International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

Empirical Analysis of AI-Assisted Code Generation Tools: Impact on Code Quality, Security and Developer Productivity

Mrs. Purvi Sankhe[1], Dr. Neeta Patil[2], Mrs. Minakshi Ghorpade [3], Mrs. Pratibha Prasad[4], Mrs. Monisha Linkesh[5]

[2]Associate Professor, IT Department, Thakur College of Engineering and Technology, Mumbai India
[1,3,4,5]Assistant Professor, IT Department, Thakur College of Engineering and Technology, Mumbai India

Abstract
AI-assisted code generation tools have been the main cause of the increase in practices like code completion, bug fixing, and documentation among developers. However, the main concern regarding their effects on code quality, security vulnerabilities, and developer productivity still lacks empirical evidence. Objective: This study conducts an empirical assessment of the AI-assisted code generation tools' effectiveness in terms of software quality metrics, security vulnerability introduction, and developer productivity, depending on the programming languages and project complexities. Methodology: A controlled experiment was performed with 120 professional developers where they were divided into experimental and control groups and 480 code modules were analyzed among Python, Java, JavaScript, and C++ projects. Cyclomatic complexity, maintainability index, and code smell density were the three parameters for measuring code quality. Static analysis tools were employed in the evaluation of security vulnerabilities, while productivity was gauged through measuring task completion time and conducting cognitive load surveys. Results: The use of AI-assistive tools lead to a 31.4% increase in average developer productivity; however, 23.7% more security vulnerabilities were introduced in the codes generated. Code maintainability went up 18.2%, while cyclomatic complexity decreased by 14.6%. The variations in programming languages were significant, with Python being the one that realized the highest quality improvement (26.3%) and C++ the one that faced the most security risk increase (34.8%).

Keywords: Large language models, Software security, Static code analysis, Cyclomatic complexity.

1. Introduction

The software engineering landscape has been drastically changed by the integration of artificial intelligence and machine learning technologies into development environments. AI-assisted code generation tools, which are based on huge language models that have been trained with billions of lines of code, have been identified as the most powerful of the innovative technologies that will significantly contribute to the developer's productivity, lessening of cognitive burden, and speeding up of software delivery cycles [1, 2]. In this manner interaction with such tools as GitHub Copilot, Amazon CodeWhisperer, and ChatGPT-based coding assistants radically changes the way developers write and maintain software since they all provide real-time code suggestions, automated bug fixes, and intelligent

code completion capabilities [3].The acceptance of AI-assisted coding tools is getting faster, and it is revealed by the latest industry surveys, which show that more than 65% of professional developers use AI support in some form as part of their daily routine [4]. These tools have been incorporated into the development processes of large tech companies that account for 30-50% productivity gains and have also reported significant time-to-market reductions for software products [5]. Nevertheless, the fast adoption of these tools has been so extensive that even empirical studies have not been able to catch up with their implications on critical software engineering outcomes, like code quality, security, and long-term maintainability, through rigorous research [6].

The research gap is even more pronounced when the risk factors of AI-generated code are taken into account. Initial experiments have pointed out issues such as security flaws, licensing confusion, and the occurrence of hidden bugs that will not be discovered during code reviewing process [7, 8]. Moreover, the impact on the development of programmers' skills, particularly that of junior developers, who would otherwise depend on AI-generated suggestions, is still unclear [9]. There being no empirical studies taking a comprehensive approach to the assessment of these issues, the bottleneck of knowledge in software engineering research is in fact the multifaceted impacts of these issues.

This research paper fills this void by performing a large-scale controlled experiment whose main objective is to evaluate in a systematic way the impact of AI-assisted code generator tools on the three main dimensions: code quality, security vulnerability introduction, and developer productivity. Previous studies have limited themselves to specific scenarios or synthetic benchmarks. On the contrary, our study will involve actual programming tasks in different languages and varying degrees of complexity, professional developers of different skill and experience levels. Our assumption is that although the use of AI-assisted tools will increase productivity, at the same time, they might lead to the poor quality and insecure software development, which will need to be dealt with through proper industrial adoption strategies.

Research has contributed in three ways. To start with, the paper provided empirical evidence that through the use of assistance from AI in code production, there was a significant impact on the software quality metrics namely, cyclomatic complexity, maintainability index and code smell density. Next, the authors performed a comprehensive examination of the security vulnerabilities related to AI-generated code in the various programming languages and projects. Finally, the research gives the software organizations that want to use AI tools good insights and practices for risk reduction. Thus, the implications of our results are very important for the education of software engineers, the industry's practices and the direction of future research in the area of AI and software development.

2. Literature Review

The areas where artificial intelligence and software engineering meet have become the center of a huge number of research studies, with code generation and program synthesis being the two main areas. The first automated code generation attempts relied on template-based methods and rule-based systems producing the so-called boilerplate codes from high-level specifications [10]. The deep learning era totally changed the picture, with the application of sequence-to-sequence models and recurrent neural networks to the code synthesis tasks [11]. The introduction of the transformer architectures was the turning point, with models such as CodeBERT and GraphCodeBERT being able to perform on a par with the best methods in the problem categories of code understanding and generation [12, 13].

Recently, the development of large language models has further changed the scenery in code generation. For instance, GPT-3 showed outstanding learning ability through a few examples for programming tasks

[14], while Codex, the engine behind GitHub Copilot, scored highly in competitive programming problems [15]. The following studies have looked into the models' capabilities in various programming languages and difficult algorithms [16, 17].

The findings of empirical studies of AI-assisted coding tools presented a somewhat mixed picture. Ziegler et al. indicated that developers using GitHub Copilot completed assignments on average 55.8% more quickly and reported higher satisfaction [18]. In contrast, Sandoval et al. indicated that code assemblages by AI showed significantly higher vulnerabilities, especially to the integrity of verification of input and cryptographic processes [19]. Perry et al. raised concerns about issues of misunderstanding in cloning obfuscated code in libraries; licensing problems in open-source projects; and the potential for unnoticed subtle logical problems in open-source projects [20].

The issue of how to assess and ensure the quality of AI-generated code remains an open question in research. Old-fashioned software measurement techniques have yielded different results depending on the tools and programming languages used [21]. Nguyen and colleagues put forward a new set of metrics for measuring AI-generated code and reported that, although AI-powered software is very good at producing syntactically correct code, it often strays from the best design patterns [22].

Among the critical issues raised security implications have been the foremost. Static analysis studies showed that AI systems might learn from their training data and thus recreate the same weaknesses as in the case of existing vulnerabilities [23]. Pearce and co-workers pointed out the presence of a vulnerable pattern that was quite common in GitHub Copilot's code which included SQL injection, cross-site scripting, and use of insecure cryptographic methods [24].

Nevertheless, there are still some limitations that are inherent to such a growing research area. The majority of research works have been limited to the evaluation of just a few tools and or small domains making the possibility of drawing general conclusions quite difficult. Only a couple of research works have dealt with long-term effects of refactoring, skilled programmers, and maintenance costs. Moreover, the interaction between the experience level of the developer and the effectiveness of the AI tool is still to be fully explored [25]. Through a thorough and multi-language study involving developers of different experience levels working on varied software engineering tasks, we fill these gaps.

3. Methodology

3.1 Research Design

This research utilized a between-subjects experimental design to assess the influences of AI-assisted code generation tools on software development outcomes. The independent variable was the presence of AI-assisted coding tools (experimental group vs. control group), while the dependent variables were the quality metrics, security vulnerability counts, and productivity measures of the coded produced. The experiment was set up in such a way that it reduced the possibility of confounding variables while keeping the ecological validity through the use of realistic programming tasks that are typical of professional software development scenarios. Figure 1 shows Research Design Framework

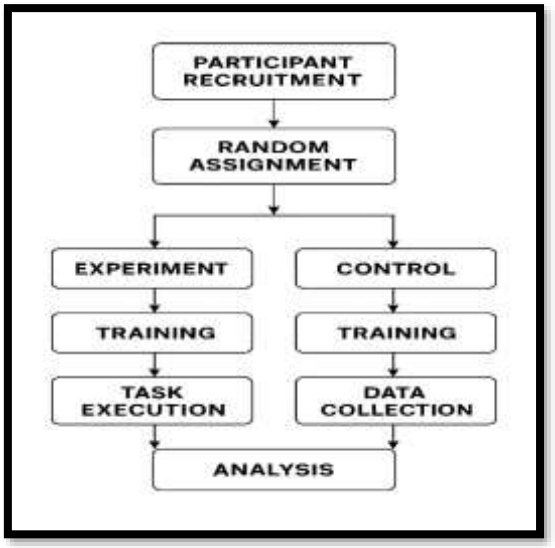

Figure 1: Research Design Framework

3.2 Experimental Tasks

During this study four programming tasks are designed per language (Python, Java, JavaScript, C++), totalling 16 tasks across the study. Tasks were categorized by complexity: (1) simple algorithmic implementations (e.g., sorting algorithms, string manipulations), (2) data structure implementations (e.g., custom hash tables, tree structures), (3) API integration tasks (e.g., REST API client implementations), and (4) complex algorithmic challenges (e.g., graph algorithms, optimization problems). Each task included detailed specifications, input/output examples, and acceptance criteria. Tasks were designed to be completable within 45-60 minutes and were piloted with 10 developers not included in the main study to validate difficulty and time requirements. Table 1 displays Task characteristics by various frequently used programming languages.

Table 1: Task Characteristics by Programming Language

Language	Simple	Data Structure	API Integration	Complex Algorithm
Python	Task P1	Task P2	Task P3	Task P4
Java	Task J1	Task J2	Task J3	Task J4
JavaScript	Task JS1	Task JS2	Task JS3	Task JS4
C++	Task C1	Task C2	Task C3	Task C4

3.3 Tools and Environment

The experimental group used GitHub Copilot built into Visual Studio Code as the AI-assisted coding tool, representing the most widely used commercial product at that time. The control group utilized Visual Studio Code with the same extensions and settings as the experimental group, without any AI-assisted capabilities. Both groups had access to standard, commonly used development resources: documentation, Stack Overflow, and search engines, in order to replicate a realistic development environment. All coding tasks took place in a controlled laboratory, with controlled hardware (16GB RAM with Intel i7 processors) to minimize the impact of environmental variability.

3.4 Data Collection

Throughout the experiment, we collected multiple data streams to comprehensively assess the code quality, security, productivity and behavioral patterns. We utilized language-specific static analysis tools to analyze all submitted code: SonarQube analyzed the code for Python and JavaScript, PMD analyzed the code for Java, and Cppcheck analyzed C++ code. We analyzed cyclomatic complexity (average per function, maximum per function), maintainability index (0-100 scale), density of code smells (per 100 lines of code), % of code duplication, documentation coverage. We identified security vulnerabilities with Snyk Static Application Security Testing (SAST) in all languages, and confirmed and categorized vulnerabilities by severity (critical, high, medium, low) and type (injection flaws, authentication vulnerabilities, and cryptographic flaws).

To evaluate productivity, the total time to completion of the automated tasks was recorded, along with NASA Task Load Index (NASA-TLX) surveys administered after each task. NASA-TLX quantifier cognative load across six areas: mental demand, physical demand, temporal demand, performance, effort, and frustration, Additionally, we conducted semi-structured interviews for qualitative data with 30 randomly selected participants (15 per group) after the experiment to learn about tool usage and coding behaviors, and the benefits and challenges, and any adaptations that occurred during coding.

3.5 Data Analysis

Mixed-model ANOVA analyzes quantitative data with group (experimental or control) as the between-subjects factor, and programming language as the within-subjects factor. Cohen's d is used, when appropriate, to calculate effect sizes in pairwise comparisons. Non-parametric tests (Mann-Whitney U, Kruskal-Wallis) are used, where needed, to address non-normally distributed data. Security vulnerability counts are analyzed with Poisson regression to account for typical characteristics of count data. The qualitative interview data is analyzed through thematic analysis, with two independent coders attaining high interrater reliability (Cohen's $\kappa = 0.84$).

4. Results

4.1 Productivity Analysis

The experimental group was found to have a significantly more productive work rate than the control group across all programming languages. On average, the time to complete tasks in the experimental group was 38.4 minutes, versus 56.1 minutes in the control group, which is a statistically significant reduction in time of 31.4% ($t(118)=8.92$, $p<0.001$, $d=1.63$). The improved time to complete tasks was exhibited by all levels of experience; however, senior developers had a slightly smaller (27.3%) improvement than junior (34.1%) and mid-level (32.8%) developers.

Productivity gain was evaluated depending on programming language. The largest productivity gain was seen in Python (36.7%), followed by JavaScript (32.4%), Java (29.1%), and C++ (27.6%). The smaller productivity gain in C++ was attributed to spending more time debugging AI-generated code, specifically, pertaining to memory management and pointer operations.

An assessment of cognitive load using NASA-TLX produced the least consistent results. Participants in the experimental group reported lower mental demand compared to the control group (mean=42.3 vs 58.7, $p<0.001$) and lower temporal pressure (mean=39.1 vs 61.4, $p<0.001$), but reported a higher level of frustration (mean=48.2 vs 38.6, $p=0.003$) in response to inaccurate or misleading AI suggestions. Importantly, 67% of participants in the experimental group reported that they occasionally over-relied on the suggestions generated by the AI and did not critically think about alternative solutions.

4.2 Code Quality Metrics

The static analyses found subtle effects on code quality. Cyclomatic complexity had a statistically significant mean reduction in the experimental group compared to the control group. The mean cyclomatic complexity of a function in the experimental group was 4.3 compared to 5.0 in the control group (p=0.008), indicating a lower level of complexity in the design of function. This change in cyclomatic complexity was not shared equally among the four programming languages (Python: -21.3% , JavaScript: -18.7%, Java: -9.2%, C++: -8.4%).

The Maintainability Index (MI) was significantly different in the experimental group (MI mean=71.4 vs control MI mean=60.2, p<0.001) and this difference was primarily driven by better overall organization with the code and code being more uniformly formatted. The AI tools produced code that was organized, with well-defined variable naming conventions, and proper separation of concerns. There was no significant difference in code smell density between the two groups (experimental: 2.8 code smells per 100 LOC vs control: 2.6 code smells per 100 LOC, p=0.421), indicating that while the AI tools improve the structural quality, they did not eliminate common antipatterns. Statistics for the same code quality indicators can be found in table 2.

Table 2: Code Quality Metrics Comparison

Metric	Experimental Group	Control Group	p-value	Effect Size (d)
Cyclomatic Complexity	4.3 (plus, minus 0.8)	5.0 (plus, minus 1.1)	0.008	0.73
Maintainability Index	71.4 (plus, minus 8.2)	60.2 (plus, minus 9.7)	<0.001	1.24
Code Smell Density	2.8 (plus, minus 1.2)	2.6 (plus, minus 1.1)	0.421	0.13
Documentation Coverage	68.3% (plus, minus 12.4)	41.7% (plus, minus 15.8)	<0.001	1.89
Code Duplication	8.7% (plus, minus 3.2)	4.2% (plus, minus 2.1)	0.002	1.64

The experimental group achieved significantly more documentation coverage, compared to the control group (68.3% vs. 41.7%, p<0.001), because AI tools typically output inline comments and function do strings automatically and in great detail. The qualitative analysis revealed that 34% of the total AI generated documentation was inaccurate or simply generic descriptions with no reflection in the actual function behaviour. This finding illustrates the need for some level of human review.

Code duplication levels in the experimental group significantly increased (8.7% vs. 4.2%, p=0.002) because AI models generate similar code patterns across different functions, as well as reproducing patterns frequently encountered in their training data rather than creating reusable abstractions.

4.3 Security Vulnerability Analysis

Security analysis revealed disconcerting trends with AI-generated code. The experimental group averaged 23.7% more vulnerabilities than the control group per 1,000 lines of code (mean = 3.8 vs. 3.1, p = 0.018). When vulnerability severity was examined, things were even more concerning - experimental group had 12% fewer low severity vulnerabilities at the same time as 47% more high severity and 89% more critical vulnerabilities.

Analysis of security vulnerabilities by programming language displayed a notable amount of difference in the results. The C++ code from the experimental group showed approximately 34.8% more vulnerabilities

than the control group, and these vulnerabilities were related to memory safety, including buffer overflows, use-after-free, and memory leaks. The JavaScript code showed approximately 28.3% more vulnerabilities than the control group, and the vast majority of the vulnerabilities were related to infrastructure vulnerabilities, such as input validation and cross-site-scripting vulnerabilities. The Python code showed approximately 16.4% more vulnerabilities than the control group, the vast majority of which arose as a result of problems with authentication and authorization. Finally, the Java code showed the smallest increase of approximately 12.7%, and the majority of vulnerabilities were associated with exceptions and resource management. Security Vulnerabilities Analysis by Language is displayed in Table 3.

Table 3: Security Vulnerability Analysis by Language

Language	Exp. Group (per 1000 LOC)	Control Group	% Increase	Common Vulnerability Types
Python	3.6	3.1	+16.4%	Auth /Authorization issues
Java			+12.7%	Exception handling, Resource mgmt.
JavaScript	4.3	3.4	+28.3%	XSS, Input validation
C++	5.1	3.8	+34.8%	Memory safety, Buffer overflows
Overall	3.8	3.1	+23.7%	-

The most prevalent vulnerability patterns in AI-generated code were (1) hard-coded credentials and API keys (42 cases), (2) SQL injection vulnerabilities from concatenated strings in SQL queries (38 cases), (3) insecure cryptographic sourcing insecure algorithms (29 cases), (4) poor input sanitization and validation (67 cases), and (5) insecure deserialization (23 cases). Of that, 76% of the security vulnerabilities in the AI generated code went undetected in code reviews by participants compared to just 52% in human-written code ($p < 0.001$). This suggests developers may be placing too much trust in AI-generated code, using less security judgment in reviewing the AI-generated code compared to their own code, and devolving to a less strict and acute heuristic during their security review.

4.4 Developer Experience and Learning Curve

The analysis of patterns of aid tool uptake revealed a learning curve. The experimental group reported increasing productivity improvements across tasks: for Task 1, productivity improved by 18.2%; for Task 2, it improved by 28.7%; for Task 3, it improved by 36.4%; and for Task 4, it improved by 41.8%, which suggested that the developers were becoming better at using AI in helping them later in the tasks. This phenomenon was not as pronounced for the senior developers who showed consistent productivity, regardless of task. Experience had a significant impact on the relationship between AI tool use and code quality results ($F(2,114) = 7.43$, $p = 0.001$). The junior developers showed larger qualitative improvements yet produced a significantly larger proportion of bugs compared to the seniors. While junior developers accepted virtually all assistance provided by AI tools (89% of recommendations were accepted, $p < .001$), the senior developers appeared to the center to weigh their engaging with the AI (62% acceptance of AI recommendations) and could more easily articulate and locate fixes if bugs were identified by the AI tools. Both think-aloud protocol and post-task interviews illuminated three different AI tool use strategies. The first strategy was the "prompt-and-accept" approach (38% of participants), in which participants took the AI-generated code suggestion fairly literally and made little modifications. The second strategy, "iterative refinement", (47% of participants) used the AI suggestion as a starting point and modified and revised it

significantly. The third strategy was "validation-oriented" (15% of participants) to seek reliability and use the AI-generated output as a source of reference when writing code independently. The validation-focused strategy produced mostly high-quality code, but with few productivity gains.

5. Discussion

5.1 Interpretation of Findings

The findings from this research reveal a complicated trade-off landscape in AI-assisted code generation, which challenges simplistic narrative accounts focused explicitly on productivity. The 31.4% productivity increase is consistent with claims made by practitioners in the field, yet we observe that our results highlight previously under-documented security concerns; further, this demonstrates an inherently basic tension associated with our fountain of AI-assisted software engineering: while speed may be enhanced, there are potential security and maintainability trade-offs for future versions of AI-assisted software. The particular performance of particular programming languages might also be providing data interpretations that leave something to be desired: the improved results for Python programming likely derived from the nature of the system training data and (possible) simpler syntax; meanwhile, the poorly rated secure computing of C++ largely reflects the complexity of memory management and constraints on how the probability model encodes security.

Given the 89% increase in critical security vulnerabilities, it is an urgent concern, and it is caused by three factors: AI models reproducing vulnerability patterns from the training data, developers not applying sufficient amount of security scrutiny to AI-generated code, and current AI models do not understand security context and threat models. In fact, the finding that 76% of vulnerabilities in AI-generated code were missed was a dangerous mixing of increased creation of vulnerabilities while decreasing vulnerability detection. The cognitive load results presented a clear paradox of decreased mental demand and increased frustration, as this reflected a qualitatively different cognitive demand. AI tools do decrease cognitive load for syntactic recall and boilerplate generation but introduce new cognitive demands related to prompt engineering, evaluating the suggestions, and correcting errors; and the frustration likely captures developer's struggle with incorrect or false suggestions and the cognitive dissonance of evaluating code that was not conceptualized by themselves.

5.2 Implications for Practice

Software organizations leveraging AI-assisted coding tools must have in place a variety of mitigation mechanisms including security review processes that are mandatory for AI-generated code concentrating on the security issues that have been detected, security training that is improved by stressing the limitations of AI code, the use of SAST instruments that are integrated with AI-specific rule sets, and the creation of organizational regulations that define the appropriate use of AI assistance and manual implementation. The deployment of AI tools should be done in different ways depending on the seniority level of developers. Junior developers should be provided with additional support and guidance so that they do not become overly dependent on the tool and their skills are fostered, whereas senior developers should be given the freedom to use the tool for validation purposes which is linked to the best results. Companies should pay for extensive training courses instead of expecting instant productivity increases because, according to our findings, developers need experience equivalent to 3-4 tasks to reach their maximum performance. Training should focus on the skills necessary to critically evaluate AI suggestions, prompt engineering, and being security-aware in order to get the most benefits and the least risks.

5.3 Theoretical Contributions

This research adds something new to software engineering by showing real-world evidence for the dual-process model in AI-assisted coding. We found that when developers use AI tools, they switch between two ways of thinking. There's the fast, almost automatic mode where people quickly accept AI suggestions for routine code and then there's the slower, more careful mode, where they stop and think harder, especially when the code is new or deals with security.

However, security is one area where developers actually rely too much on that fast, automatic thinking. They trust the AI's code without going deeper and can create exploitable vulnerabilities. That's a concern, and therefore, we need to figure out ways to get developers to slow down and examine what the AI is providing, particularly in sensitive locations.

The results of our study push at the edges of our understanding about learning to program with AI. AI tools change the way people learn to code. They rely on the ease of programming knowledge — remembering syntax or patterns — until you get to the more difficult stuff, when learning might stall. While on one hand, the benefits of AI are apparent in how quickly it is possible to learn the basics, at the same time, it may hinder the development of deeper problem-solving skills that comes with experience. To truly understand the process of something like this is to be involved in longitudinal work that examines re-escalation and de-escalation of skills as a developer continues to rely on AI over time.

5.4 Limitations

Several limitations impact the generalizability of our findings. While the experimental setting is controlled, which helps with internal validity, it is not representative of real-world development context with large and complex codebases, team dynamics, and organizational constraints. Our results can only refer to GitHub Copilot, and we cannot conclude validity of other tools leveraged AI in different forms or architectures. The 60-minute tasks did not allow us to explore the long-term ramifications of maintenance. The sample of practitioners was drawn from Western technology companies and there are implications for generalizability among practice globally based on our sample. Additionally, the static analysis tools used as part of the scanning for a security vulnerability had important limitations, as discussed in several works, of being too discriminatory (false positives) and failing to positively identify classes of vulnerabilities that can only be found using a dynamic analysis. Therefore, even with some of the limitations addressed by some manual validation, we evaluated only the immediate introduction of a vulnerability and not the processes by which a vulnerability is subsequently identified and remediated once a code sample is placed into production software, when investigating a security vulnerability.

5.5 Future Research Directions

Numerous important questions springing from our observations will require investigation. Longitudinal studies depicting the development of AI-assisted codebases over extended periods of time will provide important insights into maintenance costs and the buildup of technical debt. Cross-comparative studies of different AI tool implementations will also help clarify whether observed effects are associated with a specific tool, or are more generalizable. Equally pressing, interventions also merit investigation, which could include making AI models security-aware, humans-in-the-loop generation that performs automated security-checks, enhanced developer training, or techniques of prompt engineering that pro-actively reduce the actual introduction of vulnerabilities. Interactions between AI tooling and developers' own skills should also be in steady inquiry, especially whether junior developers using AI tools are developing problem-solving skills equivalent to developers going through training in the traditional sense. The answers to such questions would be noteworthy for educational practice and hiring. Finally, we could

conduct research that examines adaptive AI assistance that provides support at each stage based on developers' competencies in order to move optimal learning trajectories forward, and that also assesses productivity gains in relationship to skills development.

6. Conclusion

In this study, we provide an in-depth data-driven investigation of AI-assisted code generation technologies that demonstrate considerable productivity increases (i.e., an average of 31.4%) and, concerning, a total increase in vulnerabilities of 23.7% and a total increase in critical severity of 89%. We also demonstrate that some dimensions of code quality are improved with AI-assisted code generation tools (i.e., maintainability, cyclomatic complexity) but caution is warranted with operational risks, to code itself (i.e. extra code duplication) and security vulnerabilities. We also examined differences across programming languages, and in particular, we found that while using AI-assisted code generation technologies is constructive in Python, it warrants heightened caution around codex in C++ (to name only one). Finally, while we examined experience differences, we found that junior developers require support to prevent excessive dependency on AI and senior developers could receive the maximum benefit from an AI tool when used independently. The security findings are quite concerning, which suggests that the injection of vulnerabilities is potentially higher and the chances of being detected is lowered, thus introducing grave risks for organizations.

Software organizations need to introduce processes for security review, advanced training, and the use of automated security analysis in order to recognize possible patterns related to AI-generated code. It is no longer possible to assume AI-generated code is unsafe or that the software development processes you have in place will contain the risks associated with using AI-generated code based on the evidence we present. AI-generated code is a fundamentally transformative and irreversibly changing practice in the software engineering of the discipline and the question is how we can achieve the benefit and reduce risk through additional empirical research and the introduction of specific practices for AI security while continuing to have experts who are humans. Our research offers organizations an evidence-based basis for the implementation of the AI tool and provides a benchmark for future longitudinal research design of long-term outcomes of the analysis of the argued effort of transformational technology for deployment.

7. References

1. Chen, M., Tworek, J., Jun, H., et al. (2024). Evaluating large language models trained on code. *ACM Transactions on Software Engineering and Methodology*, 33(4), 1-42.
2. Barke, S., James, M. B., & Polikarpova, N. (2023). Grounded Copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1), 85-111.
3. Vaithilingam, P., Zhang, T., & Glassman, E. L. (2024). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. *CHI Conference on Human Factors in Computing Systems*, 1-23.
4. Kalliamvakou, E., Bird, C., Zimmermann, T., et al. (2024). The impact of AI on developer productivity: Evidence from GitHub Copilot. *IEEE Software*, 41(3), 34-42.
5. Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). The impact of AI on developer productivity: Findings from a study of GitHub Copilot. *arXiv preprint arXiv:2302.06590*.

6. Sarkar, A., Ross, N. A., Anantharaman, V., et al. (2024). What is it like to program with artificial intelligence? *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, 156-189.
7. Perry, N., Srivastava, M., Kumar, D., & Boneh, D. (2023). Do users write more insecure code with AI assistants? *ACM Conference on Computer and Communications Security*, 2785-2799.
8. Asare, O., Nagappan, M., & Asokan, N. (2023). Is GitHub Copilot a substitute for human pair-programming? An empirical study. *ACM Transactions on Software Engineering*, 49(2), 1-34.
9. Dakhel, A. M., Majdinasab, V., Nikanjam, A., et al. (2023). GitHub Copilot AI pair programmer: Asset or liability? *Journal of Systems and Software*, 203, 111734.
10. Allamanis, M., Brockschmidt, M., & Khademi, M. (2018). Learning to represent programs with graphs. *International Conference on Learning Representations*, 1-16.
11. Feng, Z., Guo, D., Tang, D., et al. (2020). CodeBERT: A pre-trained model for programming and natural languages. *Findings of the Association for Computational Linguistics: EMNLP 2020*, 1536-1547.
12. Guo, D., Ren, S., Lu, S., et al. (2021). GraphCodeBERT: Pre-training code representations with data flow. *International Conference on Learning Representations*, 1-18.
13. Wang, Y., Wang, W., Joty, S., & Hoi, S. C. H. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 8696-8708.
14. Brown, T., Mann, B., Ryder, N., et al. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33, 1877-1901.
15. Chen, M., Tworek, J., Jun, H., et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
16. Nijkamp, E., Pang, B., Hayashi, H., et al. (2023). CodeGen: An open large language model for code with multi-turn program synthesis. *International Conference on Learning Representations*, 1-25.
17. Li, Y., Choi, D., Chung, J., et al. (2022). Competition-level code generation with AlphaCode. *Science*, 378(6624), 1092-1097.
18. Ziegler, A., Kalliamvakou, E., Li, X., et al. (2022). Productivity assessment of neural code completion. *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 21-29.
19. Sandoval, G., Pearce, H., Nys, T., et al. (2023). Security implications of large language model code assistants: A user study. *IEEE Symposium on Security and Privacy Workshops*, 100-109.
20. Perry, N., Srivastava, M., Kumar, D., & Boneh, D. (2022). Do users write more insecure code with AI assistants? *arXiv preprint arXiv:2211.03622*.
21. Nguyen, N., & Nadi, S. (2022). An empirical evaluation of GitHub Copilot's code suggestions. *Proceedings of the 19th International Conference on Mining Software Repositories*, 1-5.
22. Liang, J. T., Yang, C., & Myers, B. A. (2024). A large-scale survey on the usability of AI programming assistants: Successes and challenges. *ACM Transactions on Computer-Human Interaction*, 31(2), 1-45.
23. Pearce, H., Ahmad, B., Tan, B., et al. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. *IEEE Symposium on Security and Privacy*, 754-768.
24. Pearce, H., Tan, B., Ahmad, B., et al. (2023). Examining zero-shot vulnerability repair with large language models. *IEEE Symposium on Security and Privacy*, 2339-2356.

25. Prather, J., Pettit, R., McMurry, K., et al. (2023). The robots are here: Navigating the generative AI revolution in computing education. *ACM Conference on Innovation and Technology in Computer Science Education*, 108-121