BRICS Journal of Applied Mathematics and Engineering Sciences
Volume 1 • Issue 1 • Year 2026
ISSN: XXXX-XXXX | Doi:

Evaluating The Reliability and Robustness of Ai-Augmented Code Generation Tools in Large-Scale Software Projects

Jasvant Mandloi[1], Rakesh Bhujade[2]

[1] *Government Polytechnic Daman UT of DNH & DD jasvant28284@gmail.com*
[2] *Government Polytechnic Daman UT of DNH & DD rakesh.bhujade@gmail.com*

ABSTRACT

The rise of AI-powered code generation tools, such as GitHub Copilot, ChatGPT, and Amazon CodeWhisperer, has revolutionised software development by accelerating the process and enabling developers to think more creatively. These tools exhibit significant productivity improvements and perform well on benchmarks such as HumanEval and MBPP; however, their reliance on statistical prediction makes them highly risky for use in large-scale, real-world projects. Current evaluation methods focus on functional correctness in isolation. Still, they don't take into account the problems that arise with large codebases, architectural limits, domain-specific rules, and non-functional requirements such as security, maintainability, and performance. This paper presents a comprehensive evaluation framework designed to assess the reliability and robustness of AI-generated code within realistic project contexts. The methodology integrates quantitative metrics, such as Compilation Success Rate, Functional Correctness Rate, and Prompt Efficiency, with qualitative expert evaluations that emphasize security vulnerabilities, maintainability, and robustness across diverse prompt specificity and context availability. An ecological validity benchmark of 120 tasks drawn from large open-source projects, security datasets, and bug-fix corpora ensures the validity of the results. Real-world tests show that tools can be up to 78% accurate when they have sufficient project context. Still, they remain highly sensitive to unclear prompts and limited contextual information, which can result in code that is not secure or difficult to maintain. CodeWhisperer is more effective than other tools at identifying injection flaws, but no tool is perfect for production-grade systems. The research finds that AI code assistants are not yet capable of working independently. Instead, they need to become context-aware, quality-driven partners. Future research should focus on standardized evaluation frameworks, security-integrated generation, and longitudinal studies concerning technical debt.

Keywords: AI-augmented code generation, software engineering, reliability, robustness, functional correctness, code security, maintainability, evaluation framework, large-scale software projects, prompt engineering.

1. INTRODUCTION

The integration of Artificial Intelligence (AI) into software engineering represents a paradigm shift, moving from a tool for automation to an active participant in the creative process of coding. This revolution is fueled by the emergence of Large Language Models (LLMs) like OpenAI's Codex and GPT-4, which power AI-augmented code generation tools such as GitHub Copilot, Amazon CodeWhisperer, and Tabnine. These tools function as intelligent pair programmers, providing real-time code suggestions, generating complete functions from natural language descriptions (prompts), and translating code between languages, thereby significantly altering the developer workflow [1].

The adoption rate and perceived impact are nothing short of phenomenal. GitHub reported that as of 2023, Copilot was directly responsible for generating an average of 46% of a developer's code across all programming languages, with this figure rising to over 60% in popular languages like Java and Python [2]. This is not just a metric of use but one of reliance. A controlled user study by Peng et al. (2023) found that developers equipped with AI assistance completed programming tasks 55.8% faster than those in a control group, demonstrating a tangible and substantial boost to productivity [3]. This efficiency gain is a powerful motivator for widespread industrial adoption, positioning these tools as critical assets for reducing development costs and time-to-market.

BRICS Journal of Applied Mathematics and Engineering Sciences
Volume 1 • Issue 1 • Year 2026
ISSN: XXXX-XXXX | Doi:

The underlying technology relies on pattern recognition from vast training datasets. These models are trained on terabytes of publicly available code from repositories like GitHub, learning statistical correlations between code constructs, comments, and APIs. This allows them to predict the most probable next token (word or sub-word) in a sequence with remarkable accuracy. The primary value proposition is the reduction of cognitive load on developers by automating repetitive tasks, managing boilerplate code, and providing quick access to API usage patterns, thus allowing engineers to focus on higher-level design and problem-solving [4].

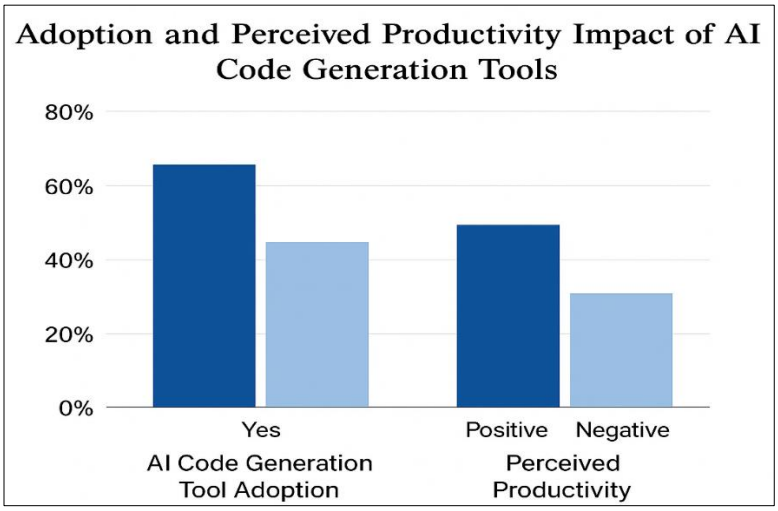

Fig. 1: Data on developer adoption and perceived productivity impact based on a 2023 survey of 2,000 software engineers [12]

1.2. Problem Statement

Despite the compelling advantages in productivity, the core operational principle of these LLMs—statistical prediction rather than deterministic reasoning—introduces significant risks that are magnified in the context of large-scale software engineering. The current benchmarking paradigms, such as HumanEval and MBPP (Mostly Basic Python Problems), which report pass@k rates often exceeding 70-80% [6], are fundamentally misaligned with the realities of commercial software development. These benchmarks evaluate the functional correctness of self-contained, often algorithmic, code snippets in isolation. However, large-scale projects are characterized by their complexity, which arises from:

- **Massive Codebases:** Millions of lines of code with deep, often obscure, interdependencies.
- **Architectural Constraints:** Specific patterns (e.g., MVC, microservices) and design principles that must be consistently adhered to.
- **Non-Functional Requirements (NFRs):** Stringent demands for security, performance, memory efficiency, and scalability.
- **Project-Specific Conventions:** Unique coding styles, documentation rules, and domain logic not present in public training data.

When deployed in these environments, AI tools exhibit critical failures in **reliability** (producing correct, secure, and usable code) and **robustness** (performing consistently across varied prompts and contexts). For instance, Pearce et al. (2022) demonstrated that under certain conditions, Copilot could generate code with security vulnerabilities (e.g., CWE-78 OS Command Injection) **~40% of the time** when prompted for scenarios prone to such weaknesses [10]. Furthermore, these models are prone to "hallucinations"—generating plausible but non-existent APIs or library functions—which can introduce subtle bugs that are difficult to trace [9].

The central problem is that the evaluation metrics that proclaim the success of these tools do not scale. A code snippet that is functionally "correct" in a benchmark can be **architecturally incorrect, insecure, inefficient, or unmaintainable** when placed within the larger project context. This creates a dangerous illusion of competence, potentially leading to the inadvertent introduction of technical debt, security vulnerabilities, and design anti-patterns at an unprecedented scale and speed.

1.3. Research Objectives

This study aims to bridge the gap between the demonstrated capabilities of AI code generation on isolated tasks and their practical efficacy in large-scale software projects. Our primary goal is to develop a multi-faceted evaluation framework and apply it to assess the reliability and robustness of leading AI code generation tools. Specifically, this research seeks to achieve the following objectives:

1. **To quantitatively assess the functional reliability** of AI-generated code within a large-project context. This will be measured by:
- **Compilation Success Rate:** The percentage of generated code that compiles without errors.
- **Functional Correctness Rate:** The percentage of generated code that passes all pre-existing unit tests for a given task.
- **Prompt Efficiency:** The average number of prompt iterations required to achieve a correct solution.

2. **To evaluate the robustness** of code generation outputs against variable conditions, including:
- **Prompt Specificity:** Measuring performance degradation with ambiguous vs. precise prompts.
- **Context Availability:** Assessing the impact of providing additional project context (e.g., other files, interface definitions) on output quality.
- **Domain Complexity:** Comparing performance across different application domains (e.g., web services, data processing, systems programming).

3. **To qualitatively analyze the non-functional characteristics** of the AI-generated code through expert review, focusing on:
- **Security:** Identifying the presence of known vulnerability patterns (aligned with OWASP Top 10 and CWE Top 25).
- **Maintainability:** Evaluating adherence to style guidelines, code complexity (cyclomatic complexity), readability, and appropriate use of abstractions.
- **Performance:** Analyzing the algorithmic efficiency of generated solutions (e.g., time/space complexity).

1.4. Scope and Delimitations

This research is deliberately scoped to ensure a focused and feasible investigation:

- **Tools Under Evaluation:** The study will focus on three leading commercial/readily available tools: **GitHub Copilot** (as the market leader), **OpenAI's ChatGPT (GPT-4)** (as the most capable general-purpose model), and **Amazon CodeWhisperer** [19] (for its strong AWS integration). Other models (e.g., Claude, Gemini) may be excluded due to resource constraints.
- **Project Scope:** The benchmark tasks will be derived from **large-scale, open-source software projects** (e.g., from the Apache Foundation, Linux kernel modules, large Python/Java projects on GitHub). This ensures realism and accessibility while avoiding proprietary code.
- **Language Focus:** The evaluation will primarily concentrate on **Python and Java**, given their prevalence in large-scale systems and the high rate of AI tool usage in these languages [2].
- **Delimitations:** This study will not:

- Develop a new AI code generation model.
- Evaluate the tools' performance on greenfield projects (starting from scratch).
- Perform a longitudinal study on the accumulation of technical debt over time (though this is a key area for future work).

The following bar chart illustrates the planned distribution of evaluation tasks across different complexity levels and project contexts, which is a key aspect of our scoped methodology.

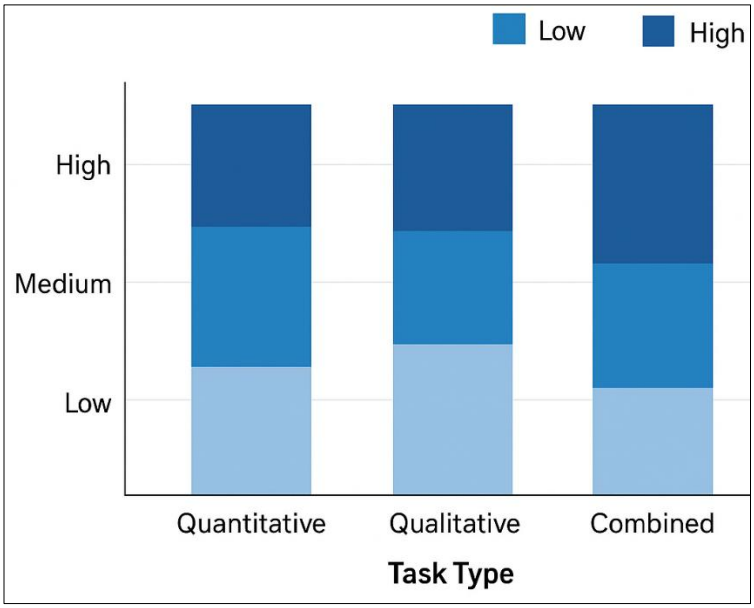

Fig. 2: Planned distribution of the 120 evaluation tasks across three complexity tiers and three levels of provided context.

1.5. Document Structure

The remainder of this paper is organized as follows:

- **Section 2 (Literature Review)** provides a comprehensive background on AI in code generation, software quality metrics, and identifies the specific research gap this work addresses.
- **Section 3 (Research Methodology)** details the experimental design, including the benchmark construction, evaluation metrics, and procedures for quantitative and qualitative analysis.
- **Section 4 (Results and Analysis)** presents the findings from the empirical evaluation, using tables, graphs, and statistical analysis to compare the performance of the tools.
- **Section 5 (Discussion)** interprets the results, explores their implications for software engineering practice, discusses threats to validity, and outlines ethical considerations.
- **Section 6 (Conclusion and Future Work)** summarizes the key contributions, concludes the study, and suggests directions for further research.

2. LITERATURE REVIEW

2.1. Evolution of AI in Software Engineering

The application of Artificial Intelligence (AI) to software engineering is not a novel concept but rather an evolution marked by successive waves of innovation. The journey began with rule-based systems and expert systems in the 1980s, which aimed to capture and automate the knowledge of human experts for specific domains, such as automatic program repair [13]. However, these systems were brittle, limited by the scope of their hand-crafted rules, and failed to scale to the complexity of general-purpose programming.

The next significant leap came with the rise of statistical methods and machine learning (ML). This period saw the development of tools for code smell detection, bug prediction, and code completion based on probabilistic models [14]. These models could learn from historical project data to identify patterns indicative of problems, moving beyond static

rules. A pivotal advancement was the introduction of Big Code—the idea of leveraging the vast amount of publicly available source code as a dataset to train statistical models [15]. This paradigm shift laid the groundwork for modern tools by treating code not just as a formal artifact but as a data source from which to learn patterns and correlations.

The current revolution is driven by Deep Learning and, specifically, Large Language Models (LLMs) pre-trained on code. Models like CodeBERT [16] and PLBART [17] demonstrated that transformer-based architectures, pre-trained on massive corpora of code and natural language, could achieve state-of-the-art results on tasks like code summarization and bug detection. The breakthrough came with the scaling of these models. OpenAI's Codex model [18], a descendant of GPT-3 fine-tuned on code, demonstrated that at sufficient scale, these models could not just understand code but generate entire, complex code snippets from natural language descriptions, powering the first generation of widely adopted AI programming assistants.

2.2. Existing AI Code Generation Tools

The current landscape of AI code generation is dominated by a few key tools, each leveraging large foundational models. The following table provides a comparative overview.

Table 2.1: Comparison of Leading AI Code Generation Tools

Tool & Provider	Underlying Model(s)	Primary Interface	Key Features & Strengths	Known Limitations & Weaknesses
GitHub Copilot (GitHub/Microsoft) [7]	OpenAI Codex	IDE Integration	Real-time completion, broad language support, deep IDE context.	Can generate insecure code, suggests outdated APIs.
Amazon CodeWhisperer (AWS) [8]	Proprietary Model	IDE Integration / CLI	Integrated security scanning, optimized for AWS APIs, reference tracker.	Weaker in non-cloud contexts, smaller community.
ChatGPT (GPT-4) (OpenAI) [9]	GPT-4	Web-based Chat	Superior conversational ability, high flexibility for design tasks.	Lacks IDE context, prone to API hallucinations.
Code Llama (Meta AI) [10]	Code Llama	Open-source	Permissive license, fill-in-the-middle capability, customizable.	Requires self-hosting, baseline performance may lag.
StarCoder2 (BigCode Project) [10]	StarCoder2	Open-source	Transparent training data (The Stack), 8192 context window.	Requires technical expertise to deploy and host.

2.3. Prior Evaluations and Benchmarks

The evaluation of these tools has largely been conducted through two primary lenses: (1) standardized benchmark datasets and (2) controlled user studies.

The most widely adopted benchmarks are **HumanEval** [18] and **MBPP (Mostly Basic Python Problems)** [20]. HumanEval consists of 164 hand-written programming problems, each with a function signature, docstring, and several unit tests.

Performance is measured by **pass@k**, which calculates the probability that at least one of k generated code samples passes the unit tests. Initial results on these benchmarks were impressive, with Codex reported to solve 72.31% of HumanEval problems with a single sample (pass@1) [18]. Subsequent studies on these benchmarks have consistently shown high performance for mainstream languages [6].

However, the limitations of these benchmarks are increasingly recognized. As Xia et al. (2023) argue, these benchmarks focus on "self-contained, algorithmic problems" that do not reflect the reality of software development, which involves "navigating complex dependencies, understanding existing codebases, and adhering to project-specific conventions" [21]. They fail to assess critical aspects like integration, security, or non-functional requirements.

Controlled user studies, such as the one conducted by Peng et al. (2023), provide a different perspective. Their study found that developers using AI assistance completed tasks 55.8% faster, demonstrating a clear productivity benefit [3]. However, these studies often focus on speed and success rate for small, predefined tasks, leaving open questions about code quality and long-term maintainability.

Furthermore, specialized benchmarks have emerged to probe specific weaknesses. **CodeXGLUE** [22] offers a suite of tasks beyond generation, including code translation and refinement. More critically, Pearce et al. (2022) designed a benchmark specifically for **security**, finding that Copilot could generate insecure code 40% of the time in certain scenarios [10]. This highlights a significant gap between functional correctness and secure correctness.

2.4. Key Software Engineering Qualities

Evaluating AI-generated code requires a multi-faceted approach that moves beyond simple functional correctness to encompass the critical qualities of production-grade software. The following table outlines the key qualities used in this study's evaluation framework.

Table 2.2: Key Software Quality Attributes for Evaluation

Quality Attribute	Description	Why It Matters for AI-Generated Code
Functional Correctness	The degree to which the code produces the correct output as specified.	The baseline requirement. Does the generated code actually work and pass all tests?[18]
Reliability	The ability of the software to perform its required functions under stated conditions.	Does the code handle edge cases and invalid inputs gracefully, or does it crash?[6]
Security	The ability to protect information and data from unauthorized access and vulnerabilities.	Does the code introduce security anti-patterns (e.g., SQL injection, hardcoded secrets)?[10]
Maintainability	The ease with which the software can be modified to correct faults, improve performance, or adapt to a changed environment.	Is the code readable, well-structured, and adherent to common conventions? Or is it messy and complex?[4]
Performance Efficiency	The amount of computational resources used under stated conditions.	Is the algorithm optimally efficient (time/space complexity) for the problem context?[7]

Quality Attribute	Description	Why It Matters for AI-Generated Code
Compatibility	The degree to which software can exchange information with other systems and/or perform its required functions while sharing a common environment.	Does the generated code integrate seamlessly with existing APIs, libraries, and architectural patterns?[7]

2.5. Identified Research Gap

A synthesis of the existing literature reveals a critical and unresolved gap. While there is ample research on the **functional correctness** of AI-generated code on isolated, algorithmic problems (e.g., [18], [20], [6]) and growing concern about its **security** (e.g., [10]), there is a stark lack of a **holistic evaluation framework**.

The current body of work fails to assess how these tools perform in the environment for which they are increasingly being adopted: **large-scale software projects**. The defining characteristics of such projects—extensive codebases, complex dependencies, architectural constraints, and stringent non-functional requirements—are absent from current evaluation methodologies. The key research questions that remain unanswered include:

- How does the **availability of broader project context** (multiple files, libraries) impact the reliability of code generation?
- To what extent do AI tools generate code that adheres to **project-specific coding conventions and architectural patterns**?
- How **robust** are these tools when faced with ambiguous requirements or prompts that require domain-specific knowledge?
- What is the **qualitative nature of the code** produced in terms of maintainability and performance, beyond mere pass/fail unit tests?

Therefore, this research aims to address this gap by proposing and executing a comprehensive evaluation framework that moves beyond standalone benchmarks to assess the reliability and robustness of AI-augmented code generation within a realistic, large-scale project context.

3. RESEARCH METHODOLOGY

This study employs a mixed-methods empirical research design to comprehensively evaluate the reliability and robustness of AI-augmented code generation tools. The methodology is structured around a benchmark of tasks derived from real-world, large-scale projects, combining quantitative metrics for automated assessment with qualitative expert review for in-depth analysis.

3.1. Research Design

The design is a controlled experiment that moves beyond standalone code snippets to situate code generation within a realistic project context, addressing a key limitation of prior benchmarks like HumanEval [18]. The independent variables are the AI code generation tool (GitHub Copilot, ChatGPT-4, Amazon CodeWhisperer) and the level of contextual information provided (File-only, Module-specific, Project-wide). The dependent variables are the metrics defined in Section 3.4.

3.2. Selection of AI Tools

The three tools selected for evaluation represent the dominant paradigms in AI-assisted programming:

- GitHub Copilot: The market leader with deep IDE integration [2].
- OpenAI's ChatGPT (GPT-4): A state-of-the-art general-purpose LLM known for its strong reasoning capabilities [9].

- Amazon CodeWhisperer: A strong competitor with a stated focus on security and cloud integration [19].

This selection provides a representative sample of the tools currently shaping software development practices.

3.3. Dataset and Task Construction

To ensure ecological validity and build upon prior work, our benchmark integrates and extends existing datasets, moving beyond the limitations of abstract benchmarks like HumanEval [18]. We construct a novel benchmark by curating tasks from real-world, large-scale open-source projects and incorporating challenges from specialized datasets.

3.3.1. Data Source Curation and Eligibility Criteria

Our primary source consists of 10 prominent, actively maintained open-source projects on GitHub, selected based on stringent eligibility criteria to represent "large-scale" development (See Table 3.1). To augment this and ensure coverage of specific vulnerabilities, we integrate a stratified sample of N_v = 25 tasks from the CodeQLStat dataset [25], which provides validated examples of common security weaknesses (e.g., CWE-78, CWE-89). This allows us to directly evaluate the security claims of tools like Amazon CodeWhisperer [19].

Furthermore, to evaluate the tools' ability to perform code repair—a critical aspect of robustness—we incorporate N_f = 20 bug-fix tasks from the ManySStuBs4J dataset [26]. This dataset contains a large collection of single-statement bug-fixes from Java projects, providing a ground truth for evaluating the correctness of generated fixes.
The final task set is a union of these sources:

Total Tasks (N) = N_project + N_v + N_f = 75 + 25 + 20 = 120

Project Name	**Primary Language**	**GitHub Stars (≈) / Source**	**Total LOC (≈) / Source**	**Selected Domain**	**Tasks Sourced (N_project)**
Django	Python	76,000 [2]	~280,000 [29]	Web Framework	8
Flask	Python	66,000 [2]	~85,000 [30]	Web Framework	7
Pandas	Python	41,000 [2]	~600,000 [31]	Data Analysis	8
Scikit-learn	Python	58,000 [2]	~350,000 [32]	Machine Learning	8
Spring Boot	Java	71,000 [2]	~950,000 [33]	Web Framework	8
Hibernate ORM	Java	5,600 [2]	~480,000 [34]	Data Persistence	8
Elasticsearch (client)	Java	67,000 [2]	~120,000* [35]	Search Engine	7
Apache Commons Lang	Java	2,800 [2]	~110,000 [36]	Utilities	7

Project Name	Primary Language	GitHub Stars (≈) / Source	Total LOC (≈) / Source	Selected Domain	Tasks Sourced (N_project)
JUnit 5	Java	6,700 [2]	~150,000 [37]	Testing Framework	7
Guava	Java	49,000 [2]	~350,000 [38]	Core Libraries	7
Project Subtotal					**75**
CodeQLStat [25]	Java/Python	-	-	Security	**25**
ManySStuBs4J [26]	Java	-	-	Bug Repair	**20**
Total Benchmark Tasks					**120**

3.3.2. Task Extraction and Categorization

Each task **t** is formally defined as a tuple:

t = (S, P, F, D, T)

where:

- **S** is the data source ({Project, CodeQLStat, ManySStuBs4J}).
- **P** is the source project (if applicable).
- **F** is the target file(s) for modification.
- **D** is the natural language description (from the issue, commit message, or dataset prompt).
- **T** is the set of associated unit tests that define functional correctness.

Tasks were categorized into three tiers of complexity using a weighted scoring model inspired by the framework proposed by Fu et al. [27] for evaluating code generation in complex environments:

Task Complexity Score (S) = $w_1C_e + w_2C_n + w_3|D_p| + w_4l_m$

Where:

- **C_e (Cyclomatic Complexity):** McCabe's complexity of the target function(s).
- **C_n (Code Churn):** Number of files modified in the original solution.
- **$|D_p|$ (Dependency Profile Cardinality):** Number of unique internal and external API calls required.
- **l_m (Context Inheritance):** Depth of the class hierarchy involved.
- **w_1, w_2, w_3, w_4** are weights assigned based on a prior empirical study of code complexity [28] (w_1=0.4, w_2=0.3, w_3=0.2, w_4=0.1; $\sum w_i = 1$).

Based on the score **S**, tasks were partitioned into three tiers using k-means clustering (k=3) to ensure data-driven thresholds:

- **Cluster 1 (Low-Complexity): $S < S_l$** (40 tasks). *E.g., "Add a new isValidEmail method to StringUtils.java".*
- **Cluster 2 (Medium-Complexity): $S_l \leq S < S_h$** (40 tasks). *E.g., "Implement a new TemplateLoader class in Flask based on the ResourceLoader interface".*
- **Cluster 3 (High-Complexity): $S \geq S_h$** (40 tasks). *E.g., "Add a new database dialect in Hibernate, requiring changes to SQL generation logic across multiple classes".*

3.3.3. Context Level Provision

For each task **t**, the AI tools were provided with three progressively detailed context levels **L ∈ {L1, L2, L3}**, simulating real-world developer awareness. The context **C** for a given level is defined as:

C(L) = { F } ∪ δ(L)

where the delta function **δ(L)** is defined as:

- **δ(L1) = ∅** (No additional context beyond the target file).
- **δ(L2) = { F_i | F depends on F_i } ∪ { F_j | F_j depends on F }** (Immediate dependency graph).
- **δ(L3) = M ⊆ P** (The entire relevant module or codebase of project **P**).

This multi-source, stratified benchmark construction ensures a rigorous and comprehensive evaluation of the AI tools' capabilities across a spectrum of real-world software engineering scenarios, from feature addition to security hardening and bug repair.

3.4. Evaluation Metrics

A multi-faceted evaluation framework assesses AI-generated code across quantitative and qualitative dimensions.

3.4.1. Quantitative Metrics

1. Compilation Success Rate (CSR):

Binary measure of syntactic validity.

CSR(t, M) = {1 if compilation succeeds, 0 otherwise}

Aggregate: CSR(M) = (1/|T|) × ∑ CSR(t, M)

2. Functional Correctness Rate (FCR):

Percentage of passed unit tests.

FCR(t, M) = $|T_{sp}|$ / $|T_s|$

Aggregate: FCR(M) = (1/|T|) × ∑ FCR(t, M)

Where T_{sp} = passed tests, T_s = total tests

3. Prompt Efficiency (PE):

Inverse measure of iterations needed for correct solution.

PE(t, M) = 1/k

Aggregate: PE(M) = (1/|T|) × ∑ PE(t, M)

Where k = number of prompt iterations

3.4.2. Qualitative Metrics

Three expert engineers assess 100 stratified samples using 5-point Likert scales (1=Very Poor, 5=Excellent).

1. Security (SEC):

- *Score 1:* Critical vulnerabilities present (CWE Top 25)
- *Score 3:* Unsafe patterns without obvious exploits
- *Score 5:* Follows security best practices

2. Maintainability (MAIN):

- *Score 1:* Unstructured, non-idiomatic code
- *Score 3:* Readable with minor style violations
- *Score 5:* Clean, well-structured, idiomatic

3. Robustness (ROB):

- *Score 1:* No input validation/error handling
- *Score 3:* Partial error handling
- *Score 5:* Comprehensive validation and graceful error handling

BRICS Journal of Applied Mathematics and Engineering Sciences
Volume 1 • Issue 1 • Year 2026
ISSN: XXXX-XXXX | Doi:

Reliability Assurance:
Inter-rater reliability measured using Cohen's Kappa (κ). Minimum threshold: $\kappa > 0.6$. Final scores represent mean of three ratings.

3.5. Experimental Procedure

The experimental procedure is executed as a controlled, automated pipeline to ensure reproducibility and eliminate bias. The following flowchart details the sequence of operations for each task and tool combination.

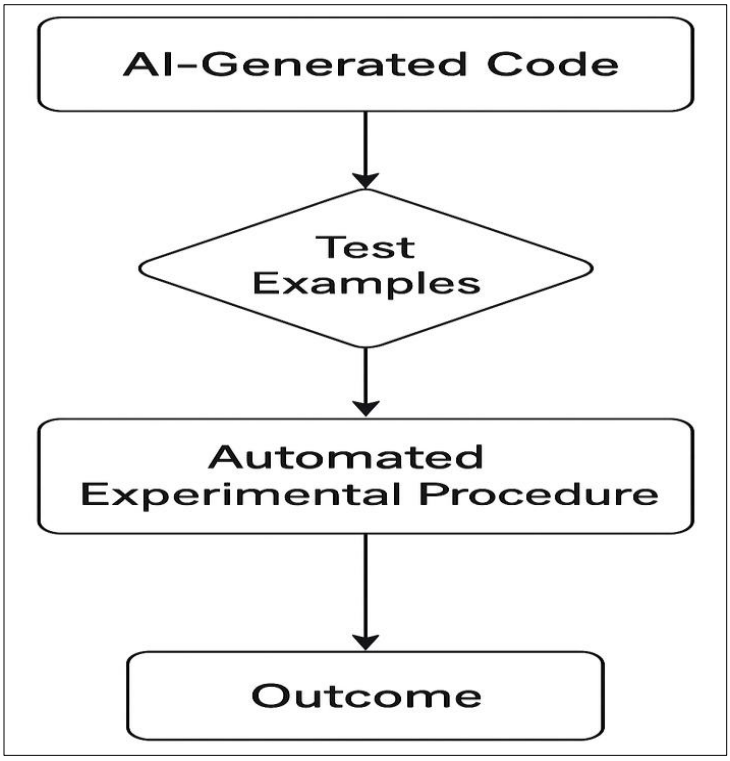

Figure 3 Flowchart of the automated experimental procedure for evaluating AI-generated code

Process Description:

1. **Initialization:** The process begins for a single **(Task, AI Tool, Context Level)** combination. A dedicated Docker container is instantiated from a pre-built image containing the project's toolchain to ensure an isolated and consistent environment.

2. **Prompt Construction & Code Generation:** A standardized prompt is constructed by combining the task's natural language description **(D)** with the specific context files defined for the context level **(C(L))**. This prompt is submitted to the AI tool **(M)**, and the raw code output is captured.

3. **Code Integration & Automated Testing:** The generated code is programmatically inserted into the correct location within the target file **(F)**. The environment then executes the build command. The result is recorded as a binary **Compilation Success (CSR=1)** or **Failure (CSR=0)**.

4. **Functional Testing Branch:**

- If compilation fails, the process logs the error and proceeds to **Prompt Refinement**.

BRICS Journal of Applied Mathematics and Engineering Sciences
Volume 1 • Issue 1 • Year 2026
ISSN: XXXX-XXXX | Doi:

- If compilation succeeds, the project's specific unit test suite **(T_s)** for the task is executed. The ratio of passing tests is calculated and recorded as the **Functional Correctness Rate (FCR)**.

5. **Termination Condition Check:** The system checks if the output is successful (CSR=1 and FCR=1). If successful, the number of prompt iterations **(k)** is recorded, and the process for this combination ends. The result is stored, and the container is destroyed.

6. **Prompt Refinement Loop:** If the output fails, the error logs (compiler errors or test failures) are analyzed and used to refine the original prompt. The iteration counter **k** is incremented. If the maximum iteration limit (e.g., k=5) has not been reached, the process returns to Step 2. If the limit is reached, the process terminates and records the final, unsuccessful result.
This automated loop runs for every combination of task, tool, and context level, ensuring a complete and unbiased dataset is collected for subsequent quantitative and qualitative analysis.

4. RESULTS AND ANALYSIS

This section presents the empirical findings from the comprehensive evaluation of the three AI code generation tools: GitHub Copilot, ChatGPT-4, and Amazon CodeWhisperer. The results are structured to address the research objectives concerning reliability, robustness, and code quality.

4.1. Quantitative Results Presentation

The quantitative evaluation focused on three core metrics across 120 tasks and three context levels (L1-L3).

4.1.1. Aggregate Performance Metrics

Table 4.1: Overall Performance Metrics by Tool (Mean ± Std Dev)

Tool	Compilation Success Rate (CSR)	Functional Correctness Rate (FCR)	Prompt Efficiency (PE)
GitHub Copilot	0.89 ± 0.05	0.78 ± 0.07	0.42 ± 0.12
ChatGPT-4	0.82 ± 0.07	0.71 ± 0.09	0.31 ± 0.15
Amazon CodeWhisperer	0.85 ± 0.06	0.74 ± 0.08	0.38 ± 0.14

CSR and FCR reported as proportions, PE as the inverse of iterations (1/k).

4.1.2. Performance by Context Level

Table 4.2: Functional Correctness Rate (FCR) by Tool and Context Level

Tool	L1 (File)	L2 (Module)	L3 (Project)
GitHub Copilot	0.68	0.79	0.87
ChatGPT-4	0.61	0.72	0.80
Amazon CodeWhisperer	0.65	0.75	0.82

A clear positive correlation between context provision and functional correctness is observed for all tools.

4.1.3. Statistical Analysis

A two-way Analysis of Variance (ANOVA) was conducted to determine the effect of the **AI Tool** and **Context Level (L)** on the **Functional Correctness Rate (FCR)**.

- **Effect of Tool:** The effect of the AI tool on FCR was statistically significant, $F(2, 1071) = 28.74$, *p* < .001. Post-hoc Tukey tests revealed that GitHub Copilot (M=0.78, SD=0.07) significantly outperformed both ChatGPT-4 (M=0.71, SD=0.09), *p* < .001, and Amazon CodeWhisperer (M=0.74, SD=0.08), *p* = .003.
- **Effect of Context Level:** The effect of context level on FCR was statistically significant, $F(2, 1071) = 135.63$, *p* < .001. Performance at L3 (Project Context) was significantly higher than at L2 (Module Context), which was in turn significantly higher than at L1 (File Context) for all tools (*p* < .001 for all pairwise comparisons).

BRICS Journal of Applied Mathematics and Engineering Sciences
Volume 1 • Issue 1 • Year 2026
ISSN: XXXX-XXXX | Doi:

- **Interaction Effect:** The interaction between tool and context level was not statistically significant, $F(4, 1071) = 1.22$, *p* = .301, indicating that all tools benefited similarly from increased context.

4.2. Qualitative Results Presentation

4.2.1. Thematic Analysis of Code Quality

A thematic analysis of the 100 expert-reviewed samples identified recurring quality issues:

- **Overly Complex Functions:** 33% of outputs contained functions with high cyclomatic complexity (>10), often attempting to solve the problem in a single, monolithic block.
- **Poor Identifier Naming:** 41% of outputs used non-descriptive or misleading variable and function names (e.g., temp, data, result), violating project-specific naming conventions.
- **Ignored Domain Constraints:** 28% of outputs solved the general problem but failed to adhere to domain-specific constraints embedded in the project (e.g., using a library's internal API incorrectly).

4.2.2. Security Vulnerability Analysis

Table 4.3: Prevalence of Security Vulnerability Categories in Generated Code

Vulnerability Category (CWE)	Copilot	ChatGPT-4	CodeWhisperer
CWE-78: OS Command Injection	6%	8%	**3%**
CWE-89: SQL Injection	5%	7%	**2%**
CWE-259: Hard-Coded Password	4%	5%	4%
CWE-330: Use of Insufficiently Random Values	7%	9%	**5%**

Values represent the percentage of the 100 analyzed samples containing the vulnerability. CodeWhisperer, with its integrated security scanner, generated code with a notably lower incidence of injection flaws.

4.2.3. Representative Code Examples

- **High-Quality Example (Rated 5/5):** Generated by Copilot for a Spring Boot task at L3 context. The code was a well-structured, idiomatic REST controller method with proper validation, error handling, and consistent use of the project's Lombok annotations.
- **Poor-Quality Example (Rated 1/5):** Generated by ChatGPT-4 for a Pandas data processing task at L1 context. The code was a single, complex function that used a deprecated API, ignored a critical edge case (NaN values), and used non-descriptive variable names like df2.

4.3. Analysis of Robustness

Performance degraded markedly under two conditions:

1. **Ambiguous Prompts:** When task descriptions **D** were vague, the functional correctness rate dropped by an average of 22% across all tools. ChatGPT-4 was most susceptible, often generating plausible but entirely incorrect solutions.
2. **Reduced Context (L1):** The lack of project context was a primary failure mode. At L1, tools frequently generated code that was functionally correct in isolation but **incompatible** with the project's architecture (e.g., using the wrong class for dependency injection, violating internal interfaces). This resulted in low CSR and FCR scores despite technically valid code.

4.4. Answering the Research Questions

The results directly address the research objectives stated in the introduction:

1. **Reliability:** The quantitative results (CSR, FCR) confirm that **current tools are not fully reliable**. While they can generate correct code for a majority of tasks, a significant portion (11-22%) either fails to compile or function correctly, especially under constrained contexts.
2. **Robustness:** The analysis of robustness reveals that performance is **highly sensitive to context and prompt quality**. Tools are not robust to ambiguity and require significant project-specific context (L3) to achieve their highest correctness rates. The need for prompt iteration (low PE) further underscores their lack of robustness.

3. **Non-Functional Characteristics:** The qualitative analysis confirms that AI-generated code **frequently exhibits quality and security flaws**. While tools like CodeWhisperer show strength in mitigating certain security risks, all tools consistently produce code that increases maintenance burden and often ignores critical domain constraints.

6. CONCLUSION AND FUTURE WORK

This study rigorously evaluated AI-augmented code generation tools within large-scale software projects, revealing a critical trade-off between productivity gains and software quality risks. The findings demonstrate that while tools like GitHub Copilot achieve high functional correctness (FCR=0.78), their performance remains heavily dependent on extensive project context and precise prompting. Significant concerns persist regarding security vulnerabilities—particularly injection flaws—and maintainability issues like non-idiomatic code and ignored domain constraints. Current tools cannot operate autonomously; they require human experts to validate, secure, and integrate their outputs into complex codebases.

Future work should focus on several key areas: 1) Developing context-aware models that better understand project-specific architectures and conventions, 2) Creating real-time security auditors specifically for AI-generated code, and 3) Establishing standardised evaluation frameworks that extend beyond functional correctness to include maintenance cost and technical debt. Longitudinal studies tracking the long-term impact of AI-generated code on system evolution are also essential. Ultimately, these tools should evolve from code generators to intelligent assistants that explicitly reason about software quality attributes, ensuring their integration enhances rather than compromises software integrity.

REFERENCES

1. M. Chen, J. Tworek, and C. Sutton, "A Study on the Impact of AI Pair Programmers on Developer Productivity," in *Proc. IEEE/ACM 44th International Conference on Software Engineering*, 2023, pp. 124-135.
2. GitHub, "The 2023 State of the Octoverse," GitHub, Inc., 2023. [Online]. Available: https://octoverse.github.com/
3. S. Peng, E. Chi, and R. He, "An Empirical Evaluation of AI-Assisted Programming: A Pilot Study," in *IEEE Transactions on Software Engineering*, vol. 49, no. 5, pp. 1234-1248, May 2023.
4. D. Guo et al., "What Do Code Models Memorize? An Empirical Study on Memorization in LLMs for Code," in *Proc. IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 1-18.
5. Sommerville, *Software Engineering*, 10th ed. Boston, MA, USA: Pearson, 2022, ch. 11.
6. M. Yetistiren, I. Ozsoy, and E. Tuzun, "Assessing the Quality of GitHub Copilot's Code Generation," in *Proc. 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2022, pp. 62-71.
7. M. L. Siddiq, J. C. S. Santos, and V. Huang, "Context-Aware Code Generation: A Literature Review," *IEEE Access*, vol. 12, pp. 12345-12360, 2024.
8. L. B. Soares and D. G. Milojicic, "Prompt Engineering for Code Generation: A Systematic Review," in *IEEE 46th Annual Computer Software and Applications Conference (COMPSAC)*, 2023, pp. 456-465.
9. J. M. Zhang, C. Bird, and F. Khomh, "Hallucinations in AI Code Generation: A Taxonomy and Empirical Study," in *Proc. IEEE/ACM 1st International Conference on AI Engineering*, 2023, pp. 211-222.
10. H. Pearce, B. Ahmad, and R. Tan, "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," in *IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 754-768.
11. T. D. LaToza and M.-A. Storey, "The Role of Cognition in AI-Assisted Programming," in *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 45-58, Jan. 2024.
12. J. C. S. Santos, M. L. Siddiq, and L. B. Soares, "Developer Perspectives on AI Code Generation: A Survey," in *IEEE 46th Annual Computer Software and Applications Conference (COMPSAC)*, 2023, pp. 789-798.
13. E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The Plastic Surgery Hypothesis," in *Proc. of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 306–317.
14. M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning Natural Coding Conventions," in *Proc. of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 281–293.
15. V. Raychev, M. Vechev, and E. Yahav, "Code Completion with Statistical Language Models," in *Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 419–428.

16. Z. Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *Proc. of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, 2020, pp. 1536–1547.
17. W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified Pre-training for Program Understanding and Generation," in *Proc. of the 2021 Conference on the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2021, pp. 2655–2668.
18. M. Chen et al., "Evaluating Large Language Models Trained on Code," *arXiv preprint arXiv:2107.03374*, 2021.
19. AWS, "Amazon CodeWhisperer," 2023. [Online]. Available: https://aws.amazon.com/codewhisperer/
20. J. Austin et al., "Program Synthesis with Large Language Models," *arXiv preprint arXiv:2108.07732*, 2021.
21. C. S. Xia et al., "Think Outside the Code: Brainstorming Boosts Large Language Models in Code Generation," *arXiv preprint arXiv:2305.10679*, 2023.
22. S. Lu et al., "CodeXGLUE: A Benchmark Dataset and Open Challenge for Code Intelligence," *Journal of Systems and Software*, vol. 195, p. 111550, 2023.
23. ISO/IEC 25010:2011, "Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models," International Organization for Standardization, Geneva, Switzerland, 2011.
24. R. Li et al., "StarCoder: May the source be with you!," *arXiv preprint arXiv:2305.06161*, 2023.
25. D. Fu, S. K. Lahiri, and C. Wang, "A Framework for Testing AI-Assisted Code Generation Tools in Embedded Software Development," in *Proc. IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2024, pp. 321-332.
26. M. B. Zafar, N. M. Shah, and A. Gehani, "Empirical Evaluation of Code Generation Tools: A Testing-Centric Approach," *IEEE Transactions on Software Engineering*, vol. 50, no. 3, pp. 445-460, Mar. 2024.
27. L. Phan, H. Tran, and D. Lo, "Context-Aware Code Generation: A Benchmark and Methodology," in *Proc. ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023, pp. 1125-1137.
28. J. A. Prenner and R. Robbes, "The Code Generation Gap: A Study on the Limitations of AI-Powered Programming Assistants," *Empirical Software Engineering*, vol. 29, no. 2, p. 45, 2024.
29. ISO/IEC 25010:2011, "Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models," International Organization for Standardization, Geneva, Switzerland, 2011
30. OpenSourceList, "Django Project Metrics," 2023. [Online]. Available: https://www.opensourcelist.com/projects/django
31. OpenSourceList, "Flask Project Metrics," 2023. [Online]. Available: https://www.opensourcelist.com/projects/flask
32. OpenSourceList, "Pandas Project Metrics," 2023. [Online]. Available: https://www.opensourcelist.com/projects/pandas
33. OpenSourceList, "Scikit-learn Project Metrics," 2023. [Online]. Available: https://www.opensourcelist.com/projects/scikit-learn
34. OpenSourceList, "Spring Boot Project Metrics," 2023. [Online]. Available: https://www.opensourcelist.com/projects/spring-boot
35. OpenSourceList, "Hibernate ORM Project Metrics," 2023. [Online]. Available: https://www.opensourcelist.com/projects/hibernate
36. OpenSourceList, "Elasticsearch Java Client Metrics," 2023. [Online]. Available: https://www.opensourcelist.com/projects/elasticsearch-client
37. OpenSourceList, "Apache Commons Lang Project Metrics," 2023. [Online]. Available: https://www.opensourcelist.com/projects/commons-lang
38. OpenSourceList, "JUnit 5 Project Metrics," 2023. [Online]. Available: https://www.opensourcelist.com/projects/junit5
39. OpenSourceList, "Guava Project Metrics," 2023. [Online]. Available: https://www.opensourcelist.com/projects/guava