Evaluating LLM-Generated Unit Tests with Mutation Testing: ChatGPT vs DeepSeek

Pedro Fernando Marinho Cabral
Federal University of Pará - UFPA
Belém, Pará, Brazil
pedro.cabral@icen.ufpa.br

João Pedro Souza Arruda
Federal University of Pará - UFPA
Belém, Pará, Brazil
joao.arruda@icen.ufpa.br

Cleidson Ronald Botelho de Souza
Federal University of Pará - UFPA
Belém, Pará, Brazil
cleidson.desouza@acm.org

Victor Hugo Santiago Costa Pinto
Federal University of Pará - UFPA
Belém, Pará, Brazil
victor.santiago@ufpa.br

ABSTRACT

Recent advances in Large Language Models (LLMs) have driven significant progress in automating software testing, particularly in generating unit tests. However, the effectiveness of these models in detecting real defects through mutation testing remains underexplored in the literature. This study aims to address this gap by evaluating the performance of ChatGPT (GPT-4o) and DeepSeek V3 in generating unit tests for six Java classes from the Defects4J dataset, covering different levels of cyclomatic complexity. The main objective is to investigate the ability of LLMs to maximize mutant coverage and elimination, while also analyzing the impact of code complexity and semantic factors related to execution failures. The methodology involved generating tests via structured prompts, executing them 5 times per class for both models, and performing quantitative analysis based on Mutation Coverage (MC) and Mutation Score (MS), as well as qualitative analysis of runtime failures. Results indicate that DeepSeek exhibits greater stability and effectiveness in eliminating mutants, whereas ChatGPT demonstrates broader applicability by producing valid test suites for a wider range of classes. Moreover, no significant correlation was found between cyclomatic complexity and compilation success, with failures primarily linked to semantic limitations of the models. This study presents both quantitative and qualitative evidence on the application of LLMs for automated test generation, offering insights for future AI-driven test engineering strategies.

KEYWORDS

large language models; unit test generation; mutation testing; cyclomatic complexity; software test automation

1 Introduction

The evolution of Large Language Models (LLMs) has influenced several domains of software engineering [11], including code generation [5, 32], pull request workflows [31] and test automation [34]. Within this context, unit test generation remains a central challenge for software quality assurance, as it demands robust and relevant test cases capable of revealing potential faults at the source code level [1]. To assess the effectiveness of such tests, mutation testing provides a fault-based strategy that introduces small syntactic changes into the program and evaluates whether the tests can detect them [12].

The growing interest in applying LLMs to unit test generation suggests that these models may contribute to improving test coverage and productivity in software testing, their effectiveness however, can still be evaluated through different approaches, including mutation testing [7]. Recent studies [29] indicate that LLMs can generate diverse and comprehensive tests, reducing dependence on traditional heuristics in test case design. Nonetheless, there is no clear consensus in the literature: while some works emphasize the potential of LLMs [29], others highlight challenges such as the generation of equivalent or semantically neutral mutants that often go undetected [27, 28]. These limitations underscore the need for systematic evaluation strategies to better understand the effectiveness of LLMs in mutation testing, as well as how different usage configurations, including prompt formulation [9, 17, 24], can affect the quality of generated tests. To guide our investigation, we defined the following research questions:

*RQ*1: How does the cyclomatic complexity of classes influence the success rate of compiling unit tests generated by LLMs?

*RQ*2: What are the reasons behind the runtime failures of test suites generated by LLMs that compile correctly?

*RQ*3: To what extent can LLMs generate high-quality unit tests that effectively maximize coverage and detect faults in source code?

RQ1 investigates structural characteristics, focusing on the cyclomatic complexity of the target classes and the test suite compilation success of the LLM-generated test suites. This research question complements RQ2, which analyzes the semantic behavior during runtime, and RQ3, which evaluates the mutation-based effectiveness of the generated tests. Moreover, RQ1 serves as a foundational stage in the experimental workflow, directly influencing the subsequent phases of analysis and validation.

A comparative empirical study was conducted between ChatGPT and DeepSeek, with controlled test generation for six target classes from the Defects4J repository [25]. The models were selected for their state-of-the-art performance in code generation and their distinct architectural characteristics, providing a meaningful basis for comparison. This design enabled the identification of complementary strengths and limitations of LLMs in automated unit test generation.

In addition, the compiled test suites were initially analyzed for runtime failures to verify semantic correctness and identify error patterns. Only the suites that compiled and executed successfully were subsequently evaluated using mutation metrics, providing

an integrated view of the quality and robustness of the generated tests. Mutation Coverage (MC) and Mutation Score (MS) [1, 8] were employed as the leading indicators of test suite quality.

The results revealed each model's strengths and weaknesses, as well as the leading causes of execution failures, providing actionable insights for LLM-assisted software testing. Beyond quantitative and qualitative evidence of effectiveness, this study advances automated testing by demonstrating LLMs' potential to improve productivity, fault detection and overall software quality.

The remainder of this paper is organized as follows. Section 2 introduces the background on software testing and LLMs. Section 3 reviews related work. Section 4 describes the experimental setup, and Section 5 presents the results. Section 6 discusses practical implications. Section 7 discusses threats to validity, while Section 8 concludes the paper.

2 Background

2.1 Software Testing and Automation with LLMs

Software testing plays a fundamental role in ensuring that a system behaves as expected and meets its requirements [1]. Among the various testing levels, including integration, acceptance, and system testing, unit testing focuses on verifying individual components, such as methods or classes. This enables early defect detection and helps prevent failures from propagating into production systems [3, 20]. Furthermore, to improve reliability and reduce maintenance costs [3], test automation enhances the development process by enabling frequent executions after each modification, which is crucial for maintaining quality over time [1].

In addition, in recent years, LLMs have emerged as promising tools to support unit test automation. Initially developed for Natural Language Processing (NLP) tasks, LLMs are increasingly being applied to software engineering contexts, including test generation, defect localization, and suite optimization [2, 29]. According to Bayri and Demirel [2], these models enable the automation of critical stages in the testing lifecycle, reducing manual effort and accelerating defect identification. Wang et al. [29] further emphasize that LLMs are promising not only for increasing productivity but also as a transformative technology for traditional testing practices.

Despite their potential, the use of LLMs for automatically generating relevant test cases still poses challenges. Recent studies indicate that even advanced models such as GPT-4 struggle to correctly map inputs to expected outputs in complex scenarios due to limitations in reasoning and computational precision [16, 22, 33]. Additionally, the non-deterministic behavior of LLMs can produce inconsistent results for the same prompt. A study conducted by Ouyang et al. [24] found that only 21.1% of the 76 papers analyzed explicitly considered this factor, adopting practices such as multiple executions and the use of metrics with variance. Given this context, this paper investigates LLMs' ability to generate unit tests from Java program source code, thereby enabling evaluation of the reliability and effectiveness of the generated artifacts.

2.2 Mutation Testing as a Quality Indicator

Mutation testing is a fault-based testing approach widely adopted to evaluate the effectiveness of test suites in revealing defects in software systems [7]. This technique operates by introducing small syntactic changes referred to as mutants into the source code [8]. These mutants simulate typical programming errors that may occur during development. When a test case causes a mutant to produce an output different from the original program's, the mutant is considered killed, indicating that the test can detect that type of fault.

Two primary metrics are employed in mutation testing to quantify test quality: Mutation Coverage (*MC*) and Mutation Score (*MS*). *MC* measures the proportion of generated mutants that are executed by the test suite, reflecting the reachability of potential faults. *MS*, in turn, represents the percentage of executed mutants that are effectively killed by the tests, indicating the suite's fault-detection capability. These metrics, widely adopted in the literature [12], are formally defined as:

$MC = (M_t/M)$ x 100, where M is the total number of mutants and M_t is the number of mutants exercised by the test cases.

$MS = (M_k/M_t)$ x 100, where M_k denotes the number of killed mutants among those executed.

Although mutation testing is a valuable approach to generate or assess the quality of test suites, it is often regarded as a high-cost criterion due to: (i) the large number of mutants generated; (ii) the time-consuming process of identifying equivalent mutants; and (iii) the execution time required for running the mutants.

Determining equivalent mutants [7] is known to introduce redundant costs and biases, hindering the effectiveness of mutation testing in practice. In such cases, the mutant behaves identically to the original program for all possible inputs and, therefore, cannot be killed by any test case. In this way, the presence of equivalent mutants may negatively affect the Mutation Score values, thereby underestimating the test suite's actual fault-detection capability.

2.3 Cyclomatic Complexity and Class Selection Rationale

Cyclomatic complexity, introduced by McCabe [18], is a structural metric that quantifies the logical complexity of a program based on its control flow graph. It reflects the number of independent execution paths and serves as an indicator of the testing effort required for adequate coverage. Higher complexity typically entails greater branching and conditional logic, posing additional challenges for automated test generation.

To analyze the influence of structural complexity on LLM performance in test generation, the experiment used six Java classes with varying cyclomatic complexity. Three criteria guided this selection: project diversity, testability and complexity variation, as detailed in Subsection 4.1. Rather than applying categorical thresholds (e.g., low, medium, high complexity), this study preserved the absolute values of complexity to maintain analytical fidelity and avoid arbitrary classification.

3 Related Work

Nan et al. [21] proposed *IntUT*, a novel approach that leverages explicit test intentions comprising input parameters, mock behaviors, and expected results to guide LLMs in generating unit tests. Their methodology integrates program analysis via the PAINT technique to automatically extract test intentions from control-flow graphs and mocking decisions, which are then embedded into the prompts provided to LLMs. Evaluations on three industrial Java projects

and a live user study demonstrated that IntUT significantly improves branch coverage (up to 94%) and line coverage (up to 49%) compared to baseline LLM-generated tests, while also achieving high developer acceptance in practice. Unlike our study, which emphasizes mutation testing as the primary metric of test quality and explores qualitative runtime failure analysis, their focus lies in enhancing structural coverage and mocking accuracy through intention-guided prompting.

Hossain and Dwyer [10] introduced *TOGLL*, an innovative LLM-based approach for automated test oracle generation. Their study fine-tuned seven code LLMs with six prompt formats on the SF110 dataset and evaluated the most effective configuration on the OracleEval25 benchmark as well as on Defects4J. TOGLL was compared with EvoSuite and the neural baseline TOGA, achieving up to 3.8 times more correct assertion oracles, 4.9 times more exception oracles, and detecting 1,023 unique mutants missed by EvoSuite. The results demonstrated that fine-tuned LLMs can generate accurate, strong, and diverse test oracles, significantly reducing false positives and surpassing TOGA in both mutation-based and real bug detection effectiveness. In contrast to our study, TOGLL specifically addresses the oracle generation problem, complementing but not overlapping with our research scope.

Additionally, Tip et al. [28] propose LLMorpheus, a tool that leverages LLMs to generate diverse mutants in JavaScript code, replacing traditional mutation operators with context-aware suggestions via prompts. The approach aims to increase the variety and realism of mutants, overcoming limitations of conventional tools. While the goal of LLMorpheus is to enrich the mutation testing process by generating more expressive mutants, this study investigates the ability of LLM-generated unit tests to detect defects, using indicators such as Mutation Coverage (MC) and Mutation Score (MS). Furthermore, unlike LLMorpheus, which was evaluated on 13 JavaScript/TypeScript packages using the StrykerJS tool, this study uses Java classes extracted from the Defects4J benchmark to explore different scenarios and challenges related to code structure, including variations in the absolute values of cyclomatic complexity.

Gaps in the Literature and Paper Contribution

Although recent works, such as those by Nan et al. [21], Hossain and Dwyer [10], and Tip et al. [28], have explored different perspectives on the use of LLMs for unit test generation, there remains a need for rigorous empirical investigations to support the reliable use of these models. To address this demand, this paper employs mutation testing as the central evaluation criterion and compares, under controlled experimental conditions, the ability of the ChatGPT and DeepSeek models to generate unit tests written in Java. The study uses quantitative indicators (Mutation Coverage and Mutation Score) and qualitative analyses of runtime failures to provide empirical evidence on the applicability and limitations of LLMs, advancing strategies for AI-assisted test automation.

4 Methodology

This study adopts a structured experimental methodology to evaluate LLMs' ability to generate unit tests, with effectiveness assessed through mutation testing. Figure 1 summarizes the experimental workflow, whose stages directly correspond to the methodological components detailed in Section 4: dataset and class selection (Section 4.1), test case generation with LLMs (Section 4.2), test validation (Section 4.3), mutant injection (Section 4.4), and evaluation metrics (Section 4.5).

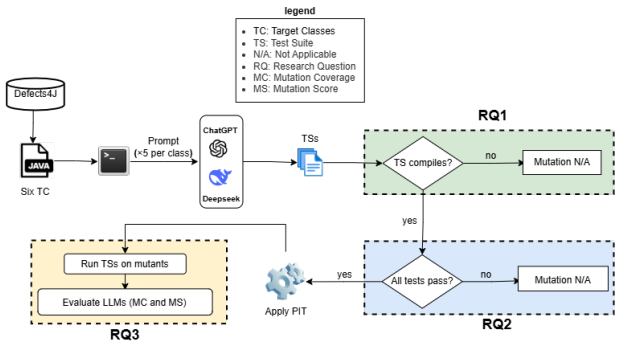

Figure 1: Experimental Workflow

4.1 Dataset and Class Selection

We selected six Java classes from the Defects4J benchmark, a widely used dataset comprising real-world bugs and their corresponding fixes [25]. Classes were chosen from three distinct projects based on three main criteria: (i) diversity in code structure and domain; (ii) presence of methods suitable for unit testing; and (iii) variation in cyclomatic complexity, assessed via SonarQube. The goal was to investigate whether internal code characteristics, beyond complexity alone, influence LLM performance.

Diversity in code structure and domain: The six target classes were selected from three distinct projects within the Defects4J dataset. Although not all classes belong to different projects, we ensured that each selected project contributes multiple classes with distinct characteristics. This choice aimed to maintain diversity in domains, coding styles, and internal structures, enabling a broader analysis of the LLMs' ability to handle different implementation patterns in real-world systems.

Methods suitable for unit testing: The project should include classes with sufficiently diverse methods to enable the creation of meaningful unit tests; and

Cyclomatic Complexity Variation: Classes with different levels of cyclomatic complexity were selected to analyze the impact of code structure on LLM performance. The metric was obtained using the SonarQube tool [26], employed exclusively for this purpose, without considering or correcting other reported issues, while keeping the original Defects4J classes to ensure greater reproducibility.

The decision to use only six classes in this phase of the study is due to the fully manual execution of all stages of the experiment, including prompt formulation, repeated model execution, validation of the generated test suites, mutant injection, and failure log analysis. This manual approach was necessary to gain a deeper understanding of LLM behavior and the limitations of the tools involved. While restrictive in scale, this methodological decision provided greater control over experimental variables and

enabled more detailed qualitative analyses. In Table 1, the six selected classes are shown with their corresponding projects and cyclomatic complexity value.

Table 1: Selected classes with corresponding projects and cyclomatic complexity.

Class	Project	Cyclomatic Complexity
Primes.java	Commons-math3-3.2	17
Hex.java	Commons-codec	21
BinaryCodec.java	Commons-codec	35
CharRange.java	Commons-lang	45
Range.java	Commons-lang	57
Fraction.java	Commons-math3-3.2	77

4.2 Test Case Generation with LLMs

4.2.1 Selected LLMs. As discussed before, the use of LLMs has emerged as a promising strategy for test suite generation, showing strong potential to automate and enhance the execution of unit tests [2]. Previous studies indicate that GPT-4 is superior in unit test generation compared to open-source LLMs and traditional approaches such as Evosuite [32], in terms of both coverage and defect detection capabilities [2]. Therefore, we decided to use ChatGPT, based on GPT-4o, in this study, given its enhanced potential as a direct successor to GPT-4.

Meanwhile, DeepSeek Chat V3 is an evolution of the DeepSeek series, designed for advanced programming and code-generation tasks and optimized for competitive performance and accessibility compared to high-end proprietary models [6]. Although its exact architecture is not publicly documented, it belongs to the DeepSeek family, whose open-source variant (DeepSeekCoder) has shown competitive performance in unit test generation tasks [32]. These factors reinforce the selection of this model, considering its alignment with software engineering tasks.

Based on the arguments above, this study compares two models widely used in real-world development environments: ChatGPT Plus (GPT-4o) by OpenAI [23], and DeepSeek Chat V3 [6], a recent and promising alternative designed for programming tasks. The comparison between these models is also justified by their distinct architectural and training approaches, as discussed in recent studies [32], enabling the evaluation of different strategies for automated unit test generation.

4.2.2 Prompt Strategy. The input strategy employed is referred to as zero-shot prompting [14][29], in which the task to be performed is explicitly described to the model without providing prior input-output examples. This decision aimed to avoid semantic interference caused by previous examples while also leveraging the models' generalization and reasoning capabilities when presented with well-defined instructions.

More specifically, six prompts were formulated—one for each class. Each prompt followed a standardized structure, with clear and well-defined instructions to guide the models in generating an appropriate test suite. Accordingly, it was established that the tests should be written in Java using the JUnit 5 framework [13], adhering to naming best practices and covering different usage scenarios of the target class. Additionally, the LLMs were instructed to behave as experienced test professionals, focusing on covering relevant execution paths and maximizing the *Mutation Score*.

As a complementary strategy, each prompt was executed five times for each target class using the web interface of the respective model, always initiating a new conversation to avoid the influence of prior interaction history. This is necessary to deal with the LLM non-determinism in code generation [24]. Therefore, the results obtained from each of the five executions per prompt were stored for subsequent analysis.

Although recent studies have proposed empirically validated prompting techniques for unit test generation [30, 35], their formats differ significantly from the approach adopted in this work. For instance, HITS [30] is an LLM-based approach for unit test generation that uses multi-step prompts with method slicing and chain-of-thought reasoning to optimize branch coverage. At the same time, Yuan et al. [35] focus on iterative prompt refinement for method-level tests. In contrast, our objective required standardized zero-shot prompts applied at the class level, designed explicitly for mutation-based evaluation. This choice enables fair comparisons across models under consistent and reproducible conditions, with an emphasis on the semantic quality of the generated tests rather than structural coverage alone.

4.3 Test Validation

Five test suites (TS), each comprising multiple test cases (TCs), were generated by each LLM for each target class and executed in the integrated development environment (IDE) to validate their functionality and ensure they did not produce failures due to incorrect generation. Importantly, only the test suites that successfully compiled and achieved a 100% pass rate across all their test cases were deemed eligible for subsequent mutation testing.

4.4 Mutant Injection

The PIT (Pitest) tool [4] was adopted in this study for injecting mutants into the selected Java classes due to its wide recognition for efficiency, scalability, and native integration with testing frameworks such as JUnit. It enables the automated execution of unit tests on instrumented versions of the code, identifying which mutants were "killed" (i.e., detected by the tests) and which ones "survived" (i.e., undetected). In addition to its effectiveness in evaluating the quality of test suites, PIT stands out for its optimized performance, active community support, and clear and detailed report generation [4]. These features have contributed to its consolidation as a reference tool in recent academic experiments, as evidenced by its use as a foundation for higher-order mutation extensions, such as PIT-HOM [15].

The injection of mutants was performed as a step independent of the execution of the test suites (TS) produced by the LLMs. This methodological decision aimed to ensure a fair and appropriate comparison between the models by guaranteeing that each TS, associated with a target class, was always executed on the same set of mutants. To achieve this, a single PIT run was performed for each target class, generating a fixed set of mutants. These mutants were then reused across all executions of the test suites (TS1 to TS5) generated by each LLM, eliminating any variation that could compromise the equivalence of experimental conditions. However,

PIT requires at least one test suite that invokes methods from the target class to generate mutants. Thus, a base class containing minimal tests had to be manually created for each target class. This minimal test suite served solely to activate the mutation points in the code through the tool and was later replaced by the test suites generated by the LLMs during the execution phase.

All mutation operators used in this experiment were enabled by default in PIT, with no manual modifications. The tool performs mutations directly on the Java bytecode rather than on source files, making the process faster and easier to integrate into the build environment. The default operators are automatically selected based on the characteristics of the code and include transformations such as arithmetic substitutions, logical condition modifications, return value alterations, among others [4]. According to the official documentation[1], these operators are designed to be stable, that is, not trivial to detect, which minimizes, but does not eliminate, the generation of equivalent mutants. Furthermore, operators that do not meet these criteria are disabled by default. Therefore, the choice to use the default configuration is justified, as this study does not aim to evaluate the mutation operators themselves, but rather to analyze the effectiveness of the tests generated by LLMs. It is worth noting that alternative approaches have recently emerged, such as LLMorpheus [28], which leverages large language models to generate realistic JavaScript mutants. However, in this study we adopted PIT given its maturity, integration with JUnit, and wide adoption in empirical software testing research, ensuring reproducibility of our experimental setup.

Finally, although this study acknowledges the issue of equivalent mutants, a well-known limitation of the mutation testing technique widely discussed in the literature [7], no specific strategy was employed to detect or handle them. As previously mentioned, this limitation may affect the absolute Mutation Score (MS) values, since the presence of mutants that are semantically equivalent to the original code and cannot be killed may artificially lower the score. However, to mitigate this issue and ensure fair comparison across test suites, all executions were performed on the same fixed set of mutants for each target class. By holding the mutation conditions constant, the experimental design enables reliable relative comparisons between LLM-generated test suites, even in the presence of potentially equivalent mutants.

4.5 Evaluation Metrics

The analysis was structured around the research questions, each addressed through specific metrics and procedures:

To answer RQ1 (*How does the cyclomatic complexity of classes influence the success rate of compiling unit tests generated by LLMs?*), we computed the Compilation Success Rate (CSR) for each target class and model. Additionally, we applied Spearman's correlation coefficient (ρ) to investigate potential relationships between cyclomatic complexity and compilation success.

> **Compilation Success Rate (CSR)**: Represents the percentage of test suites that compiled successfully relative to the total number generated per class and LLM. This metric enables the evaluation of the feasibility of using the generated tests in subsequent stages of the experiment. CSR is calculated as $\text{CSR} = \left(\frac{T_{sc}}{T_{st}}\right) \times 100$, where T_{sc} is the number of test suites that compiled successfully and T_{st} is the total number of test suites generated.
>
> **Spearman's Correlation (ρ)**: To measure the association between the cyclomatic complexity of the classes and the compilation success rate, Spearman's correlation (ρ) was used. This is a non-parametric statistical technique suitable for monotonic relationships and small sample sizes [36].

Regarding RQ2 (*What are the reasons behind the runtime failures of test suites generated by LLMs that compile correctly?*), we conducted a manual inspection of execution logs from test suites that compiled successfully but failed during execution. These failures were categorized into two main types: Assertion Failures and Runtime Exceptions. This classification enabled us to identify semantic limitations in the generated tests and to detect recurring error patterns across different models and classes.

> **Initial Quantitative Evaluation**: A filtering process was conducted on the test suites that, although compiled successfully, failed during execution. This analysis allowed for the identification of actual functional failures[2] in suites considered syntactically valid.
>
> **Qualitative Analysis of Error Logs**: The runtime error logs were manually reviewed and categorized based on the nature of the identified issue. Failures were grouped into two main categories: **Assertion Failure**: failures related to incorrect assertions or inverted expectations in the tests; and **Runtime Exception**: failures caused by unhandled exceptions, invalid inputs, or type conversion errors (casts).

This approach enabled the construction of a catalog of recurring failures by class and LLM model, revealing patterns of behavior by the LLMs and their semantic limitations in understanding the target classes.

Finally, to address RQ3 (*To what extent can LLMs generate high-quality unit tests that effectively maximize coverage and detect faults in source code?*), we employed MC and MS as metrics to evaluate the effectiveness of the generated test suites. To statistically compare the performance of the models, we conducted a t-test on the MS values, enabling us to assess whether the differences in test quality were significant across models.

5 Results

This section presents the results obtained from the execution of the experiment. The data are analyzed based on the previously defined metrics, enabling a direct comparison between the evaluated LLMs. The organization of the results aims not only to describe the findings but also to support a critical discussion on the performance and limitations observed in each model.

[1] https://pitest.org/quickstart/mutators/

[2] Functional failures refer to cases in which a test suite compiles and runs, but produces incorrect or unexpected outputs, such as failed assertions or invalid results.

5.1 RQ1: How does the cyclomatic complexity of classes influence the success rate of compiling unit tests generated by LLMs?

To answer RQ1, the analysis was structured into three topics: (i) Compilation Success Rate (CSR), (ii) Correlation between Cyclomatic Complexity and CSR, and (iii) Graphical Analysis. Each aspect is discussed individually in the following subsections.

5.1.1 Compilation Success Rate (CSR). Table 2 presents the information about the cyclomatic complexity (CC), the class name, the LLM used, the number of test suites that compiled successfully, and the corresponding success rate. These data provide a clear view of the models' ability to generate syntactically valid tests, as this condition is a prerequisite for the tests to be used in subsequent stages of mutant analysis.

Table 2: Test suite compilation success per class, LLM, and cyclomatic complexity.

CC	Class	LLM	Compiled TS	CSR
17	Primes	ChatGPT	5	100.00%
		DeepSeek	5	100.00%
21	Hex	ChatGPT	4	80.00%
		DeepSeek	0	0.00%
35	BinaryCodec	ChatGPT	0	0.00%
		DeepSeek	0	0.00%
45	CharRange	ChatGPT	5	100.00%
		DeepSeek	0	0.00%
57	Range	ChatGPT	5	100.00%
		DeepSeek	5	100.00%
77	Fraction	ChatGPT	5	100.00%
		DeepSeek	5	100.00%

Upon inspecting the data from Table 2, it is possible to notice that the ChatGPT model achieved a 100% compilation success rate in four out of the six evaluated classes, failing partially in `Hex` (CC = 21) and completely in `BinaryCodec` (CC = 35). DeepSeek, on the other hand, achieved 100% success only in the `Primes`, `Range`, and `Fraction` classes, and failed completely in the others. On average, the compilation success rate was 63.33% for ChatGPT and 50.00% for DeepSeek, reflecting a noticeable difference in syntactic viability between the models.

Interestingly, classes with higher cyclomatic complexity, such as `Fraction` (CC = 77) and `Range` (CC = 57), did not prevent either model from generating tests that compiled correctly. On the other hand, failures occurred in classes with intermediate complexity, such as `Hex` (CC = 21) and `CharRange` (CC = 45), especially for DeepSeek, which showed a 0% success rate in both cases. These results suggest that cyclomatic complexity alone is not a reliable factor for predicting compilation success. We hypothesize that the internal structure of the classes, the use of APIs, generic types, or overloaded methods appears to have a more direct influence on the LLMs' ability to generate syntactically valid code. This is further explored in the next subsection through a statistical correlation analysis.

5.1.2 Correlation between Cyclomatic Complexity and CSR. To verify whether there is a consistent relationship between the cyclomatic complexity of the classes and the compilation success rate of the generated test suites, we used Spearman's correlation coefficient (ρ). This non-parametric statistical technique is suitable for small data sets and does not assume linearity between variables, making it appropriate for empirical studies in software engineering [36]. The correlation values obtained for each model were:

- ChatGPT: $\rho = 0.37$ (weak positive correlation);
- DeepSeek: $\rho = 0.29$ (weak positive correlation).

The values were calculated using `Python` scripts with the *scipy.stats* library[3]. These results indicate that, for both models, no significant monotonic relationship was observed between the cyclomatic complexity of the classes and the compilation success rate.

The reported correlation values suggest a weak positive relationship between cyclomatic complexity and the compilation success rate. From a methodological perspective, this supports the notion that cyclomatic complexity alone does not adequately explain the syntactic success of test suites generated by LLMs. Although this study did not formally evaluate other factors quantitatively, empirical observations during the experiment indicated that certain structural characteristics of the code, such as the restrictive encapsulation of constants or incompatible type casting, may negatively impact the generation of syntactically valid tests. This is, however, an exploratory hypothesis specific to this experiment, which may be further investigated in future studies focusing on internal class properties.

5.1.3 Graphical Analysis. Figure 2 shows the dispersion between the cyclomatic complexity of the classes and the compilation success rate of the test suites generated by each LLM.

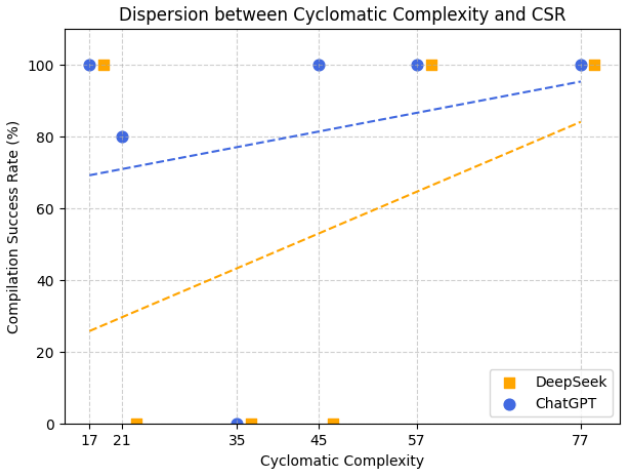

Figure 2: Dispersion between cyclomatic complexity and compilation success rate for ChatGPT and DeepSeek.

As evidenced in Figure 2, the trend lines indicate weak positive correlations. However, the chart reveals complementary nuances: ChatGPT exhibits relatively stable behavior, maintaining high compilation success rates even for classes with higher complexity, such as `Range` (CC = 57) and `Fraction` (CC = 77).

DeepSeek, on the other hand, shows a more irregular pattern, achieving full success at extreme levels of complexity, such as

[3]https://docs.scipy.org/doc/scipy/reference/stats.html

`Primes` (CC = 17) and `Fraction` (CC = 77), but failing completely in classes with intermediate complexity values. The dispersion of data points, particularly in the case of DeepSeek, visually reinforces the lack of a consistent relationship between complexity and model performance, suggesting the influence of other structural aspects of the code.

RQ1 Answer: The results do *not* indicate a direct relationship between cyclomatic complexity (CC) and the compilation success of the tests. Despite 100% compilation in high-complexity classes (`Fraction`, CC=77 and `Range`, CC=57), failures occurred in intermediate-complexity classes.

5.2 RQ2: What are the reasons behind the runtime failures of test suites generated by LLMs that compile correctly?

The results obtained for RQ1 indicate that cyclomatic complexity alone does not account for the compilation success of the test suites generated by the evaluated LLMs. This finding suggests that other structural characteristics of the code may have a more substantial influence on the syntactic validity of the generated tests.

Notably, even among the test suites that compiled successfully, runtime failures were still observed, raising additional concerns regarding their practical reliability. In this context, RQ2 extends the analysis by examining the underlying causes of such execution failures, intending to uncover the semantic limitations of the models. Identifying and categorizing these causes allows for the recognition of recurring patterns, which may, in turn, inform improvements in prompt design and contribute to the development of more effective LLM-based test generation strategies.

5.2.1 Quantitative Assessment. To identify test suites that exhibited runtime failures despite successful compilation, a filtering process was applied to the experimental results. The analysis focused exclusively on cases where at least one test failed during execution, even though the suite compiled without errors.

These findings reveal that, even when syntactically valid, several test suites exhibited execution failures caused by inconsistent test cases. Each model generated 30 suites (5 per class), of which 9 from ChatGPT and 11 from DeepSeek compiled successfully but failed at runtime, forming the subset analyzed to identify semantic limitations associated with the observed errors. This proportion highlights a relevant obstacle to the complete application of the mutation testing technique, as compilation, although necessary, does not guarantee the effectiveness or eligibility of tests generated by LLMs. Although the difference between the models (55% for DeepSeek and 45% for ChatGPT) is not statistically significant, both demonstrated a tendency to produce seemingly valid tests that do not behave correctly at runtime, which underscores the importance of a qualitative investigation into the causes of such failures, as detailed in the following subsection.

5.2.2 Qualitative Analysis. We conducted a qualitative analysis of test suites that compiled successfully but failed at runtime, manually inspecting their error logs and categorizing the failures into two types: *Assertion Failures*, related to incorrect or inverted expectations, and *Runtime Exceptions*, involving issues such as unhandled inputs or invalid type conversions. This categorization supported the construction of a failure catalog aimed at identifying recurring semantic limitations in the generated tests.

ChatGPT. Table 3 presents the catalog of failures identified in the test suites generated by ChatGPT, organized by class, interpreted subcategory, frequency and error type.

Table 3: Summary of interpreted failure subcategories in test suites generated by ChatGPT.

Class	Interpreted Subcategory	Freq.	Category
`Hex`	Invalid cast: byte[] → char[]	2	Runtime Exception
`CharRange`	Comparison failure in negated range	5	Assertion Failure
	Empty iterator in range	2	Assertion Failure
`Fraction`	Incorrect result in arithmetic operation	1	Assertion Failure
	Imprecise numeric type conversion	1	Assertion Failure

The failure patterns observed in the test suites generated by ChatGPT highlight semantic limitations that compromise correct runtime behavior. In the `Hex` class, type conversion errors, such as invalid casts between byte arrays and character arrays, indicate difficulties in inferring appropriate data representations for specific structures. For the `CharRange` class, the high incidence of failures related to negated range comparisons and empty iterators reveals a limited understanding of conditional logic involving inclusion and exclusion, which affects the generation of coherent and meaningful assertions. In the case of the `Fraction` class, errors stemming from incorrect arithmetic results and imprecise type conversions point to weaknesses in numerical reasoning and the proper use of class constructors.

Together, these patterns suggest that, despite producing syntactically correct code, the model struggles to semantically interpret the functional behavior of certain classes, especially those involving complex logic or domain-specific operations, thereby limiting the reliability of the generated test suites in practice.

DeepSeek. Table 4 summarizes the failures identified in the test suites generated by the DeepSeek model, categorized by class, interpreted subcategory, frequency, and error type.

As the data indicate, the test suites produced by DeepSeek exhibited a broad spectrum of semantic issues, with Assertion Failures representing the most frequent failure category. The `Range` class, in particular, concentrated eight cases involving incorrect ordering logic, such as inverted outcomes in comparison methods, pointing to a consistent difficulty in reasoning about relational conditions. Similarly, the `Fraction` class presented both assertion and runtime failures, especially in scenarios involving arithmetic overflows, extreme input values, and invalid type conversions. For the `Primes` class, failures were primarily caused by the absence of proper input validation, particularly in boundary cases, leading to unhandled exceptions.

When compared to ChatGPT, DeepSeek exhibited a greater diversity of failure types, suggesting a broader syntactic generation capacity. However, this broader coverage appears to come at the expense of semantic precision, as evidenced by the higher frequency of incorrect or inconsistent test behavior during execution.

Table 4: Summary of interpreted failure subcategories in test suites generated by DeepSeek.

Class	Interpreted Subcategory	Freq.	Category
`Primes`	Incorrect result in expected value test	2	Assertion Failure
	Invalid input not handled	1	Runtime Exception
	Negative input without verification	1	Runtime Exception
`Range`	Null comparator used in comparison	1	Assertion Failure
	Inverted result in comparison method	8	Assertion Failure
	Unexpected exception type	1	Runtime Exception
	Incorrect textual representation	1	Assertion Failure
`Fraction`	Incorrect textual representation	1	Assertion Failure
	Failed conversion to primitive type	1	Runtime Exception
	Uncaught exception in float-based constructor	1	Runtime Exception
	Unhandled overflow in addition	2	Assertion Failure
	Division by zero not handled	1	Assertion Failure
	Overflow in extreme value conversion	1	Runtime Exception
	Unhandled overflow in multiplication	1	Assertion Failure
	Overflow in fraction constructor with extreme values	1	Runtime Exception

RQ2 Answer: Runtime failures primarily stemmed from semantic limitations of the models in interpreting the classes, with the following main issues observed: (i) inverted assertions in comparisons (e.g., `compareTo`, `isBefore`); (ii) lack of handling for invalid inputs; (iii) use of null comparators; (iv) unexpected textual formats; and (v) absence of arithmetic overflow checks.

5.3 RQ3: To what extent can LLMs generate high-quality unit tests that effectively maximize coverage and detect faults in source code?

Table 5 summarizes the application of the mutation testing technique to the test suites generated by the LLMs. The results are organized by increasing cyclomatic complexity (CC) and include: the number of generated mutants, the LLM used, the average Mutation Coverage (MC) and Mutation Score (MS) with their respective standard deviations ($\mu \pm \sigma$), and the number of test suites considered eligible out of the five generated per class by each model. Cases in which the test suites did not compile or failed during execution are marked as "Not Applicable".

Table 5: Mutation testing summary for eligible test suites.

CC	Class	Mutants	LLM	MC ($\mu \pm \sigma$)	MS ($\mu \pm \sigma$)	Eligible TSs
17	Primes	32	ChatGPT	76.88% ± 11.44%	85.61% ± 3.49%	5/5
			DeepSeek	98.44% ± 1.56%	90.47% ± 0.16%	2/5
21	Hex	35	ChatGPT	100% ± 0%	100% ± 0%	2/5
			DeepSeek	*Not Applicable*	*Not Applicable*	0/5
35	BinaryCodec	*Not Applicable*	ChatGPT	*Not Applicable*	*Not Applicable*	0/5
			DeepSeek	*Not Applicable*	*Not Applicable*	0/5
45	CharRange	*Not Applicable*	ChatGPT	*Not Applicable*	*Not Applicable*	0/5
			DeepSeek	*Not Applicable*	*Not Applicable*	0/5
57	Range	94	ChatGPT	85.74% ± 5.62%	84.24% ± 2.15%	5/5
			DeepSeek	96.81% ± 2.13%	86.22% ± 1.95%	2/5
77	Fraction	140	ChatGPT	67.38% ± 15.18%	80.63% ± 2.89%	3/5
			DeepSeek	*Not Applicable*	*Not Applicable*	0/5

As shown in Table 5, classes with higher cyclomatic complexity tend to have more generated mutants. This is because mutation testing introduces changes mainly at decision points in the code, such as conditional operators and control structures, which are more frequent in logically complex code [12]. Again, mutant generation was performed independently from the LLM-generated test suites using the PIT (Pitest) tool to ensure all test suites were evaluated against the same set and number of mutants, providing a fair basis for comparison between the models.

To answer RQ3, the analysis was structured into three topics: (i) Mutation Coverage – MC, (ii) Mutation Score – MS, (iii) Interpretive Comparison. Each aspect is discussed individually below.

5.3.1 Mutation Coverage - MC.

- In the **Primes** class, DeepSeek achieved a significantly high average coverage, reaching 98.44% ± 1.56%, indicating excellent capability in covering the generated mutants. ChatGPT showed lower values, with an average of 76.88% ± 11.44%, reflecting good coverage, but with less consistency across executions.
- For the **Hex** class, the ChatGPT model achieved perfect coverage (100%), though limited by the low eligibility of its suites (2 out of 5), highlighting that while effective when applicable, its use may be hindered by a lack of stability. DeepSeek had no eligible suites in this class, making its evaluation impossible.
- In **Range**, DeepSeek once again stood out with high and consistent coverage of 96.81% ± 2.13%, clearly outperforming ChatGPT, which recorded 85.74% ± 5.62%. These results reinforce DeepSeek's effectiveness in generating tests that provide broader coverage of the mutant space.
- In the **Fraction** class, only ChatGPT had eligible suites, achieving a moderate coverage of 67.38% ± 15.18%. This result reveals considerable variability and suggests limitations in the breadth of the tests generated by the model for this specific class.

It is worth noting that the classes **BinaryCodec** and **CharRange** did not yield any eligible test suites from either of the evaluated models, which prevented the analysis of MC in these cases. This limitation suggests additional challenges for the practical application of these models in contexts with more complex or specific characteristics.

An interesting pattern emerges when comparing the results for the Hex and Fraction classes. In both cases, DeepSeek failed to generate any eligible test suites, while ChatGPT was able to produce valid executions, albeit with significant variability, especially in Fraction. This suggests that DeepSeek demonstrates high precision when it succeeds, as reflected by its stable and high MC scores in other classes. However, its applicability appears more limited across different classes, particularly in scenarios involving encoding or mathematical logic as the ones tested. These findings indicate that DeepSeek's effectiveness may be constrained by the structural or domain-specific characteristics of the target class, whereas ChatGPT, despite being less stable, tends to provide broader applicability across classes. This result is confirmed by the result of a t-test (p-value = 0.04) comparing the mutation score (MS) metric of each LLM, i.e., there is a statistically significant difference in the MS values reported with DeepSeek having higher values, although in a smaller number of classes.

5.3.2 Mutation Score - MS.

- In the **Primes** class, both models achieved good results, but DeepSeek performed better with 90.47% ± 0.16%, surpassing ChatGPT, which reached 85.61% ± 3.49%. The low variance in DeepSeek's results demonstrates its consistency in mutant detection, while ChatGPT, although effective, showed slight fluctuation.
- In **Hex**, only ChatGPT generated eligible suites, achieving a perfect MS of 100% ± 0%. This indicates that when valid tests are produced, the model is capable of detecting all inserted mutants in this class. On the other hand, the absence of eligible suites from DeepSeek made its evaluation in this context unfeasible.
- The **Range** class presented a scenario similar to what was observed for MC. DeepSeek achieved 86.22% ± 1.95%, once again outperforming ChatGPT, which obtained 84.24% ± 2.15%. Both models showed low dispersion, indicating good stability in mutant elimination.
- In **Fraction**, only ChatGPT had eligible suites, with an MS of 80.63% ± 2.89%. Although this value is considered high, the lack of data from DeepSeek limits direct comparisons in this class.
- The remaining classes, **BinaryCodec** and **CharRange**, did not yield any eligible test suites from either model, making the analysis of MS unfeasible.

In general, a t-test comparing the reported MS metrics *for all classes* did not identify any statistical difference between ChatGPT and DeepSeek with the reported p-value=0.5. However, the results suggest that DeepSeek outperformed ChatGPT in terms of mutant elimination *whenever it could be evaluated*. The consistency observed in its low standard deviations indicates less variability between executions, which is especially important for automated testing scenarios in production environments. On the other hand, ChatGPT stood out in terms of broader applicability, generating more eligible test suites across the analyzed classes, albeit with greater performance variation.

In the context of this study, the *Mutation Score* (MS) metric serves to simulate the defect detection capability [12]. Thus, the classes for which the LLMs achieved high MS values, such as `Primes` and `Hex` with ChatGPT, can be interpreted as examples of effective test generation. This interpretation aligns with empirical evidence from the literature. For instance, McConnell [19] reports that programs with lower cyclomatic complexity tend to have fewer defects in production. Although this refers to defect incidence rather than test generation, it highlights the relevance of structural code characteristics, such as complexity, as factors that influence software quality. In this sense, mutation-based metrics (like MS) and complexity-based indicators can be considered complementary when assessing the effectiveness of automated test generation.

5.3.3 Comparative and Interpretive Analysis of the Results. The Mutation Coverage (MC) and Mutation Score (MS) data presented in Table 5 suggests that the effectiveness of the tests generated by the LLMs varies non-linearly in relation to the cyclomatic complexity of the classes. Classes with varying complexity levels yielded contrasting results, depending on the model used.

The **DeepSeek** model demonstrated high performance in the `Primes` (CC = 17) and `Range` (CC = 57) classes, achieving high MC values (98.44% and 96.81%, respectively) and MS values (90.47% and 86.22%), with low variability between executions. This pattern suggests that, regardless of the absolute complexity value, when the model can generate valid suites, the tests tend to be robust and accurate. However, for the remaining classes, DeepSeek was unable to generate eligible suites. A possible explanation for DeepSeek's failure to generate eligible test suites in some classes could be related to factors not captured by cyclomatic complexity. Although no explicit analysis of structural code characteristics was conducted in this study, this limitation provides an opportunity for future work to investigate additional factors that may affect LLM performance.

In contrast, **ChatGPT** achieved broader class coverage, managing to generate eligible suites in five out of the six evaluated classes, including the class with the highest cyclomatic complexity in the experiment, `Fraction` (CC = 77). Nonetheless, the quality of the generated suites was more unstable. Although it achieved 100% MC and MS in `Hex` (CC = 21), it showed high variability in other classes such as `Fraction`, with MC of 67.38% ± 15.18%. This variation suggests that, although the model is more adaptable, its effectiveness in mutant elimination strongly depends on the specific class. In summary, when DeepSeek can generate test suites, it performs better than ChatGPT; however, DeepSeek is not as effective as ChatGPT.

Overall, no direct correlation is observed between the absolute value of cyclomatic complexity and the effectiveness of the generated tests. For example, both `Primes` (CC = 17) and `Range` (CC = 57) achieved high MC and MS scores. In contrast, classes with intermediate complexity, such as `Hex` (CC = 21) and `BinaryCodec` (CC = 35), failed to generate valid test cases in at least one of the models. These results suggest that, although cyclomatic complexity reflects the number of independent paths through control structures, it alone does not determine test effectiveness. Structural factors, such as assertion failures, invalid inputs and incompatible cast attempts, appear to directly impact the LLMs' ability to comprehend the logic and structure of the code and generate effective tests.

RQ3 Answer: DeepSeek presented greater effectiveness in generating effective tests, with higher *Mutation Coverage* (MC) and *Mutation Score* (MS) values, but was only applicable to a limited number of classes. In contrast, ChatGPT showed broader applicability, generating tests for more classes, although with more variation in the results.

6 Discussion

Practical implications and lessons learned

This study reinforces the potential of LLMs to support unit test automation workflows. However, the findings also reveal significant limitations, particularly in interpreting complex logic, handling invalid inputs, and constructing reliable assertions. While mutation scores were high in several classes, runtime failures in syntactically valid suites reveal that test generation must go beyond compilation success.

An important takeaway is that cyclomatic complexity alone is not a reliable predictor of test success. Instead, structural and

semantic factors, such as type casting, null handling, and domain-specific operations, appear to influence the effectiveness of LLM-generated tests.

From a tooling perspective, DeepSeek proved more consistent when successful, producing high mutation scores with low variance. However, its applicability was more restricted. ChatGPT, in contrast, generated valid suites for more classes, though with higher variability in effectiveness. These complementary strengths suggest a hybrid approach could be beneficial in practice.

Implications for software testing teams.

For practitioners, LLMs can serve as assistive tools, accelerating the creation of initial test suites, especially for less complex components. Still, due to the possibility of runtime failures, it is crucial to implement validation layers (e.g., static analysis, mutation-based assessment) before relying on generated tests in production.

A hybrid testing workflow combining LLM-generated tests with manual review and automated validation may offer the best balance between productivity and reliability. Teams should also invest in prompt engineering strategies that explicitly guide the models toward safe input handling and expected behaviors.

7 Threats to Validity

Construct Validity: In this study, the effectiveness of LLMs was evaluated using the metrics *Mutation Coverage* (MC) and *Mutation Score* (MS), which are well-established in the literature as indicators of test suite robustness[12]. To mitigate potential biases, these metrics were applied exclusively to test suites deemed eligible for testing. Additionally, repeated prompt executions were performed to minimize the impact of the inherent nondeterminism of LLMs, aligning with best practices described in recent studies [24]. However, it is acknowledged that other complementary metrics, such as instruction coverage or execution time, could expand the analysis.

Internal Validity: To ensure a fair comparison between LLMs, all tests were generated from structured zero-shot prompts without prior examples and applied to the same classes using the same set of mutants, previously generated via PIT. The minimal test suites used only to activate the mutants were excluded from any analysis to avoid contaminating the results. Nevertheless, limitations may have arisen from the LLMs' interpretation of the prompts, potentially introducing subtle variations in execution that could have affected the results.

External Validity: The study used six classes extracted from different projects in the Defects4J repository, covering various domains and levels of cyclomatic complexity. However, the number of classes is limited, and the exclusive focus on unit tests for the Java programming language restricts the generalization of findings to other types of tests, languages, or architectures. The use of only two LLMs (ChatGPT and DeepSeek) also limits the generalization of the results to models with different architectures or training data.

Conclusion Validity: The conclusions were based on both quantitative (mean and standard deviation) and qualitative analyses, applied according to each research question. RQ1 was evaluated using the Compilation Success Rate (CSR) and Spearman's correlation (ρ), which is appropriate for small sample sizes. Next, RQ2 relied on the manual categorization of error logs from suites that failed at runtime, revealing recurring patterns. Finally, for RQ3, the metrics Mutation Coverage (MC) and Mutation Score (MS) were used, derived from five executions per class. Despite the possibility of bias in the qualitative analysis, the findings were consistent and clearly bounded within the scope of the experiment, thus avoiding unwarranted generalizations.

This study acknowledges that the presence of *equivalent mutants* may affect the absolute values of the *Mutation Score* [7]. Importantly, this effect applies equally to both evaluated models, as all test suites were executed against the same fixed set of mutants per target class. Thus, while equivalent mutants may influence the absolute Mutation Score, the experimental conditions ensure a fair and consistent basis for relative comparison between the LLMs.

8 Conclusion and Future Work

This study assessed the effectiveness of ChatGPT (GPT-4o) and DeepSeek V3 in generating unit tests for six Java classes with varying levels of cyclomatic complexity from the Defects4J dataset. For each class, a structured prompt was executed five times, yielding distinct test suites composed of multiple test cases. These suites were then evaluated using the PIT mutation testing tool. The analysis encompassed Compilation Success Rate (CSR), runtime failure classification, Mutation Coverage (MC), and Mutation Score (MS), providing a comprehensive assessment of syntactic validity, semantic reliability, and fault detection capability.

For RQ1, ChatGPT achieved a higher average compilation success rate (80.00%) than DeepSeek (50.00%), indicating better syntactic validity. For RQ2, the analysis of 20 runtime-failing suites (9 from ChatGPT, 11 from DeepSeek) revealed recurring semantic issues, including incorrect assertions and unhandled exceptions. These results highlight the importance of providing clearer prompt instructions, including input constraints and expected behaviors, to improve test robustness.

Finally, as for RQ3, both models generated tests that successfully killed mutants, with MS values above 84% in the best cases—e.g., DeepSeek reached 90.47% (±0.16) in `Primes`, and ChatGPT achieved 84.24% (±2.15) in `Range`. DeepSeek showed greater stability, while ChatGPT demonstrated broader applicability, producing eligible suites in 4 out of 6 classes (vs. 2 for DeepSeek).

This study provides evidence on LLM behavior in automated unit test generation using mutation testing and presents a reproducible methodology with multiple executions. Future work includes expanding the experiment with an automated pipeline to increase the number of target classes, and, more importantly, exploring alternative prompt strategies [9] to assess their impact on test quality.

ARTIFACT AVAILABILITY

All experimental artifacts are publicly available at Zenodo: https://doi.org/10.5281/zenodo.17250744.

ACKNOWLEDGMENTS

We thank CNPq (420406/2023-9 and 308101/2025-1) for funding this research.

REFERENCES

[1] Paul Ammann and Jeff Offutt. 2016. *Introduction to Software Testing:* (2 ed.). Cambridge University Press. doi:10.1017/9781316771273

[2] Vahit Bayrı and Ece Demirel. 2023. AI-Powered Software Testing: The Impact of Large Language Models on Testing Methodologies. In *2023 4th International Informatics and Software Engineering Conference (IISEC)*. IEEE, Ankara, Turkiye, 1–4. doi:10.1109/IISEC59749.2023.10391027

[3] Rajiv Chopra. 2018. *Software Testing: A Self-Teaching Introduction* (1st ed ed.). Mercury Learning & Information, Bloomfield.

[4] Henry Coles and contributors. 2025. PIT: Mutation Testing for Java. https://pitest.org. Accessed: 13 April 2025.

[5] Zheyuan Cui, Mert Demirer, Sonia Jaffe, Leon Musolff, Sida Peng, and Tobias Salz. 2024. The Effects of Generative AI on High Skilled Work: Evidence from Three Field Experiments with Software Developers. *SSRN eLibrary* (2024). doi:10.2139/ssrn.4945566

[6] DEEPSEEK. 2024. Introducing DeepSeek-V3. https://api-docs.deepseek.com/news/news1226. Acesso em: 27 fev. 2025.

[7] Márcio Eduardo Delamaro, José Carlos Maldonado, and Mario Jino. 2007. *Introdução ao Teste de Software* (4ª tiragem ed.). Elsevier Editora Ltda., Rio de Janeiro, Brasil.

[8] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (April 1978), 34–41. doi:10.1109/C-M.1978.218136

[9] Ionut Daniel Fagadau, Leonardo Mariani, Daniela Micucci, and Oliviero Riganelli. 2024. Analyzing Prompt Influence on Automated Method Generation: An Empirical Study with Copilot. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension* (Lisbon, Portugal) *(ICPC '24)*. Association for Computing Machinery, New York, NY, USA, 24–34. doi:10.1145/3643916.3644409

[10] Soneya Binta Hossain and Matthew B. Dwyer. 2025. TOGLL: Correct and Strong Test Oracle Generation with LLMS. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 1475–1487. doi:10.1109/ICSE55347.2025.00098

[11] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* 33, 8, Article 220 (Dec. 2024), 79 pages. doi:10.1145/3695988

[12] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (Sept. 2011), 649–678. doi:10.1109/TSE.2010.62

[13] JUnit. 2025. *JUnit 5 User Guide*. Disponível em: https://junit.org/junit5/docs/current/user-guide/. Acesso em: 2 mar. 2025..

[14] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Tanaka. 2022. Large Language Models are Zero-Shot Reasoners. *arXiv preprint arXiv:2205.11916* (2022). https://arxiv.org/abs/2205.11916

[15] Thomas Laurent and Anthony Ventresque. 2019. PIT-HOM: An Extension of Pitest for Higher Order Mutation Analysis. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 82–89. doi:10.1109/ICSTW.2019.00036

[16] Kefan Li and Yuan Yuan. 2024. Large Language Models as Test Case Generators: Performance Evaluation and Enhancement. arXiv preprint arXiv:2404.13340. arXiv:2404.13340 [cs.SE] https://arxiv.org/abs/2404.13340.

[17] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. 2023. On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2149–2160. doi:10.1109/ICSE48619.2023.00181

[18] Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320. doi:10.1109/TSE.1976.233837

[19] Steve McConnell. 2004. *Code Complete: A Practical Handbook of Software Construction* (2nd ed.). Microsoft Press, Redmond, WA.

[20] Ali Mili and Fairouz Tchier. 2015. *Software Testing: Concepts and Operations*. John Wiley & Sons, Inc.

[21] Zifan Nan, Zhaoqiang Guo, Kui Liu, and Xin Xia. 2025. Test Intention Guided LLM-Based Unit Test Generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 1026–1038. doi:10.1109/ICSE55347.2025.00243

[22] Nhan Nguyen and Sarah Nadi. 2022. An Empirical Evaluation of GitHub Copilot's Code Suggestions. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 1–5. doi:10.1145/3524842.3528470

[23] OPENAI. 2024. Hello GPT-4o. https://openai.com/index/hello-gpt-4o/. Acesso em: 27 fev. 2025.

[24] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. 2023. LLM is Like a Box of Chocolates: the Non-determinism of ChatGPT in Code Generation. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. https://arxiv.org/abs/2308.02828 arXiv:2308.02828.

[25] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. 2018. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 130–140. doi:10.1109/SANER.2018.8330203 arXiv:1801.06393 [cs].

[26] SonarSource. 2025. Code Metrics - SonarQube Documentation. https://docs.sonarsource.com/sonarqube/latest/user-guide/code-metrics/metrics-definition/. Accessed: 13 April 2025.

[27] Zhao Tian, Honglin Shu, Dong Wang, Xuejie Cao, Yasutaka Kamei, and Junjie Chen. 2024. Large Language Models for Equivalent Mutant Detection: How Far Are We?. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)* (Vienna, Austria). ACM, to appear. doi:10.1145/3650212.3680395

[28] Frank Tip, Jonathan Bell, and Max Schäfer. 2024. LLMorpheus: Mutation Testing using Large Language Models. doi:10.48550/arXiv.2404.09952 arXiv:2404.09952 [cs].

[29] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering* 50, 4 (April 2024), 911–936. doi:10.1109/TSE.2024.3368208

[30] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. In *Proceedings of the [Conference acronym]*. ACM, New York, NY, USA. arXiv:2408.11324 [cs.SE] https://arxiv.org/abs/2408.11324 To appear.

[31] Tao Xiao, Hideaki Hata, Christoph Treude, and Kenichi Matsumoto. 2024. Generative AI for Pull Request Descriptions: Adoption, Impact, and Developer Interventions. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), Article 47. doi:10.1145/3643773

[32] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. 2024. On the Evaluation of Large Language Models in Unit Test Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1607–1619. doi:10.1145/3691620.3695529

[33] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. Assessing the quality of GitHub copilot's code generation. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering* (Singapore, Singapore) *(PROMISE 2022)*. Association for Computing Machinery, New York, NY, USA, 62–71. doi:10.1145/3558489.3559072

[34] Gaolei Yi, Zizhao Chen, Zhenyu Chen, W. Eric Wong, and Nicholas Chau. 2023. Exploring the Capability of ChatGPT in Test Generation. In *Proceedings of the 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*. IEEE, Chiang Mai, Thailand, 72–80. doi:10.1109/QRS-C60940.2023.00013

[35] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and Improving ChatGPT for Unit Test Generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 76:1–76:24. doi:10.1145/3660783 Publication date: July 2024.

[36] Jerrold H. Zar. 2010. *Biostatistical Analysis* (5th ed.). Prentice Hall, Upper Saddle River, NJ.