International Journal of Computer Science and Mobile Computing
A Monthly Journal of Computer Science and Information Technology

Zainab Abd-al Abbas Muhsen, Volume 15, Issue 7, July 2026, pg. 100-117
(An Open Accessible, ISI Indexed, Fully Refereed and Peer Reviewed Journal)
ISSN: 2320-088X
Impact Factor: 7.056

Evaluating the Reliability and Security of AI-Generated Source Code: A Comparative Study of Large Language Models

Zainab Abd-al Abbas Muhsen

Al-Furat Al-Awsat Technical University, Technical Institute of Babylon, Iraq

DOI: https://doi.org/10.47760/ijcsmc.2026.v15i07.007

Abstract: The rapid advancement of Large Language Models (LLMs) has significantly transformed software development by enabling automated source code generation. Despite their increasing adoption, concerns remain regarding the reliability, correctness, maintainability, and security of the generated code. This study aims to evaluate and compare the performance of leading LLMs in generating source code across a variety of programming tasks and application domains. The proposed research assesses multiple state-of-the-art language models using a comprehensive evaluation framework that includes functional correctness, code quality, computational efficiency, maintainability, and vulner. This study evaluates the reliability and security of AI-generated source code by comparing the performance of leading Large Language Models (LLMs) across various programming tasks and application domains. It employs a comprehensive evaluation framework that assesses functional correctness, code quality, computational efficiency, maintainability, and vulnerability to security threats. The findings aim to provide insights into the strengths and weaknesses of different LLMs in automated code generation, guiding developers in selecting appropriate tools for their software projects.

International Journal of Computer Science and Mobile Computing
A Monthly Journal of Computer Science and Information Technology

Zainab Abd-al Abbas Muhsen, Volume 15, Issue 7, July 2026, pg. 100-117
(An Open Accessible, ISI Indexed, Fully Refereed and Peer Reviewed Journal)
ISSN: 2320-088X
Impact Factor: 7.056

1. Introduction

1.1. Background on Large Language Models (LLMs)

Large Language Models (LLMs) have significantly impacted software development through AI-generated source code. Trained on vast collections of text and code, these models produce human-like writing and handle complex coding tasks across various programming languages. This evolution began with Microsoft's IntelliCode in 2018, which used machine learning to suggest context-aware code. Later, in 2021, GitHub's Copilot advanced this by drawing on extensive real-world codebases to offer smarter coding recommendations.

However, the rise of LLMs also brings challenges related to code quality. Correctness, security, and maintainability are essential to ensure code functions well and avoids vulnerabilities. Research indicates that LLMs can generate secure code in some situations but may produce risky outputs depending on the language or complexity of the task.

As developers rely more on these models, LLMs serve not only to create code snippets but also to influence software design and refactoring. Current efforts focus on reliably measuring their effectiveness using tests and quality metrics that assess correctness and robustness.

Improving how developers work with AI tools remains important. Providing better guidance and promoting awareness help enhance the quality of AI-generated code. Still, human oversight is necessary because AI cannot replace the expert judgment and insight of experienced developers. See references: [2], [4] and [1].

1.2. Importance of Source Code Generation

AI-driven source code generation using large language models is transforming software development. These models automate coding tasks, boosting productivity and allowing developers to focus on strategic design rather than repetitive syntax. Their impact extends beyond code writing to shape software architecture, test planning, and refactoring.

AI integration aligns with a broader shift toward automation and smart assistance. Tools like GitHub Copilot use extensive code libraries to offer real-time suggestions that match developers' intentions. This increases coding speed and reduces manual errors while promoting best practices.

However, concerns about code quality and security persist. Research shows AI-generated code can be secure or vulnerable depending on the language and task. This mixed result highlights the need for clear guidance and user awareness to ensure safe, reliable code.

Robust evaluation systems play a key role by developing metrics that focus on security vulnerabilities in AI outputs. Addressing these weaknesses is vital to maintain trust and ensure software remains dependable over time.

The rise of AI assistants also changes team dynamics. Developers must balance their judgment with AI

International Journal of Computer Science and Mobile Computing
A Monthly Journal of Computer Science and Information Technology

Zainab Abd-al Abbas Muhsen, Volume 15, Issue 7, July 2026, pg. 100-117
(An Open Accessible, ISI Indexed, Fully Refereed and Peer Reviewed Journal)
ISSN: 2320-088X
Impact Factor: 7.056

advice, fostering collaboration that upholds quality standards.
Future progress in training and evaluation frameworks will require ongoing workflow adjustments and cross-disciplinary cooperation to handle new challenges posed by advancing AI code generation. This evolution significantly enhances efficiency, quality assurance, and teamwork in programming. See references: [7], [4], [1] and [5].

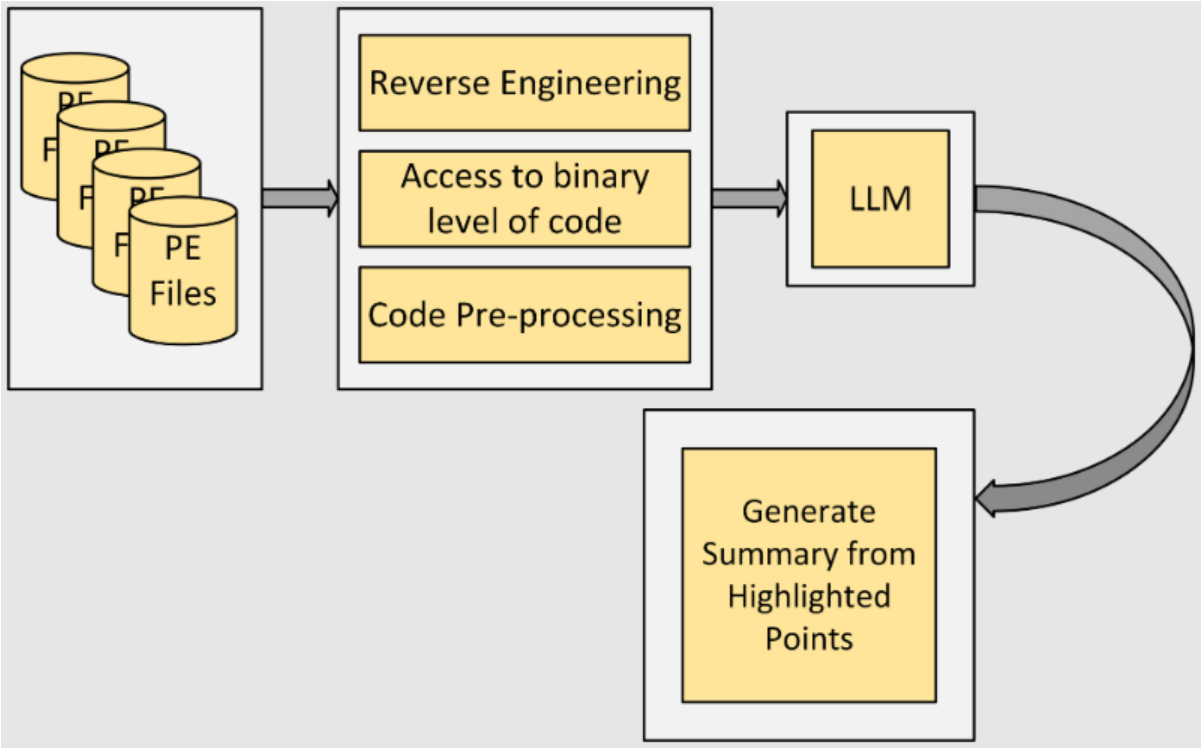

Figure 1: Binary-code summary using LLM (source: reference [5])

1.3. Objectives of the Study

This study focuses on exploring the strengths and weaknesses of large language models (LLMs) in generating source code. We work to evaluate how well different LLMs perform across a variety of programming tasks, aiming to ensure the outputs are dependable and can improve current software development practices. As AI-generated code becomes more common in engineering workflows, this raises important questions about quality control and security risks.
A key aim is to develop a thorough evaluation framework for AI-crafted code. This involves selecting language models that meet industry standards in functionality and security. As outlined in section 3.1, we

International Journal of Computer Science and Mobile Computing
A Monthly Journal of Computer Science and Information Technology

Zainab Abd-al Abbas Muhsen, Volume 15, Issue 7, July 2026, pg. 100-117
(An Open Accessible, ISI Indexed, Fully Refereed and Peer Reviewed Journal)
ISSN: 2320-088X
Impact Factor: 7.056

will pay close attention to syntax accuracy and compliance with coding conventions.
Beyond standard performance measures, we address the shortcomings spotted in previous research around how AI-generated code is tested. Section 2.3 highlights that inconsistent and scattered evaluation methods make it hard to fully assess generated code. Our study works to fill this gap by proposing a clear, organized approach that covers functional correctness, maintainability, and vulnerability to security issues.
Moreover, we want to spotlight best practices for developers using AI-generated code in their projects. Building on insights from section 6.1 about the essential role of human review, we suggest practical ways for developers to blend LLM assistance into their workflows without sacrificing quality.
Finally, we hope to guide future research aimed at boosting LLM capabilities for source code generation while tackling security concerns raised in sections 6.2 and 7.1. By investigating improvements to training datasets and evaluation frameworks, this work sets the stage for safer and more reliable AI-driven software development. See references: [19] and [5].

2. Literature Review

2.1. Overview of Current Research on AI-Generated Code

The rise of AI-generated source code owes much to the progress made in large language models (LLMs). Current studies reveal different facets of LLMs' ability to create working code, while also pointing out persistent challenges tied to code quality and security risks. Research often finds that, although LLMs can produce code that looks correct syntactically, they struggle with deeper issues like semantic accuracy and guarding against vulnerabilities. For example, models such as ChatGPT and Gemini show uneven results depending on the programming language, with Python outputs usually proving more reliable than those in C or C++.
Researchers have built evaluation systems that measure AI-generated code based on factors like functional accuracy, speed, and maintainability. New benchmarks—such as CodeMirage and AICD Bench—have appeared to offer robust assessments grounded in practical scenarios. These tools try to close gaps left by earlier evaluations by covering multiple programming languages and examining performance shifts under varying conditions.
Despite these encouraging steps forward, notable research gaps persist. Many experts stress the urgency of better techniques to detect AI-generated code, fueled by worries about copyright breaches and biased training sets. Additionally, studies confirm that human judgment remains indispensable for checking and validating the code output from these AI models. As this field continues to evolve, digging deeper into how human skills and LLM capabilities can work hand in hand will be vital for delivering top-notch software development results. See references: [2], [14], [8], [13], [1], [3], [6] and [9].

International Journal of Computer Science and Mobile Computing
A Monthly Journal of Computer Science and Information Technology

Zainab Abd-al Abbas Muhsen, Volume 15, Issue 7, July 2026, pg. 100-117
(An Open Accessible, ISI Indexed, Fully Refereed and Peer Reviewed Journal)
ISSN: 2320-088X
Impact Factor: 7.056

2.2. Analysis of Existing Evaluation Frameworks

Evaluating AI-generated source code involves different frameworks that check various aspects of code quality and security. These methods look at everything from syntax and semantic accuracy to maintainability and potential vulnerabilities. For example, LLMSecEval offers datasets crafted to test how well language models handle security-related questions in common software areas. This approach underscores the need for benchmarks that prioritize security reasoning, not just code-writing skills.

Another significant development is CWEval-Bench, a multilingual benchmark that covers many types of Common Weakness Enumeration (CWE) issues. This tool helps researchers compare LLM-generated code against known vulnerability patterns, highlighting any gaps in producing both functional and secure code.

Many evaluations rely on metrics like Pass rates and Bilingual Evaluation Understudy scores to measure performance. Yet, these numbers can miss important context around security challenges, calling for more sophisticated methods that reflect real-world usage.

Additionally, recent research has pointed out flaws in traditional evaluation techniques by showing that different analysis tools vary in detecting vulnerabilities. Relying on just one tool can cause some threats to slip through unnoticed. This insight suggests adopting a combined approach, blending multiple evaluation tools to boost the accuracy and consistency of code assessments.

Efforts like EvalPlus represent a step forward by expanding beyond standard benchmarks. It introduces a wide range of test cases created through advanced mutation methods, helping to gauge the correctness of AI-generated code across diverse programming tasks. These fresh techniques bring us closer to thoroughly understanding the quality and security of code produced by language models. See references: [2], [3], [1] and [5].

2.3. Gaps in Existing Studies

Research on AI-generated source code reveals several important gaps that need more focus. One major issue is the inconsistent use of evaluation metrics for code quality and security. Current methods rely on conventional metrics that don’t fully capture the complexities of AI-produced code. Frameworks like LLMSecEval and CWEval-Bench include security alongside functionality but lack specialized metrics for language models. Without these tools, assessments may mislead and slow progress.

Most studies also examine isolated features of code generation instead of the combined abilities of generating and understanding code. This fragmented approach hinders understanding how improvements in one area could support the other, keeping development uneven.

Human oversight and interpretability raise concerns about trust and usability. Programmers often need additional context that language models don’t provide. This shows a clear need for better tools to improve collaboration, making code review smoother and more reliable.

Long-term maintenance is rarely studied, yet automated generation adds layers that risk technical debt and sustainability. Research should explore maintaining AI-produced code effectively over time.

International Journal of Computer Science and Mobile Computing
A Monthly Journal of Computer Science and Information Technology

Zainab Abd-al Abbas Muhsen, Volume 15, Issue 7, July 2026, pg. 100-117
(An Open Accessible, ISI Indexed, Fully Refereed and Peer Reviewed Journal)
ISSN: 2320-088X
Impact Factor: 7.056

Security vulnerabilities in AI-generated code require deeper analysis to understand their causes and prevention. Additionally, open source software poses challenges around licensing and ownership when AI code merges with existing projects. This issue demands more attention to ensure smoother integration and cooperation between developers and AI tools. See references: [15], [19], [21] and [5].

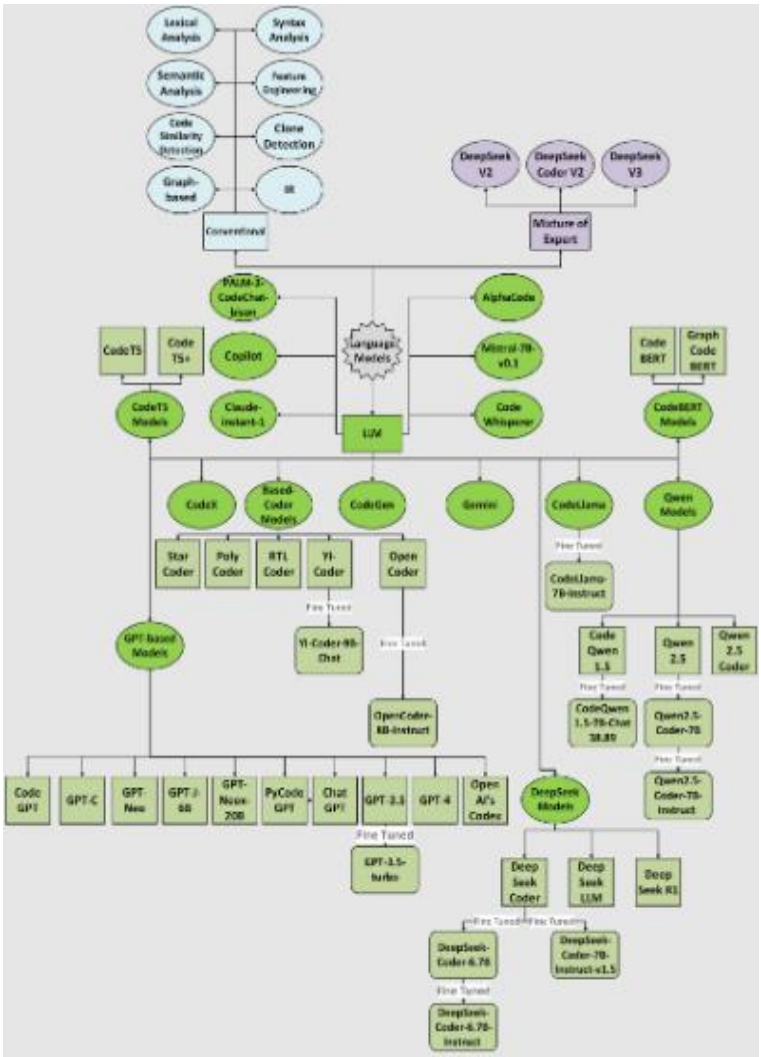

Figure 2: A taxonomy of recent NLP and LLM models for source code analysis (source: reference [5])

International Journal of Computer Science and Mobile Computing
A Monthly Journal of Computer Science and Information Technology

Zainab Abd-al Abbas Muhsen, Volume 15, Issue 7, July 2026, pg. 100-117
(An Open Accessible, ISI Indexed, Fully Refereed and Peer Reviewed Journal)
ISSN: 2320-088X
Impact Factor: 7.056

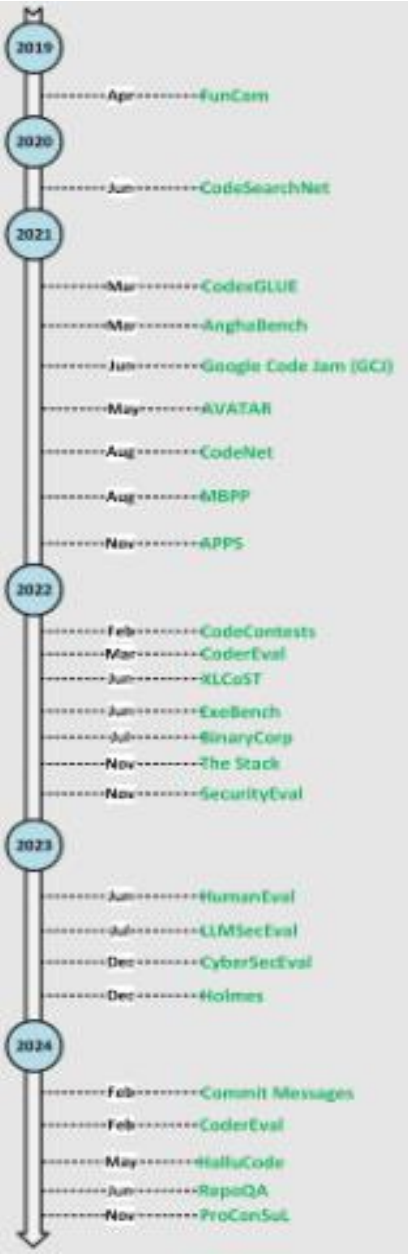

Figure 3: Time Line of LLM Datasets. (source: reference [5])

International Journal of Computer Science and Mobile Computing
A Monthly Journal of Computer Science and Information Technology

Zainab Abd-al Abbas Muhsen, Volume 15, Issue 7, July 2026, pg. 100-117
(An Open Accessible, ISI Indexed, Fully Refereed and Peer Reviewed Journal)
ISSN: 2320-088X
Impact Factor: 7.056

Survey	Code-related Tasks						
	CU	CD	CdC	CS	CG	ComG	SA
(60,)	✗	✗	✗	✗	✓	✗	✗
(115,)	✗	✗	✗	✗	✓	✗	✗
(126,)	✓	✗	✗	✗	✓	✗	✗
(1,)	✓	✗	✗	✗	✗	✗	✗
(167,)	✗	✗	✗	✓	✓	✗	✓
(200,)	✗	✗	✗	✗	✓	✗	✗
Our Work	✓	✓	✓	✓	✓	✓	✓

Table 1: Our work vs. other surveys- Code Understanding (CU), Code Disassembling (CD), Code Decompiling (CdC), Code Summarization (CS), Code Generation (CG), Comment Generation (ComG), Security Analysis (SA) (source: reference [5])

3. Methodology

3.1. Selection Criteria for Language Models

Choosing the right language models (LLMs) to evaluate AI-generated code requires careful consideration of key factors. These models must handle diverse programming challenges, from simple functions to complex systems, because developers face many coding problems.

A major factor is the model's training data. LLMs should be trained on large, varied datasets covering many languages and styles. Research shows datasets like CodeSearchNet and CodeXGLUE improve performance in tasks such as code generation and summarization. Domain-specific datasets also enhance accuracy in real-world use cases.

Models must produce both syntactically correct and semantically meaningful code. Evaluations should test code functionality and overall quality through thorough functional tests to ensure expected behavior.

LLM performance needs comparison against industry benchmarks using metrics like accuracy, speed, maintainability, and adherence to best practices. This provides a fair basis for assessment.

Security is vital since AI-generated code can introduce vulnerabilities. Models should be assessed by their security track record using frameworks like LLMSecEval and CWEval-Bench.

Human judgment remains essential. Expert developers offer feedback to improve trustworthiness and spot hidden flaws. Their input supports validation and ensures developers carefully review code for errors or risks.

International Journal of Computer Science and Mobile Computing
A Monthly Journal of Computer Science and Information Technology

Zainab Abd-al Abbas Muhsen, Volume 15, Issue 7, July 2026, pg. 100-117
(An Open Accessible, ISI Indexed, Fully Refereed and Peer Reviewed Journal)
ISSN: 2320-088X
Impact Factor: 7.056

In summary, selecting language models for evaluating AI code requires ensuring versatility, using robust training data, meeting standards for correctness and security, and keeping human oversight central throughout. See references: [2] and [18].

3.2. Evaluation Framework Components

Evaluating AI-generated source code requires examining both its functionality and quality. Functional Correctness Assessment checks if the code works as intended by running various tests to confirm its reliability. Expert human judgment supports automated testing by manually verifying results.

Code Quality Metrics assess syntax accuracy, reliability, maintainability, and complexity. They detect bugs causing errors and identify vulnerabilities that pose security risks. Maintainability evaluates how easy it is for developers to read and update the code.

Computational Efficiency Metrics measure resource use, such as processing time and memory. Efficient code matters in environments with limited resources, as poor performance can increase costs and degrade user experience.

Maintainability Assessment also focuses on clarity and organization, helping developers sustain and improve code over time. Even functional code may hide challenges, especially across different programming languages.

Security Vulnerability Analysis highlights weak points exploitable by attackers. Automated tools like SonarQube and CodeQL find potential issues, but human review is essential to confirm real threats.

In summary, assessing AI-generated code calls for balancing functional testing, quality checks, efficiency measures, and security reviews. Using both automated tools and expert analysis builds confidence when integrating AI-created code into projects. See reference [1].

4. Experimental Setup and Data Collection

4.1. Environment Configuration for Testing LLMs

To assess how well large language models (LLMs) perform in generating source code, a carefully controlled environment was set up to guarantee uniformity and reproducibility across diverse programming challenges. The focus lay on standardizing every step: code generation, compilation, execution, and validation. Experiments took place on a Lenovo ThinkPad E570, outfitted with a 7th-generation Intel Core i7 processor, 16 GB DDR4 RAM, and a 256 GB SSD. This setup offered enough power while remaining portable enough to smoothly integrate LLMs into the coding workflow.

Debian 12 was the operating system of choice due to its reliability and efficient package management, both essential when juggling multiple programming languages. This selection helped reduce technical hiccups and maintain a steady, interruption-free testing process.

International Journal of Computer Science and Mobile Computing
A Monthly Journal of Computer Science and Information Technology

Zainab Abd-al Abbas Muhsen, Volume 15, Issue 7, July 2026, pg. 100-117
(An Open Accessible, ISI Indexed, Fully Refereed and Peer Reviewed Journal)
ISSN: 2320-088X
Impact Factor: 7.056

Different software tools were employed for handling specific languages. For Java development, OpenJDK version 17.0.8 (LTS) ensured support for modern language features that LLMs might produce. Python code testing used Python 3.11.9, enabling access to up-to-date libraries and reducing compatibility headaches.

For C and C++ code, CMake version 3.28.6 managed the build process, with CMAKE_CXX_STANDARD set to 17 to align with language standards that have been widely adopted since 2017.

This carefully crafted environment provided a stable base for evaluating the outputs from chosen LLMs under consistent hardware and software conditions across several programming languages. This approach guarantees that any differences in performance metrics stem from the models themselves, not from changing environmental factors.

As mentioned in section 2.2 concerning existing evaluation frameworks, having a solid operating setup plays a key role in accurately judging not just functional correctness but also security risks embedded in AI-generated code.

Furthermore, keeping testing conditions consistent allows for clearer comparisons across various programming tasks, as noted in section 5.1. This consistency helps prevent biases that might arise from fluctuating experimental conditions. See reference [1].

Category	Task Count
Secure Coding (CWEs - All CWE identifiers grouped)	102
Data Structures and Algorithms	86
Parsing and Validation	54
Networking	48
Mathematics and Logic	26
Programming Systems and Utilities	18
Concurrency and Synchronization	6

Table 2: Distribution of tasks by tag categories after grouping related tags into broader functional areas. One task might fall into one or multiple categories. (source: reference [1])

4.2. Types of Programming Tasks Selected for Evaluation

This research assembled a variety of programming tasks to evaluate how well large language models (LLMs) generate code. The tasks covered different styles and difficulty levels to provide a thorough assessment of real coding challenges. Since LLMs perform unevenly across languages and problems, the selection highlighted these differences.

The study focused on Python, Java, C, and C++. These languages differ in important ways: Python uses

International Journal of Computer Science and Mobile Computing
A Monthly Journal of Computer Science and Information Technology

Zainab Abd-al Abbas Muhsen, Volume 15, Issue 7, July 2026, pg. 100-117
(An Open Accessible, ISI Indexed, Fully Refereed and Peer Reviewed Journal)
ISSN: 2320-088X
Impact Factor: 7.056

dynamic typing and automatic memory management, while C++ relies on static typing and manual memory handling. This variety helped reveal how LLMs handle language-specific syntax and security issues while writing functional code.

Tasks included solving math problems, designing algorithms, and manipulating data structures. LeetCode examples provided clear problems with known benchmarks, but custom challenges inspired by academic exercises encouraged originality and prevented simple memorization.

Security was a key concern in all assignments. Models were pushed to produce safe as well as correct code, addressing vulnerabilities tied to language features.

By combining straightforward logic problems with complex algorithmic tasks, this framework helps identify where LLMs excel or struggle. It confirms that performance varies with task difficulty and emphasizes the importance of human review during code evaluation. See references: [2] and [1].

5. Results and Analysis

5.1. Performance Across Different Programming Tasks

The results of AI-generated source code vary widely across different programming tasks, showing both the capabilities and shortcomings of various large language models (LLMs). Testing several LLMs on carefully chosen coding problems revealed that these models can produce effective solutions, but their success depends a lot on the task's complexity and type. For example, research shows that ChatGPT-4o shines when creating code that is both functionally correct and easy to read, especially in simpler cases like writing Arduino programs. This model often matches human-written code closely in style and structure. On the flip side, models such as Gemini 2.0 Flash may deliver faster-running code but tend to sacrifice clarity and ease of maintenance. Some generated solutions become complicated and require developers to dig deeper to understand and verify their functionality—particularly when the programming challenges grow more difficult. Human review remains essential because although LLMs usually provide working code, they often need expert checks to catch syntax errors or logical mistakes.

Beyond simply working correctly, several studies have examined how these models perform in terms of runtime speed and memory use across languages. While LLMs can handle a range of tasks—from straightforward algorithms to complex data handling—their effectiveness often drops as the problems get tougher. Difficult tasks frequently expose weaknesses in AI-generated code, especially when it comes to following best coding practices compared to easier problems. Another crucial aspect impacting results is how prompts are crafted when interacting with LLMs. Clear, well-designed prompts can significantly improve the accuracy and relevance of the responses. Conversely, vague or poorly framed queries may cause the models to generate less favorable or unexpected outputs. Therefore, successful use hinges on selecting suitable tasks and providing precise instructions to steer the model's code generation in the right direction. See references: [2], [8] and [9].

International Journal of Computer Science and Mobile Computing
A Monthly Journal of Computer Science and Information Technology

Zainab Abd-al Abbas Muhsen, Volume 15, Issue 7, July 2026, pg. 100-117
(An Open Accessible, ISI Indexed, Fully Refereed and Peer Reviewed Journal)
ISSN: 2320-088X
Impact Factor: 7.056

5.2. Comparative Analysis of Language Models' Outputs

Comparing code outputs from different language models reveals clear differences alongside some overlaps in their capabilities. One study looked at GitHub Copilot, Amazon CodeWhisperer, and OpenAI's GPT-3 across 164 programming tasks. Each model showed strong results, but GPT-3 edged ahead with a 93.3% success rate. Copilot followed closely at 91.5%, and CodeWhisperer at 90.2%. Still, all models stumbled over similar mistakes like syntax slips, wrong type handling, and missing imports. CodeWhisperer, in particular, struggled more with issues in list indexing and assertion statements, which led to additional invalid outputs.

It's not just about passing tests though—research also flagged the complexity of AI-produced code. While many solutions met functional requirements, they often came with moderate cyclomatic and cognitive complexity. This added complexity can be a headache for developers who face future maintenance or code refactoring.

Assessing the quality of generated code involved checking for defects and security flaws too. Multiple studies point out that large language models often churn out code with minimal bugs or duplication, sometimes rivaling human-created code. Yet, expert scrutiny remains vital. As AI-written code becomes more embedded in software development, vigilant review is still necessary to keep code reliable and aligned with standards.

Running benchmarks on these models under the same conditions can shed light on where each excels or falls short in real-world use. Such comparisons help not only to refine the models but also guide developers on how best to harness AI tools for writing source code. See references: [2] and [11].

6. Discussion of Findings

6.1. Implications for Software Development Practices

AI-generated code is changing how developers and companies approach software creation. Large language models like GPT-3 and GPT-4 can produce working code quickly, but relying on them without thorough checks poses risks. Human oversight remains vital to ensure code quality and security.

Developers face a verification bottleneck because AI generates code so fast. Tools like LLMSecEval evaluate not only whether code works but also if it contains security flaws. With 96% of developers doubting AI-generated code's reliability, skipping detailed reviews could threaten project success.

Testing LLM outputs in controlled environments is necessary, especially since these models handle simple tasks well but struggle with complex ones. This requires developers to review and refine AI-produced code carefully.

International Journal of Computer Science and Mobile Computing
A Monthly Journal of Computer Science and Information Technology

Zainab Abd-al Abbas Muhsen, Volume 15, Issue 7, July 2026, pg. 100-117
(An Open Accessible, ISI Indexed, Fully Refereed and Peer Reviewed Journal)
ISSN: 2320-088X
Impact Factor: 7.056

The impact extends beyond code correctness. Companies must foster a culture that prioritizes quality over speed. Implementing strong quality assurance suited for AI-assisted development is essential. A “Quality Playbook” can provide clear standards and processes tailored to projects using AI tools.
Collaboration between developers, testers, and quality engineers is increasingly important. Combining human insight with AI’s speed leads to better results. Cross-functional teamwork helps raise coding standards and maximizes AI benefits.
In short, managing AI-generated code requires layered checks and flexible teamwork. Emphasizing reliable engineering practices helps companies adopt these technologies while controlling risks. See references: [2], [22], [13] and [12].

6.2. Reliability and Security Concerns Identified in Generated Code

Concerns about the reliability and security of AI-generated code grow as more organizations use large language models in development. A major issue is detecting security vulnerabilities in the produced code. While LLMs often output syntactically correct code, these snippets can hide serious security flaws. Current evaluation tools like LLMSecEval and CWEval-Bench add security checks but still miss some threats.
The complexity of AI-generated code varies by task and model, causing inconsistent quality and security assessments across languages and situations. Some models perform well on simple projects but struggle with complex problems that require deeper understanding and close developer review.
LLMs can also introduce subtle bugs or "hallucinated" dependencies—imaginary packages alongside real code. These phantom elements increase technical debt and mislead developers into including unreliable parts in production.
Overreliance on automated tools creates a false sense of security. Because LLMs train on vast, unfiltered codebases, they may replicate existing security weaknesses. Research shows AI-generated code often copies these flaws, so thorough manual reviews remain essential before deployment.
The opaque nature of AI outputs makes tracking code errors or identifying the generating model difficult. This lack of traceability creates risks related to intellectual property and licensing violations. Improving reliability requires better tools and a culture that prioritizes secure coding. Encouraging cross-disciplinary teamwork ensures human judgment guides machine-generated code, not blindly follows it. See references: [10], [1] and [5].

International Journal of Computer Science and Mobile Computing
A Monthly Journal of Computer Science and Information Technology

Zainab Abd-al Abbas Muhsen, Volume 15, Issue 7, July 2026, pg. 100-117
(An Open Accessible, ISI Indexed, Fully Refereed and Peer Reviewed Journal)
ISSN: 2320-088X
Impact Factor: 7.056

7. Recommendations for Future Research and Practice

7.1. Suggested Improvements to LLMs for Enhanced Reliability and Security

Improving the reliability and security of AI-generated code requires a comprehensive approach targeting both model training and output evaluation. Updating training datasets with vetted open-source projects that follow current security and coding standards helps models learn from quality examples and avoid outdated vulnerabilities. Filtering out code with known security issues is also vital. Collaborating with open-source maintainers helps identify and exclude libraries or snippets with documented weaknesses from future training.

Transparency plays a key role. AI platforms should provide logs showing how specific code was generated, including its training sources. This allows developers to trace bugs and understand model limits. Providers must also share any known blind spots, such as trouble handling certain vulnerability types, so users know when to apply extra caution.

Creating clear feedback channels encourages community involvement. Programs like bug bounties and vulnerability disclosures help report insecure AI-generated code and strengthen models over time. Involving open-source maintainers in testing new releases fosters community-led trials that highlight real-world issues and improvements.

As models advance, embedding evaluation methods that assess security reasoning alongside functionality becomes essential for spotting vulnerabilities. Finally, addressing earlier research gaps on human oversight remains important. Collaborative review systems ensure developers carefully check AI-generated code, keeping security a priority throughout software development. See references: [1], [5] and [17].

Framework	Models	Code Analysis Tool	Dataset	Language(s)
SecCode (82,)	GPT-3.5 DeepSeek-Coder	CodeQL	LLMSecEval (211,) SecurityEval (210,) Holmes (86,)	C Python
SALLM (79,)	CodeGen models StarCoder GPT models	CodeQL	LLMSecEval (211,) SecurityEval (210,) SALLM (79,)	C Python
LLMSecGuard (78,)	Llama2	Semgrep Weggli	CyberSecEval (212,)	Java

Table 3: Recent Security Code Generation frameworks (source: reference [5])

International Journal of Computer Science and Mobile Computing
A Monthly Journal of Computer Science and Information Technology

Zainab Abd-al Abbas Muhsen, Volume 15, Issue 7, July 2026, pg. 100-117
(An Open Accessible, ISI Indexed, Fully Refereed and Peer Reviewed Journal)
ISSN: 2320-088X
Impact Factor: 7.056

7.2. Best Practices for Developers Using AI-Generated Code

As developers rely more on AI-generated code, following best practices remains essential for quality and security. Treating AI tools as partners helps boost productivity but requires careful human oversight. This approach aligns with the need for thorough human checks to address quality issues in AI outputs.

Prioritizing code readability in AI results allows for easier debugging and better understanding of the logic. Clear code helps identify problems early, preventing larger issues in complex systems.

Consistent reviews of AI-generated code are necessary for functionality and security, using evaluation methods and tools like SonarQube and CodeQL. These tools catch vulnerabilities before deployment and strengthen review processes.

Avoiding blind copy-pasting encourages active engagement with AI suggestions. This practice fosters learning, especially for junior developers who might rely too much on AI without sufficient guidance.

Documenting the origins of AI-generated code supports license compliance and avoids legal risks related to open-source contributions. Automated license scanning and careful record-keeping help maintain transparency.

Regular updates to AI training datasets keep models aligned with current coding standards and security practices. Collaboration with open-source communities provides valuable feedback that improves AI accuracy and reliability.

Finally, fostering a culture of ongoing feedback enhances coding skills and improves AI tools. Encouraging developers to report insecure outputs strengthens software resilience. Following these practices helps manage AI coding challenges and positively influences software development. See references: [20], [16] and [17].

8. Conclusion and Future Work Directions

8.1. Summary of Key Findings

Research into AI-generated source code shows both promising opportunities and challenges when using large language models (LLMs) in software development. Models like GPT-3 and GPT-4 perform well on typical coding exercises but often struggle with more complex tasks. This makes careful human review crucial to ensure accuracy and reliability.

Repeated tests with tools such as ChatGPT find that code quality remains steady and comparable to human output, but occasional bugs still appear. This highlights the need for thorough testing before AI-generated code enters production.

Evaluation methods vary in how they measure code quality and security. It is important to use comprehensive frameworks that assess both whether the code functions and its vulnerability to security flaws. Benchmarks like LLMSecEval provide useful insights but require improvement to detect weaknesses more effectively.

International Journal of Computer Science and Mobile Computing
A Monthly Journal of Computer Science and Information Technology

Zainab Abd-al Abbas Muhsen, Volume 15, Issue 7, July 2026, pg. 100-117
(An Open Accessible, ISI Indexed, Fully Refereed and Peer Reviewed Journal)
ISSN: 2320-088X
Impact Factor: 7.056

Collaboration among experts from different fields helps maximize the benefits of LLMs. Developers and researchers must work together to refine coding practices, integrate AI tools effectively, and address trust and safety concerns.

Providing LLMs with clear, detailed instructions reduces errors and improves code quality. Additionally, keeping careful records on licensing remains essential as developers harness these powerful tools to boost productivity.

Overall, the findings suggest cautious optimism for AI-generated code in software engineering, encouraging thoughtful use alongside careful safeguards. See references: [2] and [11].

8.2. Potential Areas for Further Investigation

To deepen insights into AI-generated source code, several promising research directions come to light. One key focus lies in crafting specialized datasets that suit particular programming tasks like code summarization, disassembly, and comment generation. Generic datasets often miss the finer details these tasks demand. Creating well-curated, task-specific data collections would boost large language models (LLMs) training quality and help them tackle complex coding challenges more effectively.

Another exciting path involves designing model architectures that perform consistently across various programming languages. Current LLMs tend to shine in some languages but falter in others. Exploring hybrid models that draw on the strengths of multiple languages could yield more versatile and reliable tools. This work might include fine-tuning existing models to harness modern language features and best practices, resulting in safer and more productive code outputs.

Security also stands out as an area that deserves closer attention. Embedding security assessments directly into code generation could help catch vulnerabilities early, reducing risks before code reaches production. Alongside this, evaluating models based on reliability and maintainability in different programming environments can highlight traits that support the long-term health of software.

Equally important is examining how developers engage with AI-generated code. Understanding the ways programmers interpret, modify, and trust these outputs could shape future model improvements and user experience. Building developers' confidence in AI-driven code creation is essential for wider adoption.

Lastly, tracking LLM performance over time through longitudinal studies would shed light on their evolution as new training data and updates arrive. This research can guide strategies for developers aiming to harness AI in fast-changing software development settings.

Altogether, these avenues offer rich possibilities to sharpen AI-generated code capabilities and ensure they serve developers effectively now and in the future. See references: [1] and [5].

International Journal of Computer Science and Mobile Computing
A Monthly Journal of Computer Science and Information Technology

Zainab Abd-al Abbas Muhsen, Volume 15, Issue 7, July 2026, pg. 100-117
(An Open Accessible, ISI Indexed, Fully Refereed and Peer Reviewed Journal)
ISSN: 2320-088X
Impact Factor: 7.056

References

[1]. Mohammed F. Kharma, S. Choi, M. Alkhanafseh and D. Mohaisen. "Security and Quality in LLM-Generated Code: A Multi-Language, Multi-Model Analysis". Mar 2026. [Online]. Available: https://arxiv.org/html/2502.01853v2

[2]. D. Tosi. "Studying the Quality of Source Code Generated by Different AI Generative Engines: An Empirical Evaluation". May 2024. [Online]. Available: https://www.mdpi.com/1999-5903/16/6/188

[3]. Sardar K. Jabr and Qusay I. Sarhan. "A Systematic Survey on Large Language Models for Code Generation". Aug 2025. [Online]. Available: https://aro.koyauniversity.org/index.php/aro/article/view/2159

[4]. H. Suh, M. Tafreshipour, J. Li, A. Bhattiprolu and I. Ahmed. "An Empirical Study on Automatically Detecting AI-Generated Source Code: How Far are We?". (accessed Jun 21, 2026). [Online]. Available: https://ieeexplore.ieee.org/document/11029804/

[5]. H. Jelodar, M. Meymani and R. Razavi-Far. "Large Language Models (LLMs) for Source Code Analysis: applications, models and datasets". Mar 2025. [Online]. Available: https://arxiv.org/html/2503.17502v1

[6]. H. Suh, M. Tafreshipour, J. Li, A. Bhattiprolu and I. Ahmed. "An Empirical Study on Automatically Detecting AI-Generated Source Code: How Far Are We?". Oct 2025. [Online]. Available: https://dl.acm.org/doi/10.1109/ICSE55347.2025.00064

[7]. H. Suh, M. Tafreshipour, J. Li, A. Bhattiprolu and I. Ahmed. "An Empirical Study on Automatically Detecting AI-Generated Source Code: How Far Are We?". Jun 2024. [Online]. Available: https://arxiv.org/abs/2411.04299

[8]. Sardar K. Jabr and Qusay I. Sarhan. "Evaluating Large Language Models for Arduino Code Generation". (accessed Jun 21, 2026). [Online]. Available: https://aro.koyauniversity.org/index.php/aro/article/view/2344

[9]. V. Geruslu, Z. Aliyeva and E. Tüzün. "Factors Influencing the Quality of AI-Generated Code: A Synthesis of Empirical Evidence". Mar 2026. [Online]. Available: https://arxiv.org/abs/2603.25146

[10]. T. Shaukat. "Study finds shared strengths and common challenges across popular LLMs". Aug 2025. [Online]. Available: https://www.sonarsource.com/company/press-releases/the-coding-personalities-of-leading-llms/

[11]. A. Clark, D. Igbokwe, S. Ross and Minhaz F. Zibran. "A Quantitative Analysis of Quality and Consistency in AI-generated Code". (accessed Jun 21, 2026). [Online]. Available: https://ieeexplore.ieee.org/document/10608315/

[12]. A. Chatterjee. "State of Code Developer Survey report: The current reality of AI coding". (accessed Jun 21, 2026). [Online]. Available: https://www.sonarsource.com/blog/state-of-code-developer-survey-report-the-current-reality-of-ai-coding

[13]. H. Guo, S. Cheng, K. Zhang, G. Shen and X. Zhang. "CodeMirage: A Multi-Lingual Benchmark for Detecting AI-Generated and Paraphrased Source Code from Production-Level LLMs". May 2025. [Online]. Available: https://arxiv.org/abs/2506.11059

[14]. D. Orel, D. Azizov, I. Paul, Y. Wang, I. Gurevych and P. Nakov. "AICD Bench: A Challenging Benchmark for AI - Generated Code Detection". Mar 2026. [Online]. Available: https://aclanthology.org/2026.eacl-long.325/

[15]. "Is source code generation an anti-pattern?". (accessed Jun 21, 2026). [Online]. Available: https://softwareengineering.stackexchange.com/questions/361458/is-source-code-generation-an-anti-pattern

[16]. M. Shust. "Debunking AI code myths: collaboration not automation". (accessed Jun 21, 2026). [Online]. Available: https://www.linkedin.com/posts/markshust_most-developers-think-ai-generated-code-is-activity-7371512967877804033-BsnZ

[17]. H. Sidhpurwala, https://www.redhat.com/en/authors/huzaifa-sidhpurwala-0 and 562031. "When bots commit: AI-generated code in open source projects". Apr 2001. [Online]. Available: https://www.redhat.com/en/blog/when-bots-commit-ai-generated-code-open-source-projects

[18]. R. Satyanarayana. "The IP Dilemma: Who Owns Source Code from GenAI Code Assistants?". Jul 2025. [Online]. Available: https://www.linkedin.com/pulse/ip-dilemma-who-owns-source-code-from-genai-assistants-satyanarayana-2uqgf

[19]. I. A. Fahad and Mridha Md. Nafis Fuad. "Can we trust the source? A systematic review of watermarking and attribution for AI-generated code". Jan 2026. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0950584926001461

International Journal of Computer Science and Mobile Computing
A Monthly Journal of Computer Science and Information Technology

Zainab Abd-al Abbas Muhsen, Volume 15, Issue 7, July 2026, pg. 100-117
(An Open Accessible, ISI Indexed, Fully Refereed and Peer Reviewed Journal)
ISSN: 2320-088X
Impact Factor: 7.056

[20]. A. Stukalov. "Yesterday I cleaned up about 60% of code generated by an agent that prefers to stay unnamed :).". (accessed Jun 21, 2026). [Online]. Available: https://www.linkedin.com/posts/alekseystukalov_yesterday-i-cleaned-up-about-60-of-code-activity-7388384145430822912-G433

[21]. I. Guelman, A. G. Leal, L. Xavier and M. T. Valente. "On the Quality of AI-Generated Source Code Comments: A Comprehensive Evaluation". Aug 2024. [Online]. Available: https://arxiv.org/abs/2408.14007

[22]. A. Stellman. "AI Is Writing Our Code Faster Than We Can Verify It". Apr 2026. [Online]. Available: https://oreillyradar.substack.com/p/ai-is-writing-our-code-faster-than

