically and AI in general so that its codes are not blindly taken as authoritative [19].

Synthesis of these models, along with cognitive psychology, shows that a practical approach to using AI in development environments is shaping up. This approach suggests working with AI in differentiative phases to draw design inspirations or to know about different coding methodologies in the initial phases. In later phases, a shift to manual coding of the key logic AI to ensure independent reasoning is encouraged. After that step, AI tools can be reintroduced to the process to assist in optimizing, refactoring, or finding errors in the code. This circular process mirrors a human-in-the-loop approach that seeks to avoid or balance complete automation dependence, which is educational in nature, on the principle that mastery is developed through adversity.

These frameworks are beginning to shift instructional design in top computer science departments. These frameworks are also gaining traction in the corporate world, where development teams are looking to balance the speed of delivery with deep, ongoing expertise. These frameworks highlight that AI can be embedded at different stages along a defined progression that balances cognitive development with the deep, critical thinking necessary to develop flexible skills. Figure 6 shows a layered model of AI coding assistance, from basic syntax support to advanced code generation and summarization.

7 Proposed Framework for Balanced AI Use

AI code generation tools integrated into software development environments necessitate a systematic policy framework to mitigate cognitive skill erosion. This challenge calls for an appropriate solution in a model that balances AI use with the natural progression of solving software-related tasks. This model calls for a balanced cadence—rhythmic shifts between thought and automation—where the developer retains control of critical thinking and employs automation in the most appropriate areas. It is divided into three primary iterative phases, which are: Detect, Engage, and Verify.

The rationale for this three-phase design is grounded in cognitive theory and programming pedagogy. Models such as Bloom's Taxonomy and the Dreyfus model of skill acquisition emphasize the progression from basic comprehension to advanced reasoning, making it critical to align AI use with developmental stages of problem-solving. Accordingly, the Detect–Engage–Verify framework is structured to ensure that AI supports learning without replacing essential analytical processes.

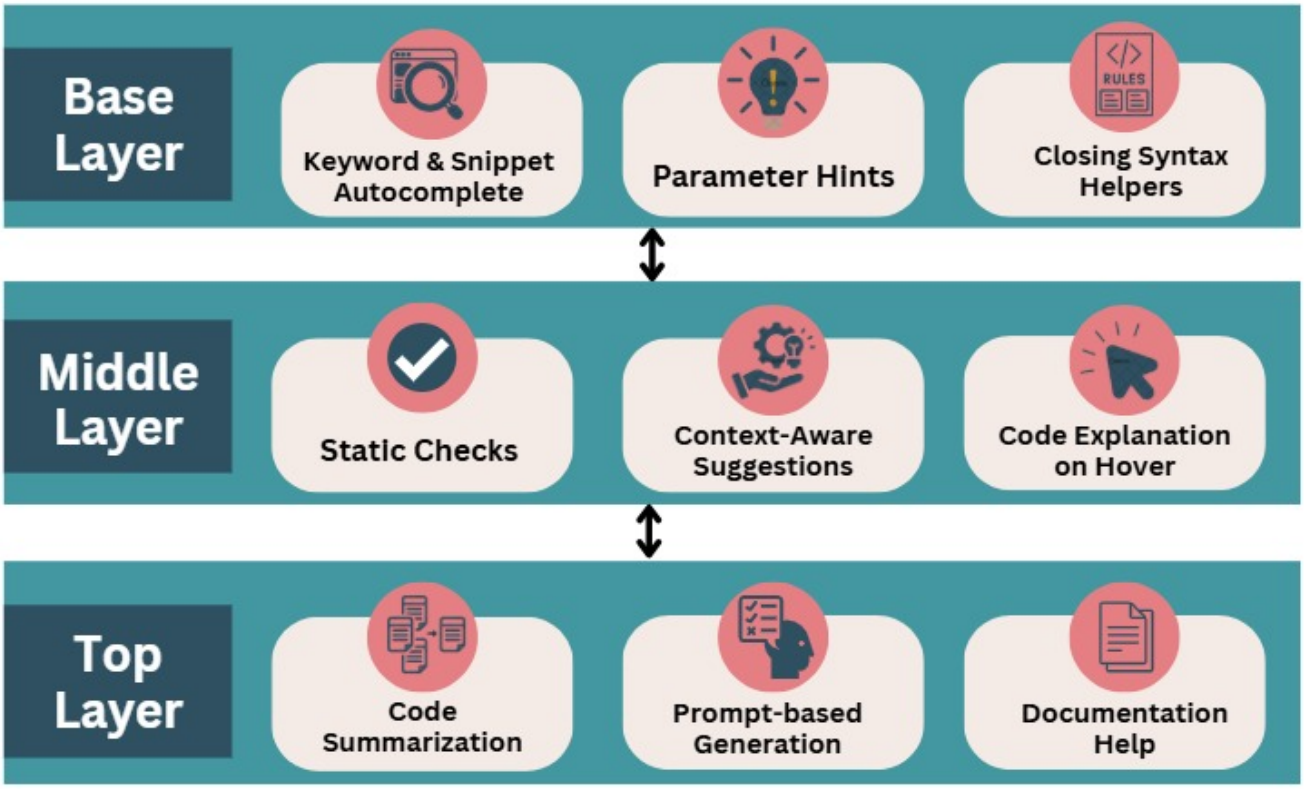


Figure 6. Layered architecture of AI-powered coding assistance.



Figure 7. Phases of AI-assisted software development.

With AI tools in the market, the Detect phase can now be optimized. This phase utilizes AI tools for library search, function comprehension, and brainstorming. The objective is now aimed solely towards aiding the developer in grasping the spatial dimensions of the problem and not towards writing code. In the Engage phase, the developer now takes charge by executing the manual writing of the application and its core logic. This phase helps in enhancing the application of design, algorithmic, and testing skills, which are very crucial for the long term. Resisting AI code suggestions in this phase strengthens the developer's cognitive engagement and helps in long-term learning. The criterion for moving beyond this stage is the achievement of a minimally functional program that executes, even if not optimized.

The last Verify phase brings back AI tools, but this time for refinement tasks. These tasks can be optimizing performance, improving readability, finding edge cases, or recommending better ways to do things. Here, AI helps as a teammate, verifying quality, as opposed to a replacement for human thought. Transition into this stage occurs once the solution achieves basic correctness and the focus shifts to improvement

and robustness. Validation of the framework draws on a systematic synthesis of empirical studies from 2020–2025 and a comparative analysis with existing pedagogical models, including GROW-AI [18] and AI-Lab [19]. The approach also aligns with established practices in AI-assisted software engineering, where machine learning models are routinely validated for reliability before integration into development workflows [11, 12].

Nonetheless, limitations must be acknowledged. First, the validation process incorporates self-reported student feedback, which may introduce subjectivity. Second, while promising in educational settings, the framework's applicability in large-scale industrial environments remains to be confirmed. Third, long-term cognitive outcomes have not yet been empirically tested, underscoring the need for longitudinal research. Figure 7 presents a three-phase AI-assisted workflow combining detection, human input, and AI-based optimization.

This framework provides steps to cut back on the excessive reliance placed on AIs while still maximizing their advantages. Finding the right balance between automation and active learning through a well-defined process on when and how AI is utilized in the coding process is feasible. This aligns with the productivity goals in professional environments where team output needs to balance with skill development, and in educational settings where managing the learner's cognitive load is a central goal.

8 Industrial Perspectives and Developer Responsibilities

The impact of generative AI tools is being felt across the software engineering domain and is leading to a change in the responsibilities of developers, teams, and even the toolmakers. One of the studies looked at the incorporation of AI into the software development life cycle and its consequential effects on productivity, quality of code, and the structure of teams. The authors point out that AI serves to automate a user's mundane or repetitive work, but human judgment is still vital in the processes of validation concerning the correctness, architecture, and security of the automaton code [20, 21].

AI coding assistants have also been studied for their impact on an organization's productivity. As noted in this study, developers reported distinct productivity improvements but also expressed concerns regarding an over-dependence on automation within an organization. As a matter of policy, the study suggests that corporations set clear policies on AI use, particularly in the evaluation of design and review of the codes, where human intelligence is essential. The study shows AI influences the responsibilities of experienced developers and mentors who, as team leaders, check AI-produced code against team expectations and fundamental engineering standards [22].

Also, an empirical study released in 2025 shows that when developers think of AI as an "expert assistant," they tend to take its suggestions too easily. This especially leads to overconfidence in AI's outputs, including when those outputs contain minor logical errors or context misalignments. Comparatively, developers who viewed AI as a partner in collaboration instead of as a decision-maker took a more critical approach and engaged in questioning and interventional coding [23].

As a whole, the integration of AI in businesses rests on upholding a notable difference: AI can derive code, but human professionals need to verify, assess, and provide the necessary context. Developer duties include more than just coding; they involve crafting prompts with intent, assessing results with due diligence, and guiding their peers to view AI as a tool, not a replacement for cognition.

9 Conclusion

Tools like GitHub Copilot and ChatGPT are transforming software development by increasing productivity and minimizing time spent on repetitive coding tasks. On the flip side, there's increasing apprehension about the lack of essential skill development for beginners and novices. Even though beginners and learners are provided the right tools, the risk of fostering a shallow grasp of debugging, reasoning, and design skills due to overreliance on automation is critical. There is a pressing need for educational and industrial bodies to respond by creating policies that balance the use of automation and cognitive functions by the human mind, and the uses of AI that require learners to think, justify, and learn purposefully. Overall, this paper contributes by synthesizing recent empirical evidence on AI-assisted programming, integrating pedagogical frameworks, and proposing a structured three-phase model (Detect–Engage–Verify) that balances automation with human reasoning. These contributions underscore its significance for both education and industry. Future research should focus

on longitudinal assessments of cognitive impacts, the development of pedagogically aligned explainable AI systems, and curriculum reforms in bootcamps and universities. Only through thoughtful integration can AI serve not as a crutch, but as a catalyst for deeper and more responsible software engineering practice.

Data Availability Statement

Not applicable.

Funding

This work was supported without any funding.

Conflicts of Interest

The authors declare no conflicts of interest.

AI Use Statement

The authors declare that no generative AI was used in the preparation of this manuscript.

Ethical Approval and Consent to Participate

Not applicable.

References

[1] Kalliamvakou, E. (n.d.). Research: quantifying GitHub Copilot's impact on developer productivity and happiness. The GitHub Blog. Retrieved August 1, 2025, from https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/

[2] Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590*.

[3] Cambon, A., Hecht, B., Edelman, B., Ngwe, D., Jaffe, S., Heger, A., ... & Teevan, J. (2023). Early LLM-based tools for enterprise information workers likely provide meaningful boosts to productivity. *Microsoft Research. MSR-TR-2023, 43*.

[4] Song, F., Agarwal, A., & Wen, W. (2024). The impact of generative AI on collaborative open-source software development: Evidence from GitHub Copilot. *arXiv preprint arXiv:2410.02091*.

[5] Yilmaz, R., & Yilmaz, F. G. K. (2023). The effect of generative artificial intelligence (AI)-based tool use on students' computational thinking skills, programming self-efficacy and motivation. *Computers and Education: Artificial Intelligence, 4*, 100147. [CrossRef]

[6] Ziegler, A., Kalliamvakou, E., Li, X. A., Rice, A., Rifkin, D., Simister, S., ... & Aftandilian, E. (2024). Measuring github copilot's impact on productivity. *Communications of the ACM, 67*(3), 54-63. [CrossRef]

[7] Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022, April). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts* (pp. 1-7). [CrossRef]

[8] Prather, J., Reeves, B. N., Denny, P., Becker, B. A., Leinonen, J., Luxton-Reilly, A., ... & Santos, E. A. (2023). "It's weird that it knows what i want": Usability and interactions with copilot for novice programmers. *ACM transactions on computer-human interaction, 31*(1), 1-31. [CrossRef]

[9] Bakal, G., Dasdan, A., Katz, Y., Kaufman, M., & Levin, G. (2025). Experience with GitHub Copilot for Developer Productivity at Zoominfo. *arXiv preprint arXiv:2501.13282*.

[10] Kohen-Vacs, D., Usher, M., & Jansen, M. (2025). Integrating generative AI into programming education: Student perceptions and the challenge of correcting AI errors. *International Journal of Artificial Intelligence in Education*, 1-19. [CrossRef]

[11] Ali, M., Mazhar, T., Arif, Y., Al-Otaibi, S., Ghadi, Y. Y., Shahzad, T., ... & Hamam, H. (2024). Software defect prediction using an intelligent ensemble-based model. *IEEE Access, 12*, 20376-20395. [CrossRef]

[12] Ali, M., Mazhar, T., Shahzad, T., Ghadi, Y. Y., Mohsin, S. M., Akber, S. M. A., & Ali, M. (2023). Analysis of feature selection methods in software defect prediction models. *IEEE Access, 11*, 145954-145974. [CrossRef]

[13] Jiang, E., Toh, E., Molina, A., Olson, K., Kayacik, C., Donsbach, A., ... & Terry, M. (2022, April). Discovering the syntax and strategies of natural language programming with generative language models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (pp. 1-19). [CrossRef]

[14] Alanazi, M., Soh, B., Samra, H., & Li, A. (2025). The Influence of Artificial Intelligence Tools on Learning Outcomes in Computer Programming: A Systematic Review and Meta-Analysis. *Computers (2073-431X), 14*(5). [CrossRef]

[15] Bloom, B. S. (1956). Taxonomy of Edu-cational Objectives, Handbook 1: Cognitive Do-main.

[16] Fan, G., Liu, D., Zhang, R., & Pan, L. (2025). The impact of AI-assisted pair programming on student motivation, programming anxiety, collaborative learning, and programming performance: a comparative study with traditional pair programming and individual approaches. *International Journal of STEM Education, 12*(1), 16. [CrossRef]

[17] Sun, D., Boudouaia, A., Zhu, C., & Li, Y. (2024). Would ChatGPT-facilitated programming mode

impact college students' programming behaviors, performances, and perceptions? An empirical study. *International Journal of Educational Technology in Higher Education, 21*(1), 14. [CrossRef]

[18] Chugh, R., Turnbull, D., Kutty, S., Sabrina, F., Rashid, M. M., Morshed, A., ... & Subramani, S. (2025). Generative AI as a learning assistant in ICT education: student perspectives and educational implications. *Education and Information Technologies*, 1-36. [CrossRef]

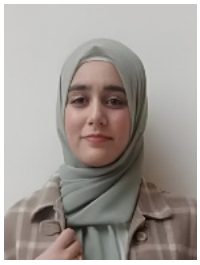
[19] Dickey, E., Bejarano, A., & Garg, C. (2023). Innovating computer programming pedagogy: The AI-lab framework for generative AI adoption. *arXiv preprint arXiv:2308.12258*.

[20] Damyanov, I., Tsankov, N., & Nedyalkov, I. (2024). Applications of Generative Artificial Intelligence in the Software Industry. *TEM Journal, 13*(4). [CrossRef]

[21] Kazemitabaar, M., Chow, J., Ma, C. K. T., Ericson, B. J., Weintrop, D., & Grossman, T. (2023, April). Studying the effect of AI code generators on supporting novice learners in introductory programming. In *Proceedings of the 2023 CHI conference on human factors in computing systems* (pp. 1-23). [CrossRef]

[22] Banh, L., Holldack, F., & Strobel, G. (2025). Copiloting the future: How generative AI transforms Software Engineering. *Information and Software Technology, 183*, 107751. [CrossRef]

[23] Li, Z. S., Arony, N. N., Awon, A. M., Damian, D., & Xu, B. (2024). AI tool use and adoption in software development by individuals and organizations: a grounded theory study. *arXiv preprint arXiv:2406.17325*.



Moomna Nazir received the B.S. degree in Computer Science from COMSATS University Islamabad, Pakistan. Her research interests span artificial intelligence, machine learning, and educational technologies. She is particularly focused on leveraging intelligent systems to enhance learning experiences and optimize educational outcomes. Her work explores the intersection of data-driven algorithms and pedagogical design, with an emphasis on developing adaptive learning environments and intelligent tutoring systems. Moomna remains actively engaged in advancing the use of AI to address challenges in digital education and personalized learning. (Email: moomnanazir@gmail.com)



Yasir Arif received the B.S. (Hons.) degree in Computer Science from the Global Institute, Lahore, Pakistan. He has since developed a strong research profile with interests primarily focused on machine learning, artificial intelligence, and natural language processing. His work involves the application of intelligent algorithms to solve real-world problems, particularly in areas involving data-driven decision-making and human–computer interaction. He is also keenly interested in the integration of deep learning techniques with NLP tasks such as sentiment analysis, text classification, and language modeling. Yasir continues to explore emerging trends in AI to contribute to both academic and applied research. (Email: yasirarif268@gmail.com)