IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p-ISSN: 2278-8727, Volume 27, Issue 1, Ser. 1 (Jan. – Feb. 2025), PP 76-82
www.iosrjournals.org

The Impact of AI-Generated Code on Software Quality and Developer Productivity

Anbarasu Arivoli
Email: anbarasuarivoli@gmail.com
Company: Target, Minneapolis, MN

Abstract: *Artificial intelligence has changed how we write code. AI-based code generation tools now help developers work faster. These tools, such as GitHub Copilot, OpenAI Codex, ChatGPT 4.5, Claude, and DeepSeek R1, generate code by learning from millions of examples. They improve productivity and reduce repetitive tasks. However, their impact on software quality varies. Some AI-generated code is clean and efficient, while other outputs need human refinement. This paper investigates how AI-based code generation tools affect software quality and developer productivity. It compares AI-generated code with human-written code. It also examines productivity metrics and how developers interact with these tools. The study relies on research data and real-world examples. It stays objective and technical. In short, this article offers a detailed, research-intensive overview of the what, how, and results of using AI for coding.*

Keywords: *AI-generated code, software quality, developer productivity, GitHub Copilot, ChatGPT 4.5, OpenAI Codex, Claude, DeepSeek R1, code generation tools, developer trust, technical analysis*

I. Introduction

In recent years, artificial intelligence has transformed the coding world. AI tools now assist developers by generating code from natural language prompts. These models use deep learning and vast training data. They gain insights by analyzing millions of lines of code. As a result, they can produce working code quickly. Moreover, they help reduce routine work and free developers to focus on complex tasks.

Many companies now use AI-based code generators. GitHub Copilot and OpenAI Codex were among the first to be widely adopted. Later, models like ChatGPT 4.5, Claude, and DeepSeek R1 entered the scene. Each model has its strengths and weaknesses. For example, ChatGPT 4.5 often generates code with high readability, while DeepSeek R1 shows promise in advanced reasoning and mathematical tasks. Furthermore, some tools emphasize speed, and others offer more detailed problem-solving.

Research shows that developers using AI coding assistants can complete tasks 20% to 45% faster than traditional methods. Studies also indicate that junior developers tend to benefit the most from these tools. At the same time, quality remains a concern. While AI-generated code can be efficient, it sometimes lacks the nuanced understanding of experienced human developers. Thus, developers must review and refine AI outputs.[1]

Additionally, the integration of these tools into development workflows has reshaped the software life cycle. AI models now work alongside version control systems and integrated development environments (IDEs) to help maintain a consistent codebase. They offer context awareness through vectorization techniques that search large code repositories for relevant patterns. Developers now benefit from faster prototyping and debugging. However, they still need to verify that the generated code meets the project's architectural standards and quality benchmarks. [2]

These rapid changes have sparked much discussion. Developers now debate the best models and workflows. They weigh the advantages of speed against potential risks to code quality. In the following sections, this article will delve deeper into the specifics of AI in code generation, comparisons with human-written code, productivity metrics, and developer trust.

II. Literature Review

The integration of AI into software development has garnered significant scholarly attention in recent years. Deniz et al. [1] show that generative AI tools are transforming developer productivity by accelerating code commits and reducing repetitive tasks. Complementing this view, Soni et al. [2] detail best practices for integrating AI into the software development life cycle, offering technical guidelines and impact analysis that help bridge the gap between traditional coding practices and modern automated approaches.

Research also highlights concerns regarding the future role of junior developers. Pantin [3] argues that although AI-generated code can speed up routine tasks, it may hinder the learning curve for less experienced developers by limiting their exposure to essential debugging processes. In parallel, literature on automatic

algorithm generation [4] illustrates that when handling complex tasks, AI may insert redundant checks or opt for non-optimal algorithms, thus challenging the balance between efficiency and code quality.
Trust in AI-generated code remains a critical area of investigation. Brown et al. [5] examine factors that influence trust in AI code completion, noting that consistency and transparency are vital for developer acceptance. Afroogh et al. [6] further explore the progress and persistent challenges in building trust in AI systems, emphasizing the need for clear explanations and robust error handling. Resources like those from IBM [7] serve to demystify AI code-generation software, thereby enhancing developers' confidence in these emerging tools.

A comparative analysis between AI-generated and human-written code is crucial to understanding both the benefits and limitations of these systems. A preliminary study [8] indicates that while AI can rapidly generate functional code snippets, human-written code often includes richer context, error checking, and commentary that enhance maintainability. Shah [9] discusses the rise of AI agents in enterprise software development, suggesting that the future will increasingly rely on these systems to assist developers. However, human oversight remains indispensable to ensure that critical business logic and complex design considerations are properly addressed.

III. Problem Statement

AI-generated code introduces challenges that affect both the quality of software and the productivity of developers [2] [3]. The core problem lies in balancing the speed of code generation with the reliability and maintainability of the resulting code. AI tools can produce code quickly, but the outputs sometimes lack consistency or context. This disparity raises issues for integration into large codebases and places an extra burden on developers who must verify and refine the generated code.

3.1. Inconsistent Code Quality and Maintainability

One major problem is the inconsistency in code quality. AI models generate code based on patterns they have learned, yet they often produce solutions that vary in quality [3]. For instance, consider the simple Python function generated by an AI tool:

```
def calculate_area(radius):
    pi = 3.14159
    if radius > 0:
        area = pi * (radius ** 2)
        return area
    else:
        return None
```

***Figure 1**: AI Generated code*

This code is clear and concise. However, when the AI tackles more complex tasks, it may insert redundant checks or use non-optimal algorithms [2] [5]. The generated code can have unnecessary nested conditions or duplicate logic that increases technical debt. Moreover, the code may not adhere to established design patterns, making it harder to maintain. Developers must refactor and test the outputs to ensure they integrate well with the rest of the system. Thus, although AI speeds up code production, it introduces variability that undermines long-term maintainability [3].

3.2. Integration Challenges in Legacy Codebases

Another problem is integrating AI-generated code into existing legacy systems. Modern AI tools often generate code in isolation, without the full context of an established codebase. Legacy systems usually have specific architectural constraints and coding conventions that the AI might overlook. For example, an AI might generate a new module that does not follow the dependency injection pattern used in the existing system. [3]
The code may work independently but can break the overall system when merged. In these cases, developers must manually align the new code with the legacy architecture. This process demands additional time and increases the risk of introducing bugs during integration, as the AI-generated code might not seamlessly interface with other components. [2] [4]

3.3. Developer Trust and Oversight

A further challenge is establishing trust in AI-generated code [5]. Developers need to verify that the code compiles and performs the intended function reliably. Although AI tools offer a significant boost in productivity, they cannot fully replace the human ability to understand the deeper context of a problem.

Developers must scrutinize AI outputs, often line by line, to ensure correctness. For example, the AI might generate a function that appears efficient but omits important error handling. Developers then have to insert additional code to manage exceptions or to validate inputs. This oversight increases the cognitive load and reduces the net productivity gains promised by AI tools. [3]

Trust is built gradually, and until AI can guarantee context-aware decisions, human intervention remains crucial. [6]

3.4. Productivity Measurement and Overhead

Measuring the productivity gains from AI coding tools also presents technical challenges. Although studies report improvements ranging from 20% to 45%, these figures often do not account for the additional time developers spend on reviewing and debugging AI-generated code.

For instance, a developer might complete a coding task faster with AI assistance but later invest extra time to understand the AI's internal logic and correct any flaws. This overhead can offset some of the initial speed improvements. Moreover, integrating these tools into the development workflow requires a strong setup with version control and continuous integration pipelines.

The extra steps needed to ensure compatibility and maintain quality further complicate productivity measurements. In technical terms, while the raw code generation rate may be high, the effective productivity is determined by the overall cycle time from prompt to deployable solution. [3][6]

3.5. Complexity in Domain-Specific Code Generation

AI tools often struggle with complex, domain-specific requirements, especially without a well-detailed and step-by-step prompt. In software projects that involve business logic or specialized algorithms, the AI might produce code that is syntactically correct but semantically flawed. [2][5]

For example, when generating code for a financial application, the AI might misinterpret regulatory constraints or calculation nuances. Consider this pseudo-code for a risk assessment calculation:

```
def assess_risk(portfolio):
    risk_score = 0
    for asset in portfolio:
        risk_score += asset.volatility * asset.value
    return risk_score / len(portfolio)
```

Figure 2: *AI Generated pseudo code for risk assessment calculations*

While the logic here is straightforward, in a real financial application, the risk calculation might need to incorporate factors like covariance, liquidity, and market conditions. The AI-generated code may omit these critical aspects. As a result, developers must invest time in adjusting the algorithms to meet specific domain standards. This complexity increases the risk of errors and diminishes the utility of AI-generated code in high-stakes environments. [5]

IV. AI in Code Generation Tools

Artificial intelligence has become central to modern coding workflows. Tools like GitHub Copilot and OpenAI Codex rely on deep learning to complete code based on natural language prompts. These tools learn from extensive code repositories and produce outputs rapidly. [6]

Their strengths include accelerating repetitive coding tasks and offering suggestions that help developers overcome writer's block. However, users report issues such as inconsistent code quality and a lack of deep context awareness, which sometimes forces developers to invest extra effort in debugging and refactoring. [1]

Table 1 compares six of the most prominent code generation tools currently out there, used by professionals [7].

Tool Name	Developer/Origin	Strengths	Common Issues
GitHub Copilot	GitHub / OpenAI	Fast autocompletion; seamless IDE integration; reduces boilerplate code	Occasional inaccuracies; can produce verbose or over-engineered solutions
OpenAI Codex	OpenAI	Strong language understanding; efficient for generating standard code snippets	Struggles with complex logic; sometimes ignores architectural context
ChatGPT 4.5	OpenAI	High readability; conversational code generation; supports iterative refinement	May generate code that lacks error handling; tends to simplify complex tasks excessively
Claude 3.7 Sonnet	Anthropic	Excellent at structured reasoning; produces clean and legible code	Response time can be slow; less effective with legacy code or highly specialized frameworks
DeepSeek R1	DeepSeek (China)	Advanced reasoning in mathematical and coding tasks; cost-efficient; high precision	Integration issues with legacy systems; sometimes produces code that deviates from conventional patterns
Codebuddy	Independent / Startup	Combines planning with generation; offers context-aware suggestions; strong in multi-file tasks	Occasional difficulties with large codebases; limited support for some niche programming languages

Table 1: *Comparison of 6 common AI Code generators.*

Each of these tools brings a unique complement of speed, ease-of-use, and technical capability. For instance, GitHub Copilot is praised for its rapid autocompletion and tight IDE integration, but it may sometimes require manual oversight to ensure that its suggestions fit within a project's architecture.

OpenAI Codex is strong in generating standard code patterns, yet it occasionally fails when faced with highly complex logic. ChatGPT 4.5 and Claude 3.7 Sonnet excel in generating legible code with proper structure, though both may simplify sophisticated problems. DeepSeek R1 shows promise in advanced reasoning tasks, especially in math-intensive or algorithmic challenges, while Codebuddy has been noted for its planning step that preemptively structures the generated code. Common across all these tools is the challenge of aligning the generated output with the larger project context, which remains a key area where human expertise is essential. [7]

V. Comparisons Between AI-Generated and Human-Written Code

When comparing AI-generated code with human-written code, technical differences emerge that affect readability, maintainability, and overall design. To express these points, consider two examples. In the first example (Figure 3), a simple function calculates the factorial of a number. [8]

In the second example (Figure 4), a class implements a basic queue data structure. Both examples include code snippets from an AI tool and a human developer.

For the factorial function, the AI-generated code might appear as follows:

```
def factorial(n):
    if n < 0:
        return None
    elif n == 0:
        return 1
    else:
        result = 1
        for i in range(1, n + 1):
            result *= i
        return result
```

***Figure 3**: AI Generated Factorial Python code*

This code is clear and concise. It checks for negative input and handles the base case explicitly. However, the human-written version might include additional comments and a slightly different structure:

```
def factorial(n):
    # Return None for negative input, as factorial is undefined
    if n < 0:
        return None
    # Base case: factorial of 0 is defined as 1
    if n == 0:
        return 1
    # Compute factorial using iterative multiplication
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

***Figure 4**: Human-written Factorial Python Code*

Both functions achieve the same goal, but the human-written code includes inline comments. These comments provide context and reasoning behind each condition, making the logic more transparent to someone maintaining the code. The AI version, while functionally correct, may lack these contextual hints that are beneficial in a collaborative environment. [5][8]

Now, consider a more complex example: a class that implements a basic queue. The AI-generated version may look like this:

```
class Queue:
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if not self.items:
            return None
        return self.items.pop(0)

    def is_empty(self):
        return len(self.items) == 0

    def size(self):
        return len(self.items)
```

***Figure 5**: AI-generated basic queue code*

In contrast, a human-written version of the same functionality might include better error handling and more detailed documentation:

```
class Queue:
    def __init__(self):
        # Initialize an empty list to store queue items
        self.items = []

    def enqueue(self, item):
        # Add an item to the end of the queue
        self.items.append(item)

    def dequeue(self):
        # Check if the queue is empty before attempting to dequeue
        if self.is_empty():
            # Return None to indicate that the queue is empty
            return None
        # Remove and return the first item in the queue
        return self.items.pop(0)

    def is_empty(self):
        # Return True if the queue is empty, else False
        return len(self.items) == 0

    def size(self):
        # Return the number of items currently in the queue
        return len(self.items)
```

Figure 6*: Human-written basic queue code*

In this second example, both versions provide the same core functionality. However, the human-written code incorporates detailed comments and a slightly more structured error check by reusing the is_empty() method within dequeue(). This approach promotes code reuse and clarity. Moreover, human developers often adopt naming conventions and code structuring practices that align with a project's style guide, ensuring consistency across a large codebase.

This shows that while AI-generated code offers speed and the ability to produce boilerplate code quickly, it may sometimes lack the depth of explanation and reliability that human-written code exhibits. The AI versions tend to follow common patterns and might omit critical contextual information, whereas human developers add nuanced comments, error checks and adhere to specific architectural guidelines. These differences showcase the importance of human oversight in maintaining high-quality, maintainable code while utilizing the productivity gains of AI tools.

VI. Productivity Metrics in Developer Workflows Using AI-Based Tools

Developers now measure productivity by the amount of code written as well as by how quickly they resolve issues and complete tasks. Many teams track metrics like the number of code commits, lines of code produced, and the time taken to fix bugs. Recent studies indicate that AI-based tools can boost productivity by 20% to 45%. [7]

These improvements are seen in faster iteration times, increased code commit frequencies, and more frequent code refactoring. Developers now integrate automated logging within their workflows. For example, a simple logging function in Python can record the time a commit is made and the duration of coding sessions [1][5].

```
import time

def log_commit(commit_message):
    start_time = time.time()
    # Developer writes code and makes a commit
    # (Simulated by a delay here)
    time.sleep(2)
    end_time = time.time()
    duration = end_time - start_time
    print(f"Commit '{commit_message}' completed in {duration:.2f} seconds")

log_commit("Added new authentication module")
```

Figure 7*: AI-Generated recording code*

This snippet shows how teams can measure the time it takes to complete a coding task. In real-world scenarios, these logs feed into analytics systems that aggregate data on task completion times and error rates. AI tools help generate code and assist in detecting bugs early by suggesting fixes. As a result, the entire software delivery pipeline sees improvement. [1][3][7][9]

Continuous integration systems use these metrics to compare performance before and after the introduction of AI assistants. Transitioning between manual and AI-assisted workflows provides quantitative data that supports investment in these tools. Despite the gains in speed, teams must account for time spent on debugging AI-generated code. Therefore, overall productivity is best measured by balancing output quantity with code quality and system stability. [9]

VII. Developer Trust and Interaction with AI-Generated Code

Establishing trust in AI-generated code is a critical challenge when it comes to automating workflows or simply speeding it up. Developers must decide whether to accept a suggestion or modify it based on their understanding of the project. Trust builds over time when the AI consistently produces correct and maintainable code. [5]

However, many developers report that AI-generated code sometimes lacks deeper context and may miss edge cases. For example, when an AI tool produces a function without clear error handling, developers must inspect and improve the code manually. This oversight process increases cognitive load but is necessary for long-term project stability.

Consider a situation where an AI generates a function for data parsing:

```
def parse_data(data):
  # AI-generated code without error handling
  return data.split(',')
```

***Figure 8**: AI-generated data parsing function*

A human-written version of the same might look like this:

```
def parse_data(data):
  # Check if data is a string and not empty
  if not isinstance(data, str) or not data:
    return []
  # Split the data by comma and strip whitespace from each element
  return [item.strip() for item in data.split(',')]
```

***Figure 9**: AI-generated data parsing function*

Here, the human version includes validation and processing steps that ensure reliability. Developers interact with AI tools by reviewing such differences. They often use code review systems and pair programming sessions to validate and improve AI outputs. Moreover, some teams integrate feedback loops into their development process. When a tool generates code that consistently misses critical patterns, developers can train it further or adjust their prompts. [8][5]

Trust also comes from transparency. Some tools now expose part of their internal reasoning or prompt history. This exposure helps developers understand how the AI arrived at a solution, reinforcing or undermining their trust based on observable logic. Over time, as AI models refine their outputs and align better with team standards, trust increases. Yet, a human in the loop remains essential to manage risks, maintain quality, and ensure that critical business logic is correctly implemented.

VIII. Recommendations

To make the most of AI in coding while mitigating risks, organizations should adopt a flexible strategy. First, it is essential to integrate AI tools directly into the existing development environment. Tight IDE integration, as seen with GitHub Copilot and Codebuddy, speeds up the coding process and minimizes context switching.

Next, teams must establish rigorous code review practices. Developers should use continuous integration pipelines that automatically run tests on AI-generated code. This ensures that any deviations from expected behavior are caught early. Additionally, teams should invest in training sessions where developers learn how to effectively prompt and interpret AI outputs. Understanding the limitations and strengths of AI tools, developers can craft better prompts and provide feedback that leads to iterative improvements.

Furthermore, measuring productivity requires more than just counting lines of code; teams should monitor metrics such as commit frequency, error rates, and time-to-deploy. These metrics help in understanding the net benefit of AI integration.

Finally, organizations must create a culture that balances innovation with accountability. Developers should be encouraged to experiment with AI tools while maintaining high standards for code quality and system reliability. With continuous assessment of both the quantitative and qualitative aspects of AI-assisted development, teams can optimize workflows and achieve sustainable productivity gains while preserving the integrity of their software systems.

IX. Conclusion

Our investigation shows that AI-generated code tools have transformed software development. These tools help developers complete tasks faster, reducing routine work and boosting productivity. However, they do not entirely replace human expertise. The generated code can lack context and sometimes miss error handling. Human oversight remains essential.

Our analysis shows that tools such as GitHub Copilot, OpenAI Codex, ChatGPT 4.5, Claude, DeepSeek R1, and Codebuddy each offer unique strengths. They improve coding speed and provide useful suggestions. Yet, they also introduce challenges in consistency, integration with legacy systems, and maintainability. Detailed

comparisons show that while AI can produce legible and efficient code, human-written code often includes critical error checks and explanatory comments that enhance clarity and reliability.

Productivity metrics indicate significant time savings and increased commit frequencies. However, the time spent reviewing and debugging AI-generated code can offset these gains. Developer trust builds over time as AI outputs improve. Feedback loops and iterative testing are key to aligning AI-generated code with project standards.

References

[1] B. K. Deniz, C. Gnanasambandam, M. Harrysson, A. Hussin, and S. Srivastava, "Unleashing developer productivity with generative AI," *McKinsey & Company*, Jun. 27, 2023. https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/unleashing-developer-productivity-with-generative-ai

[2] A. Soni, A. Kumar, R. Arora, and R. Garine, "Integrating AI into the Software Development Life Cycle: Best Practices, Tools, and Impact Analysis," *SSRN*, Jan. 2024, doi: 10.2139/ssrn.4918992.

[3] C. Pantin, "The impact of AI-generated code on the future of junior developers," *Theseus*, 2024. https://www.theseus.fi/handle/10024/866717

[4] "Automatic generation of algorithms," *Google Books*. January, 2025 https://books.google.com.pk/books?hl=en&lr=&id=ke04EQAAQBAJ&oi=fnd&pg=PP1&dq=when+the+AI+tackles+more+complex+tasks,+it+may+insert+redundant+checks+or+use+non-optimal+algorithms+&ots=M0bvnrGd-7&sig=f6ro-F8ONaJEpNQpA7BsvBqdsmc&redir_esc=y#v=onepage&q&f=false

[5] A. Brown, S. D'Angelo, A. Murillo, C. Jaspan, and C. Green, "Identifying the Factors That Influence Trust in AI Code Completion," *Proceedings of the 1st ACM International Conference on AI-Powered Software*, pp. 1–9, Jul. 2024, doi: 10.1145/3664646.3664757.

[6] S. Afroogh, A. Akbari, E. Malone, M. Kargar, and H. Alambeigi, "Trust in AI: progress, challenges, and future directions," *Humanities and Social Sciences Communications*, vol. 11, no. 1, Nov. 2024, doi: 10.1057/s41599-024-04044-8.

[7] I. Education, "AI code-generation software: What it is and how it works?," *AI code-generation software*, Nov. 25, 2024. https://www.ibm.com/think/topics/ai-code-generation

[8] "A Comparative Analysis between AI Generated Code and Human Written Code: A Preliminary Study," IEEE Conference Publication | IEEE Xplore, Dec. 15, 2024. https://ieeexplore.ieee.org/abstract/document/10825958/

[9] S. Shah, "THE RISE OF AI AGENTS IN ENTERPRISE SOFTWARE DEVELOPMENT," Oct. 11, 2024. https://lib-index.com/index.php/IJCET/article/view/IJCET_15_05_074