International Journal of Scientific Research & Engineering Trends
Volume 12, Issue 6, Nov-Dec-2026, ISSN (Online): 2395-566X

Impact of Generative Artificial Intelligence on Modern Software Development: Opportunities, Challenges, and Future Perspectives

Rutvik Narendra Nashikkar[1] Dr. Deepak Singh[2]
Msc Cs,
Suryadatta College Of Science And Commerce, Pune, Maharashtra, India Research Guide:- Dr. Deepak Singh

Abstract— Generative Artificial Intelligence (GenAI) is changing modern software development through tools based on large language models (LLMs). Applications such as ChatGPT and GitHub Copilot can assist developers in generating code, explaining unfamiliar programs, debugging errors, preparing test cases, writing documentation, and exploring implementation alternatives. These capabilities may reduce repetitive work and make programming knowledge more accessible. However, generated outputs can also contain logical errors, insecure patterns, outdated information, or code that does not fit the architecture and requirements of a project. The benefits are therefore not automatic and may vary according to task complexity, developer experience, project context, and the effort required to review AI-generated work. This paper presents a narrative review of research on the opportunities, limitations, and likely future role of GenAI in software engineering. It discusses applications across the software development life cycle (SDLC), summarizes contrasting evidence about productivity, and identifies concerns involving security, privacy, reliability, accountability, and overdependence. The review argues that GenAI is most effective when used as a human-supervised assistant within established engineering practices. Developers remain responsible for requirements, design decisions, validation, security, and final software quality.

Keywords— Generative Artificial Intelligence, Large Language Models (LLMs), Software Development Life Cycle (SDLC), AI-Powered Coding Assistants, Code Generation, Software Testing, Code Quality, Developer Productivity.

I. INTRODUCTION

Software development involves a series of connected activities, including requirement analysis, system design, implementation, testing, deployment, and maintenance. These activities require technical knowledge, problem-solving ability, collaboration, and careful decision-making. In recent years, generative artificial intelligence has introduced a new way to support many of these tasks. Unlike traditional tools that mainly follow predefined rules, GenAI systems can produce new text, code, explanations, and other outputs in response to natural-language instructions.

Large language models are trained on large collections of text and code and can recognize patterns that help them respond to programming-related prompts. Tools such as ChatGPT and GitHub Copilot use these capabilities to provide code suggestions, explain programming concepts, generate examples, and assist with routine development work. Their use has expanded the idea of an AI coding assistant from simple autocomplete to conversational support across several stages of software engineering.

Despite this potential, generated code should not be treated as automatically correct. A response may look convincing while containing a defect, a security weakness, or an assumption that does not match the project. Research has also shown that productivity effects differ across settings. Consequently, the impact of GenAI must be considered in terms of both the work it can accelerate and the additional review, integration, and testing it may require [1], [4], [6].

1. Problem Statement

The increasing availability of GenAI tools creates an important question for software organizations and developers: to what extent can these tools improve software development without reducing reliability, security, and human control? While AI-generated code can help complete routine tasks, developers may spend additional time checking outputs, correcting errors, integrating suggestions, and ensuring that the resulting program satisfies project requirements. There is also a risk that users may accept plausible-looking answers without understanding their limitations.

The central problem addressed in this paper is the gap between the expected benefits of GenAI and the practical challenges of using generated outputs in real software projects. A balanced review is needed to understand where these tools are useful, what risks they introduce, and how human oversight can be maintained.

© 2026 IJSRET

2. Objectives Of The Study

The main objectives of this review are:

- To explain the role of GenAI and LLM-based tools in modern software development.
- To identify applications of GenAI across major SDLC activities.
- To examine potential benefits such as faster task completion, learning support, and reduced repetitive work.
- To discuss limitations related to correctness, security, privacy, maintainability, and overdependence.
- To compare research findings on developer productivity without assuming that one result applies to every context.
- To outline responsible practices and future research directions for human–AI collaboration.

3. Review Of Related Literature

Research on LLMs for software engineering has grown rapidly. Hou et al. conducted a systematic literature review covering 395 research articles published between January 2017 and January 2024. Their review organizes the ways LLMs are applied to software engineering tasks and discusses the opportunities and limitations identified in the literature [2]. This broad review indicates that LLM-based techniques are relevant to more than code completion alone.

Peng et al. evaluated GitHub Copilot through a controlled experiment in which developers were asked to implement a JavaScript HTTP server. Participants with access to Copilot completed the assigned task 55.8% faster than the control group [1]. This is evidence of a productivity benefit in a particular, time-bounded programming task. It should not be interpreted as proof that all developers or all software projects will experience the same improvement.

Song, Agarwal, and Wen studied GitHub Copilot in collaborative open-source software development. Their analysis reported a 6.5% increase in project-level productivity and a 5.5% increase in individual productivity, alongside a 41.6% increase in integration time [6]. These findings illustrate that AI assistance may improve some measures of output while also changing coordination and integration work.

A different result was reported by Becker et al. in a randomized controlled trial involving 16 experienced open-source developers completing 246 tasks in mature projects. In that study setting, allowing early-2025 AI tools increased task completion time by 19% [4]. The result cautions against assuming that AI assistance always saves time, particularly when developers work in complex repositories that require understanding established code and project conventions.

Negri-Ribalta et al. reviewed the security implications of AI-generated code and highlighted that the use of AI does not guarantee secure software. Generated code may contain vulnerabilities, making security review and testing necessary [3]. Nguyen-Duc et al. proposed a research agenda for GenAI in software engineering and identified open questions across 11 areas, including dependability, accuracy, data access, transparency, and sustainability [5].

Taken together, these studies suggest that GenAI can provide meaningful assistance, but its effect depends on the task, developer, project, tool, and evaluation method. The literature supports a balanced approach rather than a universal claim that AI either improves or harms productivity.

II. RESEARCH METHODOLOGY

This paper follows a narrative literature-review approach based on secondary research. It synthesizes selected published research papers and research reports concerning GenAI, LLMs, software engineering, developer productivity, and the security of generated code. The selected sources include peer-reviewed journal publications and clearly identified preprints. The purpose is to summarize major themes and compare findings relevant to the paper's objectives.

The review focuses on five themes: (i) applications across the SDLC, (ii) productivity and developer experience, (iii) code correctness and quality, (iv) security and privacy, and (v) future directions for responsible adoption. The paper does not report a new experiment, survey, interview, or statistical analysis. No primary dataset was collected, and no independent productivity percentage is claimed.

As a narrative review, the paper is not presented as an exhaustive systematic literature review. The conclusions are limited by the selected sources and by the fact that GenAI tools evolve quickly. Reported study results are interpreted within their original experimental or observational contexts.

III. APPLICATIONS OF GENAI IN THE SDLC

1. Code Generation

One of the most visible uses of GenAI is generating code from natural-language instructions or surrounding program context. A developer can describe a function, request a code example, or ask for a common implementation pattern. This may reduce time spent writing repetitive or boilerplate code. However, the generated implementation must be checked

© 2026 IJSRET

against requirements, coding standards, dependencies, and edge cases. **B. Code Explanation and Learning**

GenAI tools can explain unfamiliar code, describe the purpose of functions, summarize a class, or clarify programming concepts. This can help students and developers understand existing codebases and learn unfamiliar languages or frameworks. Explanations should still be compared with the actual source code because a model may misunderstand control flow or invent behavior that is not present.

2. Debugging And Error Resolution
Developers can provide an error message, stack trace, or relevant code and ask for possible causes. GenAI may suggest debugging steps, identify likely mistakes, or propose alternative implementations. The suggestions are hypotheses rather than verified fixes; developers should reproduce the issue, test the proposed correction, and check for side effects.

3. Software Testing
GenAI can help draft unit tests, test data, boundary cases, and test scenarios from a function or requirement description. It may also assist in explaining failing tests. This can improve test preparation, especially for routine cases. Nevertheless, generated tests may simply repeat assumptions in the original code or overlook important failure conditions. Test coverage, assertions, and expected outcomes require human review. **E. Documentation**

AI assistants can produce first drafts of comments, README sections, API descriptions, release notes, and technical summaries. This can reduce documentation effort and make technical information easier to understand. Documentation must be checked for consistency with the actual implementation, since generated descriptions can become inaccurate when code changes.

4. Code Optimization And Maintenance
A developer may ask an AI assistant to compare algorithms, identify repeated logic, suggest refactoring, or explain possible performance bottlenecks. Such suggestions can be useful starting points, but performance claims should be measured with appropriate profiling and tests. A shorter or more elegant-looking implementation is not necessarily faster, safer, or easier to maintain.

5. Requirements And Design Support
GenAI can help transform rough notes into draft user stories, acceptance criteria, design alternatives, and project documentation. It can support brainstorming and communication, but stakeholders and engineers must resolve ambiguity, prioritize requirements, and make architecture decisions. The model does not have independent authority to determine business needs or system constraints.

6. Opportunities And Advantages

- **Improved support for routine work:** Code completion, template generation, and repetitive examples can reduce manual effort for suitable tasks. Developers can redirect some time toward problem analysis, architecture, and review.
- **Faster access to explanations:** Conversational interfaces allow developers to ask follow-up questions about code and concepts. This can lower the barrier to exploring unfamiliar tools and technologies.
- **Support for testing and documentation:** Draft test cases and documentation can provide a useful starting point, helping teams address activities that may otherwise receive limited attention.
- **Idea generation:** AI tools can suggest alternative approaches, examples, and refactoring options. Considering multiple suggestions may help developers explore a problem more broadly.
- **Potential productivity gains:** Controlled and observational studies report benefits in some settings, including faster completion of a specific coding task and increased output in open-source projects [1], [6]. These findings support careful adoption, not blanket expectations.

IV. CHALLENGES AND LIMITATIONS

1. Correctness And Reliability
LLMs generate likely responses rather than guaranteeing program correctness. Code may compile but still produce incorrect results, fail on edge cases, or violate business rules. Developers must review generated code, run tests, and validate behavior against requirements.

2. Security Vulnerabilities
Generated code can include insecure input handling, weak authentication logic, unsafe dependencies, or other vulnerabilities. Security-focused research warns that AI assistance does not itself ensure secure code [3]. Secure coding practices, static analysis, dependency checks, and human security review remain essential.

3. Privacy And Confidentiality
Prompts may contain proprietary source code, credentials, customer information, or internal design details. Sending such material to an external AI service may create confidentiality and compliance risks. Organizations should follow approved tool policies, avoid sharing secrets, and understand the service's data-handling terms.

© 2026 IJSRET

4. Context And Integration Limitations
An AI assistant may not fully understand the architecture, conventions, dependencies, or historical decisions of a large project. Suggested code can therefore be locally plausible but difficult to integrate. Integration work and coordination may offset some of the time saved during code generation [6].

5. Overdependence And Skill Development
Frequent reliance on generated answers without understanding them can weaken debugging, reasoning, and programming skills. This is particularly important for learners. AI should be used to explain and support practice rather than replace independent problem-solving.

6. Bias, Licensing, And Accountability
Training data and model behavior may lead to biased, unsuitable, or contextually inappropriate outputs. Questions can also arise about code provenance, licensing, and responsibility for defects. Teams should apply organizational policies and retain clear human ownership of software decisions.

V. COMPARATIVE DISCUSSION

The selected studies do not produce one uniform estimate of GenAI's effect on productivity. They examine different populations, tasks, tools, and environments. A controlled exercise involving implementation of a JavaScript HTTP server found a 55.8% faster completion time with GitHub Copilot [1]. An analysis of open-source collaboration reported higher project-level and individual productivity, but also increased integration time [6]. In contrast, a randomized trial with experienced developers working on mature repositories found that AI access increased completion time by 19% in that particular setting [4].

These findings are not necessarily contradictory. A small, well-defined coding task may be easier for an AI assistant to accelerate than a task requiring deep knowledge of a mature codebase, complex dependencies, or strict project conventions. The total work also includes understanding suggestions, verifying correctness, integrating changes, and maintaining the resulting code. If evaluation measures only the speed of producing a code snippet, it may miss the effort required to deliver a reliable software change.

The evidence therefore suggests that organizations should evaluate AI tools using realistic tasks and end-to-end measures. Useful measures may include task completion time, defect rates, test quality, review effort, security findings, integration time, maintainability, and developer satisfaction. Productivity should be assessed as the delivery of correct and maintainable software, not simply the volume of generated code.

Table 1. Illustrative comparison of selected productivity studies. Results are reported in their original study contexts and should not be treated as directly interchangeable.

Study	Context	Reported finding	Interpretation
Peng et al. [1]	Controlled JavaScript HTTP-server task	55.8% faster with Copilot	Benefit in a specific bounded task
Song et al. [6]	Collaborative open-source projects	6.5% project-level productivity increase; integration time also increased	Output gains may involve additional coordination
Becker et al. [4]	Experienced developers; mature repositories	19% longer completion time when AI was allowed	AI may add overhead in context-heavy work

1. Findings And Discussion
The review leads to several findings. First, GenAI has practical applications across multiple SDLC activities, especially implementation support, explanation, testing assistance, and documentation. Second, its benefits are task-dependent: routine and clearly specified work may be easier to accelerate than complex changes requiring extensive project knowledge. Third, the quality of generated code cannot be assumed from fluency or plausibility. Testing, review, and security checks are necessary.

Fourth, productivity must be measured end to end. Time saved during initial code generation may be offset by debugging, integration, review, or rework. Fifth, the developer's expertise remains important. Experienced developers can better judge whether suggestions fit the architecture and requirements, while learners may need additional guidance to avoid accepting incorrect answers. Finally, organizational governance matters: teams need clear rules for approved tools, confidential information, code review, and accountability.

Overall, the most defensible position is that GenAI is an assistive technology. It can increase the range of support available to developers, but it does not remove the need for software engineering knowledge, disciplined testing, and human responsibility.

© 2026 IJSRET

2. Recommended Practices For Responsible Use

- Treat generated code as a draft that requires review, not as a verified solution.
- Provide clear prompts and relevant, non-sensitive context; avoid sharing passwords, keys, personal data, or confidential codeunless explicitly permitted by policy.
- Run unit, integration, regression, and security tests appropriate to the change.
- Use code review, static analysis, dependency scanning, and version control as part of the normal development workflow.
- Ask the tool to explain its assumptions and consider alternative solutions, then verify claims independently.
- Preserve developer accountability: a human reviewer should understand and approve code before it is merged or deployed.
- Evaluate tools using representative tasks and measure quality and rework as well as speed.

3. Future Scope

Future GenAI systems may become more capable of working with larger codebases, following project-specific conventions, and coordinating multi-step development activities. Improved retrieval and repository-aware methods may help models use relevant project documentation and source files instead of relying only on a short prompt. However, broader context access must be balanced with privacy, permission controls, and secure handling of proprietary information.

A second direction is stronger verification. Future systems could connect generated suggestions with test execution, static analysis, vulnerability scanning, and formal verification methods. Rather than merely presenting code, an assistant could provide evidence about which tests were run, what assumptions were made, and which risks remain unresolved. Such evidence would make human review more efficient, although it would not eliminate the need for independent validation.

A third direction is personalized developer assistance. Tools may adapt explanations and suggestions to a developer's experience level, preferred language, and project conventions. Educational use should emphasize reasoning and feedback so that learners build their own skills rather than simply copying generated solutions.

Further research is needed to measure long-term effects on software quality, maintenance cost, team collaboration, and developer learning. Studies should include diverse programming languages, organizations, experience levels, and project types. They should also report tool versions, task characteristics, review time, and quality measures to make results easier to compare. Research agendas have already highlighted open questions concerning dependability, accuracy, transparency, data access, and sustainability [5].

VI. CONCLUSION

Generative Artificial Intelligence is becoming an important part of modern software development. LLM-based tools can assist with code generation, code explanation, debugging, testing, documentation, and design exploration. These capabilities create opportunities to reduce repetitive effort, support learning, and help developers explore solutions. At the same time, generated outputs can be incorrect, insecure, unsuitable for a project, or costly to integrate. Privacy, licensing, accountability, and overdependence are additional concerns.

The reviewed research shows that productivity effects vary by task and context. Some studies report faster task completion or increased open-source output, while another reports slower completion in a setting involving experienced developers and mature repositories. These results demonstrate why broad claims about universal productivity gains are not justified. The value of GenAI should be assessed through the quality and maintainability of delivered software, including the time required for review and rework.

In conclusion, GenAI should be viewed as a supportive tool rather than a replacement for software developers. Responsible use requires clear requirements, human judgment, code review, testing, security controls, and organizational policies. The strongest approach is human–AI collaboration in which AI contributes suggestions and automation while developers retain responsibility for correctness, safety, and final decisions.

REFERENCES

1. S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The Impact of AI on Developer Productivity: Evidence from GitHub Copilot,"arXiv:2302.06590, 2023. doi: 10.48550/arXiv.2302.06590.
2. X. Hou et al., "Large Language Models for Software Engineering: A Systematic Literature Review," ACM Transactions on SoftwareEngineering and Methodology, vol. 33, no. 8, Article 220, 2024. doi: 10.1145/3695988.
3. C. Negri-Ribalta, R. Geraud-Stewart, A. Sergeeva, and G. Lenzini, "A systematic literature review on the impact of AI models on the securityof code generation," Frontiers in Big Data, vol. 7, Article 1386720, 2024. doi: 10.3389/fdata.2024.1386720.
4. J. Becker, N. Rush, E. Barnes, and D. Rein, "Measuring the Impact of Early-2025 AI on Experienced Open-

© 2026 IJSRET

Source Developer Productivity,"arXiv:2507.09089, 2025. doi: 10.48550/arXiv.2507.09089.

5. A. Nguyen-Duc, B. Cabrero-Daniel, A. Przyby←ek, C. Arora, and D. Khanna, "Generative Artificial Intelligence for Software Engineering—A Research Agenda," Software: Practice and Experience, vol. 55, no. 11, pp. 1806–1843, 2025. doi: 10.1002/spe.70005.
6. F. Song, A. Agarwal, and W. Wen, "The Impact of Generative AI on Collaborative Open-Source Software Development: Evidence fromGitHub Copilot," arXiv:2410.02091, 2024. Submission note: Replace the bracketed author, department, college/university, city, and email fields before submitting. Verify your institution's required citation style and journal-specific submission rules.

© 2026 IJSRET