

GENERATIVE ARTIFICIAL INTELLIGENCE FOR AUTOMATED CODE GENERATION, SOFTWARE MAINTENANCE, AND PROGRAMMING EFFICIENCY ENHANCEMENT

Nadeem Shoukat^{*1}, Rafia Ahmed²

^{*1}COMSATS, University Islamabad, Sahiwal Campus

²Lecturer, Department of Information Sciences, University of Education, Lahore

^{*1}shoukat659@gmail.com, ²rafiabaig9@gmail.com

DOI: <https://doi.org/10.5281/zenodo.22653575>

Keywords

generative artificial intelligence, automated code generation, software maintenance, programming productivity, large language models, AI-assisted programming, software engineering, code quality, industrial survey

Article History

Received: 14 July 2026

Accepted: 18 August 2026

Published: 31 August 2026

Copyright @Author

Corresponding Author: *

Nadeem Shoukat

Abstract

Generative Artificial Intelligence (GenAI) is rapidly changing software engineering by enabling developers to generate source code, explain programming concepts, identify defects, refactor legacy systems, produce documentation, and automate repetitive maintenance activities. Although experimental studies increasingly report productivity gains from AI-assisted programming, questions remain concerning code reliability, security, verification effort, maintainability, and the extent to which productivity improvements observed in controlled environments transfer to industrial software development. This study examines the perceived and operational impact of GenAI on automated code generation, software maintenance, and programming efficiency through an industrial survey framework. A simulated survey dataset representing 300 software professionals working across software houses, fintech, telecommunications, e-commerce, enterprise software, and health-technology organizations was developed for manuscript demonstration. The proposed instrument measured GenAI adoption, automated code-generation use, maintenance assistance, perceived programming efficiency, code quality, verification burden, security concerns, and organizational governance. Descriptive statistics, correlation analysis, and multiple regression were used to examine relationships among the variables. The simulated findings indicate high adoption of GenAI-assisted development, with code generation, debugging, refactoring, documentation, test generation, and legacy-code explanation emerging as major applications. Programming efficiency showed a positive association with frequency of GenAI use, while perceived code quality and maintenance effectiveness were also positively related to structured human-AI collaboration. However, security concerns, inaccurate outputs, excessive verification requirements, and inadequate organizational policies were identified as important barriers. The findings support the view that GenAI functions more effectively as an augmentation technology than as a replacement for professional software engineering judgment. The study concludes that organizations can obtain greater benefits when AI-assisted programming is integrated with code review, automated testing, security scanning, developer training, and governance mechanisms.

Introduction

Traditionally, development of custom software applications has depended on an individual's expertise to articulate requirements and develop code as well as to perform software testing, diagnosis, and correction of defects, to understand and support legacy systems, and to enhance the functionality of the software. Development of modern applications with advanced features and functionalities has increased the complexity of software. Simultaneously, development cycles have become very short, technical breakage/ debt has increased, and there is a shortage of software developers. Because of these factors, the industry is now looking at implementing automation for software development. The introduction of large language models (LLMs) has facilitated this change, as now, software development can be done with the help of LLMs using natural language. Modern GenAI systems can generate code, complete code snippets, understand and explain code, develop unit tests, suggest improvements, translate code, automatically document code, and help developers refactor.

The results of contemporary developer surveys indicate the rapid adoption of AI systems in programming. According to the 2025 Stack Overflow Developer Survey, 51% of professional developers reported using AI-based programming tools regularly, while 84% of participants either used or planned to use these tools. Use of these systems has also generated a lot of concern, as 46% of respondents indicated they do not trust the systems.

These changes have great industrial relevance, as productivity in software development cannot be measured in terms of the number of lines of code developed. Efficient software engineering involves requirement analysis, architecting, testing, security, documentation, deployment, support, and collaboration. Thus, if a tool that rapidly develops code is used, it may not enhance productivity of the organization, as the developers may have to spend considerable time in ensuring the code is appropriate, secure, and is maintained.

Even with the recent lack of data on the subject, there is clearly a lot of potential. A

recent controlled study conducted by the Microsoft research team found that GitHub Copilot aided developers in completing programming tasks 55.8% faster than developers who did not have access to the AI assistant. Additionally, newer field studies conducted at the Microsoft, Accenture, and Fortune 100 organizations, that included data from 4,867 developers, found that developers who were given access to an AI coding assistant showed a 26.08% increase in completed tasks, and that the effects were variable across different developers and organizations. Less-experienced developers showed productivity gains.

However, the productivity gains must be observed alongside potential costs. Developers get syntactically correct, but semantically incorrect and inconsistent responses and outputs that are poorly aligned with the requirements of the project and are often insecure. Thinking about maintenance costs, 2025 Stack Overflow developers' surveys revealed that 66% of developers found that the AI output was "almost correct, but not quite." 45% of developers from the survey's population said that AI output required time to debug and maintain.

From a maintenance perspective, this is the most important aspect. Software maintenance consists of several tasks such as fixing errors, adapting software to the user's needs, optimizing and enhancing the software, improving the software's performance, enhancing the software by replacing outdated elements, and reducing the accumulated technical debt. It can aid developers in reducing the burden and explaining unfamiliar code, even potentially providing recommended changes. However, generating code can also increase the volume of code, making maintenance in the future a longer and more difficult task. Recently, there has been a distinction in the research made between coding productivity and maintenance productivity, especially in the prospect of GenAI.

The organizational environment is also a factor. DORA's 2025 research, based on almost 5,000 technology professionals and more than 100 hours of qualitative analysis,

describes AI as an “Amplifier.” AI can augment superior frameworks of an organization, but also enhance inferior ones. Therefore, organizations that use AI in tandem with new development practices to enhance testing, documentation, reviews, DevOps, and governance may stand to gain more from Generative AI than organizations that introduce AI without the changes.

Accordingly, this research studies Generative AI from the standpoint of industrial software engineering. In contrast to others who study the accuracy of code generation in isolation, this research studies three interrelated outcomes: effective automatic code generation, efficient software maintenance, and efficient programming. This research also studies code quality, verification, security, developer satisfaction, and organization governance.

Problem Statement

Though GenAI programming assistants are being increasingly introduced to professional software development, organizations lack empirical evidence regarding the impact of these technologies on the complete engineering process. There is a notable lack of study on the interrelations of automated code generation, software maintenance, verification, and programming efficiency in industrial settings.

There is also the concern that improved productivity may be illusory. Developers may write code faster, but spending more time reviewing, testing, debugging, and maintaining code. The growing trust gap between high technology adoption and low trust reported in developer surveys is illustrative. The goal of this research is to address the gap by introducing an industrial survey model to analyze the potential positive and negative effects of the use of Generative AI in programming.

Research Gap

This research attempts to address three major gaps:

First, early research tended to examine individual programming tasks rather than the whole SDLC. Second, research on productivity does not incorporate automated code generation with software maintenance and verification. Third, industrial implementations

and usage are increasing quicker than organization governance and measurement systems.

A recent systematic literature review covering the years 2020 to 2025 identified benchmarking of code generation as being dominated by HumanEval, MBPP, and APPS. This study finds that hallucination, security, insufficient generalization, and a lack of interpretability exist. This study also notes that benchmarking for functional correctness does not capture the complexity of software engineering in the real world.

This research contributes to the literature by identifying GenAI programming efficiency as an outcome which consists of multidimensional factors such as speed, effort reduction, maintenance, code quality, and verification.

Research Objectives

The study aims to:

1. Examine the extent to which software professionals use GenAI for automated code generation.
2. Determine the perceived effectiveness of GenAI in software maintenance activities.
3. Assess the relationship between GenAI adoption and programming efficiency.
4. Examine the influence of AI-generated code on perceived code quality and verification workload.
5. Identify major organizational, technical, security, and human barriers to GenAI-assisted software development.

1.4 Research Questions

1. To what extent do software professionals use GenAI for automated code generation?
2. How effectively does GenAI support software maintenance activities?
3. What relationship exists between GenAI adoption and programming efficiency?
4. How does GenAI-assisted development influence perceived code quality and verification effort?
5. What major barriers affect the successful industrial adoption of GenAI for software development?

Hypotheses

H1: GenAI adoption has a significant positive relationship with programming efficiency.

H2: GenAI-assisted code generation has a significant positive relationship with perceived code-generation effectiveness.

H3: GenAI-assisted maintenance has a significant positive relationship with perceived maintenance effectiveness.

H4: Perceived code quality is positively associated with structured human verification of AI-generated code.

H5: Security and reliability concerns negatively influence perceived organizational effectiveness of GenAI-assisted programming.

Literature Review

Modern automated programming uses templates, program synthesis, and generators. LLM-based systems provide a different interaction model. Developers describe programming requirements in natural language, and LLMs provide candidate implementations. So, the gap between problem specification and source code is lower in this case.

With such systems, participants can obtain inline completions, suggestions for functions and database queries, suggestions for test cases, and even explanations of compiler errors, as well as suggestions for changing code.

While flexibility is appreciated, LLM-based systems, as with all systems, have their constraints. Legislation of code is probabilistic, and LLMs can produce code that satisfies all syntactic requirements, but is semantically incorrect. This is why benchmark metrics of LLM-based code generation systems (e.g., pass@1, pass@k) are useful, but only useful to a certain extent. In industrial contexts, such systems should provide candidate code, rather than automated code completion. Developers should understand the requirements, assess the risks of the code in context, and test generated code.

GenAI could be of great assistance in software maintenance. Large codebases may include hidden functions, legacy libraries, inconsistent code styles, and complex interdependencies. Developers of legacy code systems can be forced to parse and recreate design rationales.

GenAI tools can summarize files, point out functions, identify interrelationships between assets, draft documents, propose strategies for refactoring, and construct test cases. All of these tools have the potential to save time for new employees attempting to understand a software system.

This can become especially evident in companies that have a dominant legacy system. A developer could ask their Assistant to explain a module and generate regression tests, and also suggest a course of action for migrating that module. This would not negate the need for maintenance skills, but it would lessen the need to search for information.

Recent industry research has highlighted the importance of this differentiation. In a 2025 survey on the adoption of GenAI in software engineering, it was noted that GenAI was used extensively in software development, verification, validation, maintenance, and personal assistant activities. The results of the study showed increased productivity for a reasonable cost, but the challenges of an unreliable system persisted. (Giray et al., 2025)

Typing code at a higher rate is just one component of programming efficiency. The time to understand requirements, search for solutions, and complete software tasks is also of high importance. Previous controlled studies indicate that the use of AI assistants can bring short term gains to productivity. In the Copilot study conducted by Microsoft, the participants using AI assistants were able to complete a task to build an HTTP-server 55.8% faster than the control group.

More recent experiments conducted in three workplaces provide a more solid basis for industry. Microsoft, Accenture, and a Fortune 100 organization provided evidence that the use of AI by 4,867 developers resulted in a 26.08% increase in tasks completed. Researchers also discovered differences depending on the developer's experience, suggesting that AI productivity impacts differ depending on the developer involved.

This is in line with the main proposition of this paper: GenAI should be treated as a productivity multiplier, but the effectiveness of GenAI will depend on the characteristics of the task, the developer's experience, and the

quality of oversight. Productivity gains are of little value if they result in poor quality software. Code quality comprises correctness, readability, maintainability, and adherence to standards, as well as reliability, security, and testability.

The use of Copilot in the studies conducted by GitHub positively correlated with several aspects of code quality such as functionality, readability, reliability, as well as code maintainability and conciseness. The positive correlations cannot be interpreted as a statement that AI-based coding leads to positive outcome quality. Rather, the quality and outcome of code generated via AI programming relies on the context and task for which it is created, the quality of the prompt, the model, the developer's experience, and the review process.

The results from the 2025 Stack Overflow survey illustrate this dilemma: 33% of participants indicated a level of trust in AI for code accuracy, while 46% expressed a clear lack of trust. As such, the primary element of responsible AI-based programming is verification by a human.

Conceptual Framework

The study proposes a five-component framework:

AI-based coding introduces established software security risks as well as unique risks associated with AI. AI-based code may include a wide range of insecure authentication, cryptographic failures, poor input handling, and insecure data handling, as well as dependencies that need to be properly validated.

The Generative AI Profile from NIST distinguishes systematic risk management throughout the AI Life Cycle, and NIST SP 800-218A implements secure software development practices into generative AI and foundation model creation.

Like NIST, OWASP's 2025 guidance lists risks from prompt injection, sensitive info disclosure, and supply-chain attacks that extend to other LLM threats. OWASP explicitly adds the practice of blind trust in AI-generated code to their attack surface, and advises that code be reviewed and analyzed by the developer before submitting the code.

The adoption of GenAI technologies need to include security code reviews, dependency scans, automated UI testing, access controls, DLP Tools, and most importantly training for the developers.

Developer Expertise + Human Verification + Organizational Governance

The model assumes that GenAI adoption contributes to productivity primarily through

CONCEPTUAL FRAMEWORK: GENAI ADOPTION & ORGANIZATIONAL SOFTWARE OUTCOMES



Three moderating conditions are incorporated:

faster code production, reduced repetitive work, improved access to programming knowledge, and maintenance assistance.

However, the magnitude of these benefits is constrained by verification workload, task complexity, security requirements, and organizational readiness.

Methodology

Research Design

A quantitative, cross-sectional industrial survey design was adopted. The survey model was

Industrial Sample

The simulated sample consisted of 300 software professionals employed in six industrial categories:

Industry	Frequency	Percentage
Software houses / IT services	92	30.7%
FinTech	48	16.0%
Telecommunications	42	14.0%
E-commerce	39	13.0%
Enterprise/ERP software	43	14.3%
Health technology	36	12.0%
Total	300	100%

Participants represented software developers, senior developers, software engineers, DevOps engineers, technical leads, architects, and related technical roles.

Instrument

A five-point Likert scale was used:

1	=	Strongly Disagree
2	=	Disagree
3	=	Neutral
4	=	Agree
5	=	Strongly Agree

The major constructs were:

- GenAI Adoption (GA)
- Automated Code Generation (ACG)
- Software Maintenance Effectiveness (SME)

designed for software professionals working in technology-intensive organizations. The research instrument consisted of demographic questions, GenAI adoption measures, code-generation measures, maintenance measures, programming-efficiency measures, and risk/governance items.

- Programming Efficiency (PE)
- Perceived Code Quality (PCQ)
- Verification Burden (VB)
- Security Concern (SC)
- Organizational Governance (OG)

Data Analysis

The simulated dataset was analyzed through descriptive statistics, Pearson correlation, and multiple regression. Mean scores above 3.50 were interpreted as relatively favorable perceptions.

Results

Participants Profile

Variable	Category	n	%
Experience	<3 years	54	18.0
	3–5 years	72	24.0
	6–10 years	96	32.0
	>10 years	78	26.0
Role	Software developer	126	42.0
	Software engineer	81	27.0
	Senior/lead developer	45	15.0
	DevOps/Cloud	27	9.0
	Architect/technical manager	21	7.0
AI experience	<1 year	57	19.0
	1–2 years	108	36.0

	3–4 years	87	29.0
	>4 years	48	16.0

The simulated profile indicates that the sample contained substantial representation from experienced professionals. Approximately 58% of respondents had more than five years of professional experience.

GenAI Adoption

GenAI Activity	Agree/Strongly Agree (%)	Mean
Code completion	86.0	4.31
Generating functions/classes	82.7	4.18
Debugging assistance	84.0	4.24
Unit-test generation	76.3	3.94
Code explanation	81.7	4.12
Refactoring	71.3	3.82
Documentation generation	79.0	4.01
Legacy-code analysis	68.0	3.71
Code translation	61.7	3.56
Architecture/design generation	48.7	3.22

The highest adoption was observed for code completion, debugging, code generation, and code explanation. Architecture and high-level design showed lower adoption, suggesting that developers remain more cautious when AI is applied to decisions with broad architectural consequences.

This pattern is consistent with the 2025 Stack Overflow survey, which found that developers were particularly resistant to using AI for high-responsibility systemic activities such as deployment, monitoring, and project planning.

Perceived Impact on Programming Efficiency

Statement	Mean	SD
GenAI reduces time spent writing repetitive code	4.24	0.79
GenAI reduces time spent searching for solutions	4.11	0.83
GenAI helps developers remain focused	4.02	0.87
GenAI accelerates debugging	4.06	0.84
GenAI accelerates test creation	3.91	0.91
GenAI reduces documentation time	4.09	0.82
GenAI improves overall programming productivity	4.08	0.81
Overall efficiency score	4.07	0.72

The simulated results suggest a strong positive perception of programming efficiency. The greatest perceived benefit was reduction in repetitive coding work.

This finding is broadly compatible with previous controlled and industrial evidence.

Microsoft's field experiments found increased completed-task productivity among developers receiving AI coding assistance, while earlier Copilot research documented benefits associated with reduced repetitive effort and maintaining developer flow.

Software Maintenance

Maintenance Activity	Mean
Explaining legacy code	3.89
Generating documentation	4.02
Refactoring repetitive code	3.91
Identifying likely bugs	3.87
Generating regression tests	3.83
Updating deprecated syntax/APIs	3.74
Translating legacy code	3.61
Understanding dependencies	3.68
Overall maintenance effectiveness	3.82

The results indicate that GenAI was perceived as particularly useful for documentation, legacy-code explanation, refactoring, and defect identification. These findings suggest that the maintenance value of GenAI extends beyond direct code generation.

Perceived Code Quality

The simulated data showed a mean perceived code-quality score of **3.76** (SD = **0.77**). Respondents reported that AI-generated code

frequently provided useful starting points, but most respondents also indicated that generated code required human review.

Approximately 74% of respondents agreed that human code review remained necessary when AI-generated code was incorporated into production systems. This finding is consistent with contemporary developer sentiment: despite widespread AI use, distrust in output accuracy remains substantial.

Verification Burden and Risks

Risk/Challenge	Mean
Incorrect AI-generated code	4.08
Security vulnerabilities	3.97
Need for extensive code review	4.12
Difficulty understanding AI-generated logic	3.46
Inconsistent coding conventions	3.54
Confidential-data concerns	3.88
Hallucinated libraries/APIs	3.79
Overreliance on AI	3.83
Increased maintenance burden	3.64
Overall risk score	3.81

Verification burden produced one of the highest scores. This suggests that productivity cannot be measured simply by the speed at which code is produced. If AI-generated code requires substantial review and debugging, some apparent productivity gains may be offset. This concern is particularly important in light of recent research suggesting that AI-generated

code can increase downstream rework. A 2025 study of open-source projects reported that, although productivity increased after Copilot adoption, more rework was associated with AI-generated code and experienced developers experienced increased review burdens.

Correlation Analysis

Variable	GA	ACG	SME	PE	PCQ	VB	SC
GenAI Adoption	1	.71**	.65**	.68**	.48**	.39**	-.31**

Automated Code Generation	.71**	1	.61**	.64**	.51**	.43**	-.29**
Maintenance Effectiveness	.65**	.61**	1	.67**	.56**	.34**	-.25**
Programming Efficiency	.68**	.64**	.67**	1	.58**	.29**	-.28**
Code Quality	.48**	.51**	.56**	.58**	1	.21**	-.19**
Verification Burden	.39**	.43**	.34**	.29**	.21**	1	.47**
Security Concern	-.31**	-.29**	-.25**	-.28**	-.19**	.47**	1

Note. $p < .01$.

The results indicate strong positive associations between GenAI adoption and programming efficiency ($r = .68$), automated code generation and programming efficiency ($r = .64$), and maintenance effectiveness and programming efficiency ($r = .67$). Security concerns demonstrated negative relationships with efficiency and GenAI adoption.

Regression Analysis

A multiple regression model was used to estimate programming efficiency from GenAI adoption, automated code-generation effectiveness, maintenance effectiveness, code quality, verification burden, and security concern.

Predictor	β	t	p
GenAI adoption	.29	5.18	<.001
Automated code generation	.21	3.86	<.001
Maintenance effectiveness	.31	5.67	<.001
Code quality	.18	3.41	.001
Verification burden	-.11	-2.21	.028
Security concern	-.14	-2.76	.006

Model: $R^2 = .61$; Adjusted $R^2 = .60$; $F(6,293) = 76.48$, $p < .001$.

The simulated model explains approximately 61% of the variance in programming efficiency. Maintenance effectiveness produced the strongest positive standardized coefficient, followed by GenAI adoption. Verification burden and security concern showed negative effects.

These findings support the argument that GenAI's value is not limited to producing source code. Its contribution to maintenance and knowledge work may be equally important.

Discussion

The results show GenAI has the potential to enhance productivity in coding in industry, specifically within repetitive but well-defined

and data-intensive tasks. The technologies most widely adopted include code completion, function generation, debugging, explaining code, documentation, and test generation. These tasks involve the developer having the ability to check and confirm the AI output to verify it complies with the requirements.

The low adoption of GenAI in architecture and design also has meaning. Architecture is about making decisions that provide trade-offs that span a number of dimensions including, length, time, security, cost, and interoperability. AI models struggle to incorporate some of these trade-offs, so architecture and design will likely still require human input.

These results are consistent with the growing belief that within the workplace GenAI serves best as a tool that complements the human workforce. The developer drives the goal, reviews alternative paths, assesses the output, and accepts ownership of the end output.

Automated Code Generation and Productivity

The results support H1 and H2, as there is a positive association between productivity and automated code generation. This is supported by empirical evidence that AI coding assistants are able to shorten task completion.

However, these results need to be interpreted with caution. The results from the 2025 Stack Overflow survey also provide important context. Although the adoption of AI was at 84%, 46% of developers expressed a lack of trust in AI.

The assumption seems to be that productivity will rely more on optimized AI use rather than simply using AI. The developers more likely to benefit will be those who can convey detailed prompts, add context, evaluate results, perform testing, and improve the AI-generated codes.

Software Maintenance as a Major AI Opportunity

In the simulated model, the strongest regression coefficient was observed for maintenance effectiveness. This indicates that GenAI will have a substantial impact beyond just writing initial code.

Legacy systems often present a bottleneck to an organization's processes as only a small number of senior engineers will have an understanding of critical modules. GenAI has the potential to make this knowledge more available by explaining and summarizing code and dependencies.

We are also not encouraged to take AI explanations as the only or the most complete documentation. An AI model may incorrectly think of a code-logic relationship or think of why a code is not correct. Hence, the maintenance flow combines AI explanations with repository analysis, tests, version control, architectural documentation, and expert analysis.

The Productivity Paradox

Of the many implications, perhaps the most important one is the difference between gross productivity and net productivity. By gross productivity we mean the speed at which code is produced. Net productivity includes the complete lifecycle:

Net productivity = coding gains – verification costs – debugging costs – rework costs – maintenance costs

This is important to note because an AI assistant may produce 100 lines of code, but if these lines introduce defects which take hours to debug, then the organization has not gained in productivity.

The issue is being documented in contemporary literature. It has been studied in open-source software and found that productivity gains can be made, though, in some cases, there can be additional overhead from review and rework, especially in experienced developers.

Because of this, industrial organizations should seriously check how AI tools will perform in more than just how many lines of code are estimated to be replaced, how many times developers report they need to self-report self-service, or how many other developers report they need to serve themselves. They should also consider defects, cycle time, review time, change failure rate, rework, incident frequency, and maintainability.

Human Verification Remains Essential

The simulated findings strongly support the importance of human verification. This is consistent with the contemporary software-engineering principle that AI-generated code should enter the same quality-assurance pipeline as human-written code.

Organizations should require:

- automated unit testing;
- integration testing;
- static analysis;
- dependency vulnerability scanning;
- code review;
- security testing;
- performance testing where appropriate;
- documentation validation; and
- human approval before production deployment.

The need for such safeguards is reinforced by NIST's AI risk-management guidance and secure software-development recommendations.

Organizational Governance

DORA's 2025 research describes GenAI as an organizational amplifier rather than a standalone solution. This finding has direct implications for software organizations.

If a company already has weak documentation, poor testing, inadequate code review, and fragmented development practices, introducing GenAI may increase the volume and speed of software production without addressing underlying quality problems.

Conversely, organizations with strong CI/CD pipelines, comprehensive testing, version control, security processes, and developer training can integrate AI assistance more safely. The most appropriate strategy is therefore not "AI versus developers" but AI embedded within a mature software-engineering system.

Industrial Implications

The findings provide several implications for software organizations.

First, organizations should identify suitable AI tasks.

GenAI should initially be deployed in low-risk, high-volume activities such as code completion, documentation, test generation, debugging assistance, refactoring suggestions, and code explanation.

Second, organizations should measure net productivity.

Management should avoid evaluating GenAI through the amount of code generated. Instead, organizations should measure:

- task completion time;
- deployment frequency;
- defect rates;
- code-review time;
- rework;
- maintenance effort;
- developer satisfaction;
- security incidents; and
- customer-impacting failures.

Third, organizations should establish human-in-the-loop policies.

AI-generated code should be reviewed by qualified developers before integration into production systems. High-risk domains should require additional controls.

Fourth, AI literacy should become part of developer training.

Developers require new skills involving prompt design, context management, output verification, model limitations, security awareness, and AI-assisted debugging.

Fifth, organizations should protect confidential information.

Source code, credentials, customer information, proprietary algorithms, and sensitive business information should not be entered into AI systems without approved organizational controls. Data-security guidance from OWASP emphasizes the growing importance of protecting sensitive information when deploying LLM-based systems.

Limitations

This manuscript's main limitation is the use of simulated instead of real data from industrial respondents for the numerical survey. As a result, the descriptive statistics, correlations, regression coefficients, and percentages must be interpreted only as examples and cannot be used to draw conclusions regarding real-world populations.

The second limitation is the use of a cross-sectional research design. Even with real data, a cross-sectional survey would be able to show only associations and not casual relationships.

The third limitation is the subjective perception of productivity may be different from measured productivity. It is conceivable that developers may report that using an AI assistant improves their productivity, but measurements may indicate that the effect is smaller or different.

The fourth limitation is that GenAI tools and tools that assist programming are rapidly changing. Results based on one tool or programming assistant may not hold for other tools.

Lastly, the results are not expected to hold for the full range of software organizations because of differences among them in regulations, architecture, teams, modes of security, and AI maturity.

Recommendations for Future Research

Future studies should collect longitudinal data from actual software organizations and compare developers before and after AI adoption. Objective engineering metrics should be combined with survey measures.

Future research should examine:

differences between junior and senior developers;

AI effects across programming languages;

AI performance in legacy-system maintenance;

security vulnerabilities in AI-generated code;

long-term technical debt;

code-review workload;

AI effects on developer learning;

agentic coding systems;

organizational AI governance; and

differences between human-only, AI-assisted, and AI-agent-based development.

Randomized workplace experiments are particularly valuable because they can distinguish actual productivity effects from perceptions. The growing body of industrial field evidence demonstrates the feasibility of this approach.

Conclusion

Generative AI, a new branch of artificial intelligence, is now incorporated into many industries, including software engineering. Though some have argued that GenAI can undermine the need for human programmers, GenAI tools are best utilized as complements to the work of developers. Automated code generators will remove the monotonous and tedious work of programming. GenAI can generate code and provide assistance for many programming work-related tasks such as debugging, code documentation and tests, refactoring, and explaining legacy code.

The industrial survey model presented in this study finds that positive relationships exist between GenAI adoption, automated code generation, effective code maintenance, and efficient programming. However, the model

shows that there is a concern for a tradeoff between faster code generation and the effectiveness and long-term sustainability of the organization. Current industry statistics support these findings. Productivity gains due to AI coding assistants have been documented, however, developers remain concerned about the correctness of code, trust, debugging, security, and verification.

From these findings, we are driven to the conclusion that GenAI has to be centrally integrated into the software engineering process. GenAI use should be supported with automated testing, a review of code, security, documentation, governance, and continuous evaluation. The increased productivity due to AI could mean that developers will have less time to manually code. This transformation is likely to lead to the emergence of new and complementary roles within software development. Future developers will increasingly be responsible for defining requirements, overseeing and governing AI-driven systems, ensuring responsible implementation, and assessing the ethical implications of using AI in software programming, among other responsibilities. Statements made above suggest the organizations that stand to benefit the most from AI integrated coding will be those that have the best systems for collaboration with AI and greatest emphasis on security and verification.

REFERENCES

- Autio, C., Schwartz, R., Dunietz, J., Jain, S., Stanley, M., Tabassi, E., Hall, P., & Roberts, K. (2024). Artificial intelligence risk management framework: Generative artificial intelligence profile (NIST AI 600-1). National Institute of Standards and Technology.
- Booth, H., Souppaya, M., Vassilev, A., Ogata, M., Stanley, M., & Scarfone, K. (2024). Secure software development practices for generative AI and dual-use foundation models: An SSDF community profile (NIST SP 800-218A). National Institute of Standards and Technology.

- Butler, J., Suh, J., Haniyur, S., & Hadley, C. (2025). Dear diary: A randomized controlled trial of generative AI coding tools in the workplace. 2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice.
- Cui, Z. K., Demirer, M., Jaffe, S., Musolff, L., Peng, S., & Salz, T. (2025). The effects of generative AI on high-skilled work: Evidence from three field experiments with software developers. Microsoft Research.
- DeBellis, D., Storer, K., Harvey, N., Beane, M., Edwards, R., Fraser, E., Good, B., Kalliamvakou, E., Kim, G., Maxwell, E., D'Angelo, S., Inman, S., Murillo, A., & Villalba, D. (2025). The state of AI-assisted software development. DORA, Google.
- Giray, G., Demirörs, O., Kalinowski, M., & Mendez, D. (2025). An empirical study of generative AI adoption in software engineering. arXiv.
- GitHub. (2022). Research: How GitHub Copilot helps improve developer productivity. GitHub Research.
- GitHub. (2024). Does GitHub Copilot improve code quality? Here's what the data says. GitHub Research.
- National Institute of Standards and Technology. (2024). Artificial intelligence risk management framework: Generative artificial intelligence profile. U.S. Department of Commerce.
- OWASP GenAI Security Project. (2025). LLM and generative AI security solutions landscape. OWASP.
- OWASP GenAI Security Project. (2025). Top 10 risks and mitigations for large language models and generative AI applications. OWASP.
- Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). The impact of AI on developer productivity: Evidence from GitHub Copilot. Microsoft Research.
- Stack Overflow. (2025). 2025 Developer Survey. Stack Overflow.
- Yu, L. (2025). Paradigm shift on coding productivity using GenAI. arXiv.
- Zhang, et al. (2025). LLM-based code generation: A systematic literature review with technical and demographic insights. IEEE Access, 13, 194915–194939.
- Bistarelli, S., Fiore, M., Mercanti, I., & Mongiello, M. (2025). Usage of large language model for code generation tasks: A review. SN Computer Science, 6, Article 673.
- Cui, Z. K., Demirer, M., Jaffe, S., Musolff, L., Peng, S., & Salz, T. (2025). The effects of generative AI on high-skilled work: Evidence from three field experiments with software developers. Microsoft Research. The study combines randomized field experiments involving 4,867 developers and reports a 26.08% increase in completed tasks among AI-assisted developers.
- Giray, G., Demirörs, O., Kalinowski, M., & Mendez, D. (2025). An empirical study of generative AI adoption in software engineering. arXiv. The study reports widespread practitioner adoption, particularly for implementation, verification and validation, personal assistance, and maintenance, while identifying unreliable outputs, validation overhead, security, privacy, and overreliance as continuing challenges.
- Liu, Z., Tang, Y., Luo, X., Zhou, Y., & Zhang, L. (2024). No need to lift a finger anymore? Assessing the quality of code generation by ChatGPT. IEEE Transactions on Software Engineering, 50(6), 1548–1584. <https://doi.org/10.1109/TSE.2024.3392499>
- Tosi, D. (2024). Studying the quality of source code generated by different AI generative engines: An empirical evaluation. Future Internet, 16(6), 188. <https://doi.org/10.3390/fi16060188>

- Umama, Danyaro, K. U., Nasser, M., Zakari, A., & colleagues. (2025). LLM-based code generation: A systematic literature review with technical and demographic insights. *IEEE Access*, 13, 194915–194939.
<https://doi.org/10.1109/ACCESS.2025.3631952>. The review analyzed 58 studies from 2020–2025 and identified hallucination, security vulnerabilities, limited generalizability, and interpretability as persistent challenges.
- Zhang, Q., Fang, C., Xie, Y., Ma, Y., Sun, W., Yang, Y., & Chen, Z. (2025). A systematic literature review on large language models for automated program repair. The review synthesizes research from 2020–2025 on LLM-based automated program repair and its role in software development and maintenance.
- The use of large language models for program repair. (2024). *Information and Software Technology*. This systematic review examined 41 studies concerning LLM applications in program repair, including datasets, architectures, fine-tuning approaches, and evaluation metrics.
- Large language models for software engineering: A systematic literature review. (2025). *ACM Transactions on Software Engineering and Methodology*. This review analyzed 395 studies and identified 85 software-engineering tasks across requirements engineering, software design, development, quality assurance, maintenance, and management.
- Evaluation of generative AI models in Python code generation: A comparative study. (2025). *IEEE Access*, 13, 65334–65347. The study compares contemporary GPT, Gemini, LLaMA, and Claude models using accuracy, reliability, response time, completeness, and cost.
- DORA. (2025). *State of AI-assisted software development*. Google Cloud. The report provides large-scale industry evidence concerning AI adoption, developer workflows, productivity, organizational capabilities, and the role of AI as an organizational amplifier.
- GitHub. (2024). *Does GitHub Copilot improve code quality? Here's what the data says*. GitHub Research. The research examines developer perceptions and code-quality outcomes associated with AI-assisted development.