Does Prompt Engineering Really Impact LLM-Generated Unit Tests? An Empirical Study Based on Mutation Testing

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

Christian Morais
Federal University of Pará - UFPA
Belém, Brazil
christian.morais@icen.ufpa.br

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

Abstract

Large Language Models (LLMs) have been increasingly adopted to support unit test generation, but their nondeterministic behavior and sensitivity to instruction quality make prompt engineering a central factor in this process. However, a fundamental question remains: are these tests truly capable of detecting faults? This paper presents an empirical study on the influence of different prompt engineering strategies on LLM-based unit test generation. Unlike prior work, this study combines an observational, single-pass fault-detection evaluation with the integration of data-flow testing theory into prompt design, deriving concrete recommendations from a diagnostic analysis of surviving mutants. Three LLMs were employed to generate tests for 16 Java classes from the Defects4J and TestBench datasets, using five prompting strategies: Zero-Shot, One-Shot, Few-Shot, Program of Thoughts (PoT), and Chain of Code (CoC). To mitigate nondeterminism, each combination was executed 5 times, yielding a total of 1,200 generated test suites. The results showed no statistically significant differences in the influence of prompting strategies on either test suite eligibility (i.e., the proportion of suites that both compiled and executed successfully) or effectiveness under mutation testing; however, descriptive analyses indicated better performance for structured reasoning-based approaches, particularly PoT and CoC. Evidence from surviving mutants revealed recurrent weaknesses in the generated suites, particularly with respect to boundary values, alternative branches, and assertions with values independent of the code under test. Together, these findings suggest that the contribution of prompt engineering goes beyond producing executable tests, also affecting the logical quality of the generated artifacts and offering concrete directions for their improvement.

Keywords

Empirical Study, Large Language Models, Unit Testing, Prompt Engineering Strategies, Mutation Testing

1 Introduction

Software development has been transformed by the incorporation of Generative Artificial Intelligence (GenAI), particularly Large Language Models (LLMs), which have begun to support activities across different stages of the development life cycle [13]. In this context, the development process is no longer focused solely on manually writing code but also involves the supervision, refinement, and orchestration of artifacts generated by these models [15, 36]. Consequently, software testing becomes even more relevant, being essential for verifying the reliability and correctness of applications, especially given the potential for LLMs to introduce inconsistencies and vulnerabilities [2, 4].

In the context of unit testing, LLMs emerge as a promising alternative to manual test creation, which is recognized as a costly and cognitively demanding activity [34, 44], due to their ability to generate tests at scale and with greater operational capacity [14]. On the other hand, these models exhibit nondeterministic behavior, with significant variations across successive executions and sensitivity to the contextual information provided as input [28]. Such behavior poses challenges for the systematic evaluation of the quality of generated tests, motivating investigations that account for both model variability and the structure of the provided context.

Prompt engineering emerges as a central factor for achieving more consistent results, since the clarity, organization, and level of detail of the instructions directly influence the quality of the generated artifacts [3, 17]. In this direction, several investigations have been analyzing the use of LLMs to generate tests that, in many cases, are compilable and achieve substantial coverage [16, 32, 34]. Nevertheless, a fundamental question remains: are these tests actually capable of revealing faults? Empirical evidence shows that high coverage levels do not necessarily imply greater fault detection capability, and that the absence of semantically meaningful assertions can compromise the usefulness of the generated tests [18, 19]. Despite recent advances in the use of LLMs for software test automation, important gaps remain in understanding the quality of the generated artifacts, and empirical investigations are needed to analyze how different prompting strategies affect both the practical usability of the produced tests and their fault-detection capabilities. In this regard, it is essential to advance toward evaluations guided by more rigorous criteria, such as those provided by mutation testing.

Given this scenario, mutation testing stands out as a widely recognized, robust approach for evaluating the quality of a test suite, as it provides more concrete evidence of its fault-detection capability [11, 45]. The technique is based on generating mutants, versions of the original program with small syntactic modifications, whose detection by test cases allows estimation of the effectiveness of the evaluated test suite [11]. Thus, mutation testing complements purely structural metrics, offering a fault-revealing perspective.

This paper presents an empirical study of the influence of different prompt engineering strategies on LLM-based unit test generation, evaluating the quality of the generated artifacts in terms of compilation, execution, and fault-detection effectiveness using mutation testing. To this end, three LLMs were applied to 16 Java classes selected from the Defects4J [1, 39] and TestBench [20, 46] datasets under five prompting strategies: Zero-Shot, One-Shot, Few-Shot, Program of Thoughts (PoT), and Chain of Code (CoC). To mitigate the inherent nondeterminism of these models, each combination of class, model, and strategy was executed independently

across multiple replications. The main contribution of this study lies not in the use of mutation testing per se, but in the combination of three aspects not jointly identified in the literature: (i) an observational, fault-detection-oriented evaluation conducted under a strict single-pass regime, without re-prompting or suite repair; (ii) the integration of data-flow testing theory (def-use, p-use, and c-use associations) [11] directly into the design of PoT and CoC prompts; and (iii) the derivation of concrete recommendations from a diagnostic analysis of surviving mutants, rather than using them as iterative feedback for the generation process itself. The study was guided by the following research questions:

***RQ1*:** What is the influence of prompting strategies on the eligibility rate of test suites generated by LLMs?

***RQ2*:** To what extent do prompting strategies contribute to the generation of more effective test suites according to mutation testing metrics?

***RQ3*:** What types of adjustments to test suites can be inferred from the analysis of surviving mutants?

The research questions were organized in a progressive and complementary manner. RQ1 analyzes the generated test suites with respect to their successful compilation and execution, characterizing them as eligible for the mutation testing stage. RQ2 evaluates the effectiveness of these eligible suites in fault detection using mutation testing metrics. Finally, RQ3 examines the most recurrent surviving mutants, through which the authors conduct a diagnostic analysis and derive practical recommendations for structural adjustments to strengthen the fault-detection capability of the test suites produced by the LLMs.

Overall, the results indicate that the influence of prompting strategies on LLM-based unit test generation is less direct than often assumed. Although no statistically significant differences were observed in either suite eligibility or mutation-testing effectiveness, descriptive analyses indicated a trend toward better performance for structured reasoning approaches, such as PoT and CoC. Furthermore, the analysis of surviving mutants revealed recurrent weaknesses in the generated suites, particularly with respect to boundary values, alternative branches, and independent assertions, thereby enabling objective recommendations for their refinement. These findings suggest that the contribution of prompt engineering extends beyond generating executable tests to encompass the logical quality of the produced artifacts and the pathways for their improvement.

The remainder of this paper is organized as follows. Section 2 presents the theoretical background of the study. Section 4 describes the experimental methodology. Section 3 discusses related work. Section 5 presents the results, Section 6 discusses the lessons learned, and Section 7 presents the practical implications. Section 8 discusses threats to validity. Finally, Section 9 presents the conclusion and future work.

2 Background

2.1 LLMs and Prompt Engineering

Large Language Models (LLMs) constitute a class of language models based on the Transformer architecture, trained at scale on diverse textual datasets, which endows them with the ability to understand and generate software artifacts exhibiting high semantic coherence [4]. In software testing, these models have been investigated as assistants for a diverse range of activities, including unit test case generation, test oracle generation, system test input generation, fault analysis, program repair, and debugging [47]. Corroborating this scenario, recent empirical analyses have established LLMs as a central approach to test automation, demonstrating productivity gains, increased coverage, and greater fault-detection capability compared to manual approaches [31, 47].

Interaction with LLMs can occur through different mechanisms, such as API calls, Retrieval-Augmented Generation (RAG), function calling, fine-tuning, and primarily through prompts, in which instructions are provided to guide the model's behavior, impose constraints, and shape the generated responses generation [5, 35]. Prompt engineering has established itself as an essential discipline, understood as the iterative process of designing and optimizing these inputs to maximize LLM performance on specific tasks [5, 10]. The quality and structure of prompts directly influence the accuracy of the generated outputs, such that small semantic or structural variations in instructions can produce significant differences in the resulting artifacts, especially in complex domains such as software test generation [35, 41].

Various prompting strategies have been proposed to improve the performance of LLMs. The Zero-Shot approach instructs the model to perform tasks without examples, relying on its prior knowledge [33]; One-Shot provides exactly one input-output example along with the task [37]; and Few-Shot expands this context with multiple examples, inducing pattern recognition before execution [33]. Program of Thoughts (PoT) guides the model to express reasoning through executable programs, decoupling logical formulation from numerical computation [8]; Chain of Code (CoC), in turn, instructs the model to structure reasoning in the form of code or pseudocode, integrating hybrid execution through a mechanism called LMulator, which simulates semantically complex operations [25].

2.2 Software Testing Techniques

Software testing techniques provide guidelines for designing test cases from different perspectives, including specifications, source code, and tester experience. To operationalize this activity, these techniques define test criteria, i.e., rules that guide the selection of test cases with the greatest potential to reveal faults [11]. Among the main approaches, structural testing, also known as white-box testing, stands out because its requirements derive from the software's internal structure and require the execution of source code elements such as statements, branches, and logical paths [11].

In the context of structural testing, the analysis based on the data flow criterion stands out, as it guides the derivation of test cases from the associations between the point at which a variable is defined and its subsequent uses in the program [11]. This approach is grounded in three central concepts: definitions (*def*), corresponding to occurrences of value assignments to variables; computational uses (*c-use*), references to variables that directly affect the outcome of a computation, associated with nodes of the Control Flow Graph (CFG); and predicate uses (*p-use*), references that influence the program's control flow, such as conditions in decision or loop structures, associated with transitions between graph nodes [11]. These criteria are particularly relevant because

they lie between the basic all-edges criterion and the impractical all-paths criterion, selecting paths that are strongly related to the program's functional behavior. For this reason, they are among the structural criteria with the highest fault-detection power [11].

Among fault-based criteria, mutation testing stands out as a quality criterion that uses typical implementation faults as the basis for defining test requirements [11]. The technique consists of introducing small modifications into the source code of the original program, generating mutants that simulate plausible faults. The quality of the test suite is then assessed by its ability to distinguish the behavior of the original program from that of the mutants, i.e., to kill the incorrect versions [11]. Its effectiveness is measured through the mutation score, calculated as the ratio between the number of killed mutants and the total number of non-equivalent mutants [11]. Although it is considered one of the most effective criteria for test evaluation, its use involves practical challenges, such as high computational cost and the presence of equivalent mutants, i.e., program versions that preserve the observable behavior of the original and therefore cannot be killed [11].

3 Related Work

To overcome common syntactic and semantic flaws in automatic test generation, Konstantinou et al. [23] proposed YATE, an automated test repair approach. This tool combines static analysis and re-prompting to iteratively correct outputs produced by LLMs, addressing issues ranging from invalid imports to constructor problems. Evaluated on real-world Java projects, YATE demonstrated that recovering the test suite and feeding it back to the model increases line coverage by approximately 32% and mutant detection by 21%. The study concluded that direct generation may be insufficient to meet industry's robustness requirements, validating the adoption of structured post-processing workflows to mitigate output hallucinations.

In contrast, to overcome the limitations of metrics that focus solely on coverage, Wang et al. [42] presented MUTGEN, a framework that directly uses mutation testing to guide the agent during test suite generation. Operating directly within the generation process, the tool uses surviving mutants as feedback providers, which are iteratively processed in the prompt for reassessment. Using this continuous refinement technique, a Mutation Score of 89.5% was achieved on the HumanEval-Java benchmark, surpassing static models evaluated under the same criterion. The authors showed that test generation guided by active correction of semantic defects also optimizes structural effectiveness as a positive byproduct, while maintaining practical feasibility, with processing costs competitive with, or even lower than, traditional search-based approaches.

Additionally, Walczak et al. [41] investigated the impact of code context level and prompting strategies on the quality of unit tests generated by general-purpose models. The study evaluated the effectiveness of the Chain-of-Thought (CoT) strategy under varying information constraints, ranging from method signatures only to full implementation with docstrings. The research demonstrated that the amount of context provided is the determining factor for result quality: providing the full implementation consistently yielded the best syntactic performance and fault detection, achieving high Mutation Scores. Moreover, although step-by-step reasoning induction via CoT yielded positive results, such as maximizing branch coverage, the authors showed that the cognitive effort of complex prompting yields only marginal improvements over simple prompting, while requiring considerably higher computational cost in generation time.

In a large-scale study, Ouédraogo et al. [28] compared five prompting strategies for class-level test suite generation against the SBST tool EvoSuite, finding that LLMs achieved higher readability but substantially lower compilation rates; the authors explicitly identify the absence of a fault-detection evaluation, such as mutation testing, as a limitation of their study.

In contrast to the presented studies, this work differs in three complementary aspects. (i) Unlike YATE [23] and MUTGEN [42], which employ iterative refinement through re-prompting or the incorporation of surviving mutants as feedback, this investigation adopts a strict single-pass regime, without re-prompting or suite repair. Furthermore, unlike Ouédraogo et al. [28], who do not assess the fault-detection capability of the generated tests, the evaluation conducted here is oriented toward fault detection through mutation testing. (ii) Unlike the discussed studies, which do not explore the relationship between testing theory and prompt engineering, this study explicitly incorporates elements of the data-flow criterion (def-use, p-use, and c-use associations) [11] directly into the design of the PoT (Program of Thoughts) and CoC (Chain of Code) strategies. (iii) Unlike MUTGEN, which uses surviving mutants as iterative feedback for the generation process itself, this study derives concrete recommendations from a diagnostic analysis of these mutants, treating them as an object of analysis rather than as an automatic correction mechanism. Together, these three aspects constitute the main contribution and originality of this study in the context of LLM-assisted unit test automation.

4 Experimental Setup

This study was structured in three macro-stages: Planning, Execution, and Evaluation, in accordance with experimental guidelines that recommend the definition of the experimental design, the controlled execution of procedures, and the systematic analysis of results [22, 43]. As illustrated in Figure 1, in the Planning stage, the elements that structure the experiment are defined, including the datasets (4.1.1), target classes and methods (4.1.2), prompt engineering strategies (4.1.3), the LLMs used (4.1.4), the mutation testing tool (4.1.5), and script design (4.1.6). In the Execution stage, the generation of test suites (4.2.1), their validation (4.2.2), and the application of mutation testing (4.2.3) are described. Finally, the Evaluation stage presents the metrics and analytical procedures adopted (4.3) to answer the research questions.

4.1 Planning

4.1.1 Datasets. To carry out this work, programs were selected from two repositories recognized in the software testing literature: the Defects4J dataset and the TestBench benchmark, both relevant for providing Java programs suitable for conducting empirical evaluations on unit test generation and analysis [29, 30, 39, 46]. Defects4J, accessed from version 3.0.1 [1], consists of a consolidated repository of real-world Java programs, covering 16 open-source projects

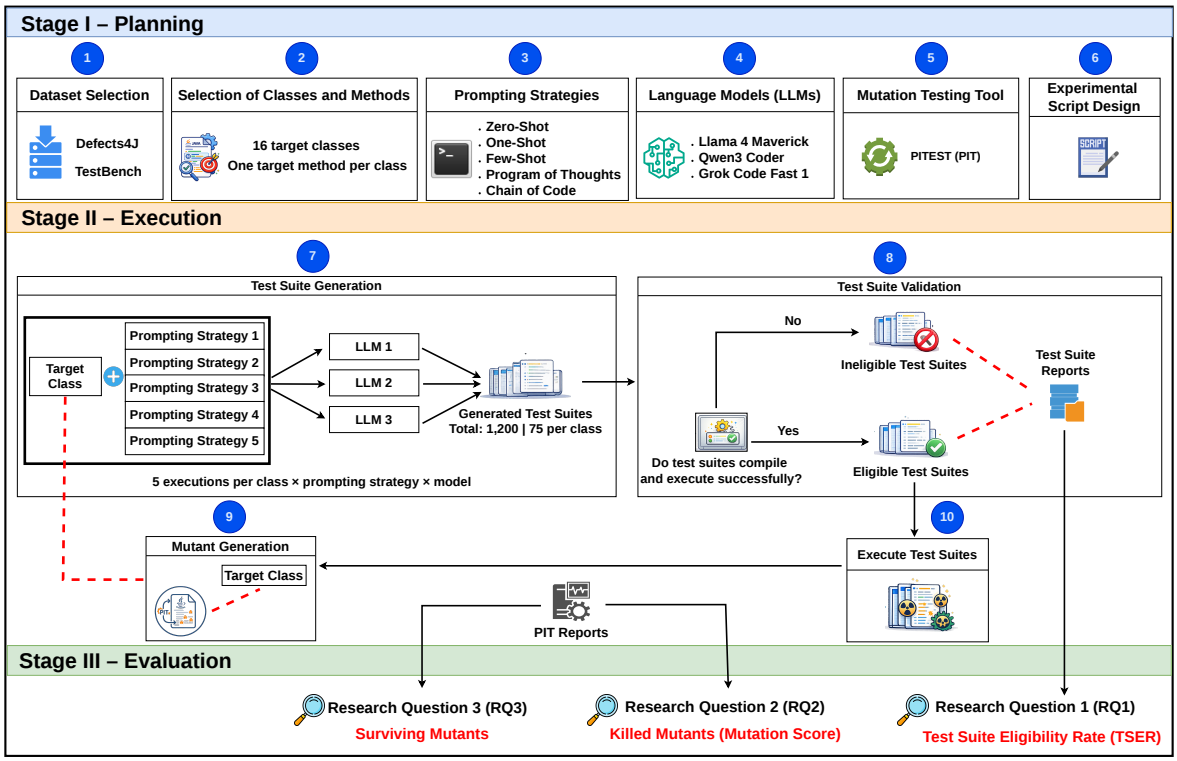

Figure 1: Experimental Flow of the Study

and 854 active bugs organized into faulty and fixed versions, and is widely used in experimental studies in the field [29, 30, 39]. TestBench, in turn, is a class-level benchmark proposed specifically for evaluating LLMs' capability in unit test generation, comprising 108 Java programs extracted from nine real-world open-source projects [20], approximating pragmatic development scenarios [46]. The joint adoption of these repositories aimed to increase the diversity of artifacts and mitigate dependence on a single source of programs. In both repositories, only fixed and stable versions were used, ensuring that test generation and evaluation were conducted on correct implementations free from known faults.

4.1.2 Class and Method Selection. In total, 16 classes were selected, eight from Defects4J and eight from TestBench. The cyclomatic complexity (CC) of the classes was measured using SonarQube, a structural metric that quantifies the logical complexity of code based on its control-flow graph, reflecting the number of independent execution paths [26]. The classes were selected according to the following criteria: (C1) diversity of CC, with values ranging from 10 to 77, corresponding to the minimum and maximum CC observed after measurement with SonarQube, ensuring that the study was not restricted to low-complexity classes; (C2) variety of application domains, including mathematical classes, string manipulation, and data structures; (C3) presence of clear control structures and deterministic outcomes, desirable for evaluation via mutation testing; and (C4) low external coupling, favoring automated test execution without additional instrumentation. For each class, only one method was selected as the test target, reflecting a common practice in unit test planning [6]. The target method was selected according to the following criteria: (M1) functional relevance within the class, avoiding trivial or purely auxiliary methods; (M2) suitability for mutation testing, featuring conditionals, iterations, or arithmetic operations; (M3) low external dependencies, enabling isolated and automated execution; and (M4) verifiable, deterministic output with public visibility, allowing direct validation of test outcomes.

4.1.3 Prompting Strategies. For the study, five prompting strategies were selected: Zero-Shot, One-Shot, Few-Shot, Program of Thoughts (PoT), and Chain of Code (CoC). For each strategy, the instructions were adapted to the specific task of unit test generation while preserving the conceptual principles and reasoning mechanisms originally described in the literature for each approach. This was intended to prevent adaptations required for the software testing domain from mischaracterizing the investigated prompting strategies. Accordingly, Figure 2 conceptually illustrates how the five prompting strategies were structured in the study.

Figure 2 organizes the strategies into two groups based on the level of cognitive guidance provided to the models. In the upper part (low-level cognitive guidance), the example-based strategies Zero-Shot, One-Shot, and Few-Shot operate with a lower level of cognitive guidance: Zero-Shot provides only a direct instruction

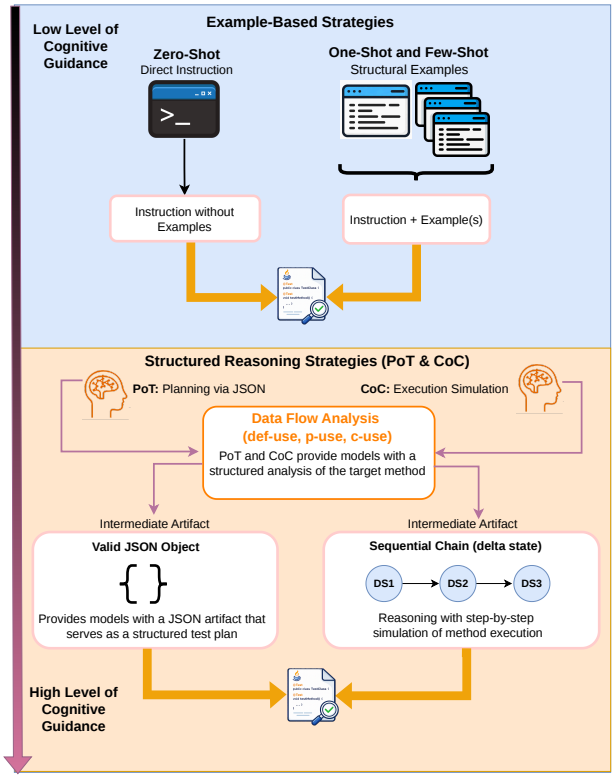

Figure 2: Prompting strategies used in the study.

without examples, whereas One-Shot and Few-Shot add, respectively, one or more structural examples (test cases) as a reference only. In the lower part (high-level cognitive guidance), the structured reasoning strategies PoT and CoC operate with a higher level of cognitive guidance. The prompts used in PoT and CoC incorporated elements of the data flow criterion, such as def, p-use, and c-use associations [11], in the construction of the instructions, unlike the other strategies. PoT and CoC also include a preliminary structural analysis of the target test method to guide models in generating test cases with greater detail. In PoT, this analysis is materialized as an intermediate JSON artifact that structures a test plan based on the identified execution paths. In CoC, the analysis guides the construction of a sequential chain of execution states (DS1 → DS2 → DS3), in which the models simulate the target method's behavior step-by-step before generating test cases.

In summary, the adoption of five strategies aimed to balance analytical coverage and experimental control. Given that the experiment involved different language models, target classes, and repeated executions, including more strategies could substantially increase experimental complexity. Therefore, this number of strategies was chosen to enable systematic comparison of different forms of guidance provided to LLMs while maintaining the feasibility of the study. The selection of strategies was based on the systematic survey conducted by Sahoo et al. [33], which ranges from fundamental context induction techniques, such as Zero-Shot and Few-Shot, widely used for adaptation to new tasks without retraining, to more advanced neuro-symbolic reasoning approaches aimed at code generation and execution (PoT and CoC). In this context, PoT and CoC seek to induce models to structure logical reasoning by simulating pseudocode and programmatic constructs (*think in code*), thereby mitigating limitations of prompts built from direct linguistic user instructions [33].

After identifying the strategies, the next step was to formulate the instructions by class. For each of the 16 classes, five prompts were developed, totaling 80 prompts in the study (5 strategies × 16 classes). The prompts were originally elaborated and executed in Brazilian Portuguese, and English translations are also available in the study's repository to support replication by researchers from other linguistic backgrounds.

4.1.4 Language Models (LLMs). To perform the test generation task, three LLMs were selected via the OpenRouter platform[1], which provides a unified API for accessing language models from different providers, enabling standardized requests and comparison across models with varying capabilities and cost profiles. The adoption of three models provides a more comprehensive evaluation of prompting strategies and reduces the risk of premature interpretations arising from observations under a single model.

Model selection considered experimental feasibility criteria: API access availability, recognized capability in code generation tasks, and cost profiles compatible with large-scale experiment execution. Free models were discarded due to limitations identified in preliminary tests, such as the generation of incomplete suites and daily usage restrictions. The selection also sought to include models with distinct characteristics, including two specialized in code generation and one general-purpose model that supports this task, used as a comparative reference. It should be emphasized that the models are not the primary focus of evaluation in this study; they are used as instruments for generating unit tests under the different prompting strategies investigated. Accordingly, Table 1 summarizes the adopted models.

Table 1: Language models used in the study

Model	Specialization
Llama 4 Maverick	General Purpose + Code Generation
Qwen3 Coder 480B A35B	Code Generation
Grok Code Fast 1	Code Generation + Reasoning

4.1.5 Mutation Testing Tool. For mutation testing, the PIT (Pitest) tool was adopted[2], selected for its maturity and widespread adoption in empirical software testing studies for Java systems [9, 21, 24, 40]. PIT integrates with the Java ecosystem and operates directly on bytecode, which contributes to its efficiency in mutant generation and execution, enabling its application in large-scale experiments [9, 24]. Furthermore, the tool supports an extensible set of mutation operators consolidated in the literature, with proven robustness in real-world development scenarios [9, 21]. These characteristics make PIT suitable for conducting the proposed experiment, which involves Java projects and execution at a considerable scale.

[1]https://openrouter.ai/
[2]https://pitest.org/

4.1.6 Script Design. After defining the experimental instruments, the preparation of the infrastructure and artifacts necessary for the study execution began. This stage comprised two main activities: (i) the design and organization of prompts for each adopted strategy; and (ii) the design of scripts responsible for automating the experimental pipeline and generating CSV reports.

The need to handle multiple experimental combinations, combined with the nondeterministic nature of LLM responses, necessitated mechanisms for execution control, logging, and orchestration. To this end, Bash and Python scripts were developed to integrate the different tools used in the study. A controlled environment was configured using a Docker container with Java 17, Maven support, PIT, and auxiliary tools. In building this infrastructure, not only the actions to be performed at each stage were defined, but also the control rules, validation criteria, and artifacts to be generated. Additionally, standards were established for result traceability, including the automatic generation of structured CSV reports to support subsequent analyses.

4.2 Execution

4.2.1 Test Suite Generation. Test suite generation was performed using a parameterized script responsible for orchestrating API calls based on the experimental combinations {`class`, `model`, `prompt strategy`}, as illustrated in Stage II, step 7 of the experimental flow (Figure 1). All calls were executed with temperature set to zero, aiming for greater consistency and reduced response variability. However, this adjustment does not completely eliminate the nondeterministic behavior of LLMs [7, 27]. To capture this residual variability, each combination was executed five times through independent API calls, in line with practices adopted in the literature [12, 27].

The models had access to the full source code for each target class and the corresponding prompts, organized in specific directories. Script execution produced collections of test cases implemented as Java test suites. The generation script automatically applied structural adjustments to the generated suites, such as package declarations, standardization of class names, and file organization, to ensure correct integration into the execution environment without altering the models' generated test logic. Note that any remaining compilation failures arise exclusively from the content generated by the LLMs. The files were named in a standardized manner, incorporating information about the model, prompt strategy, and execution number, allowing better traceability of the generated suites.

During execution, API-reported token usage was recorded in structured files to support traceability and cost estimation. Each target class produced 75 test suites (3 models × 5 strategies × 5 executions), totaling 1,200 suites in the study (16 classes × 75).

4.2.2 Test Suite Validation. All 1,200 generated suites underwent a validation process, as illustrated in Stage II, step 8 of the experimental flow (Figure 1). The first step was to automatically compile the test files and immediately discard suites that produced compilation errors. Next, the compilable suites were executed with Maven, and only those that completed successfully were classified as eligible. The quantitative validation results, i.e., eligible and ineligible suites per class, model, and prompting strategy, were automatically recorded in CSV files, enabling complete traceability and supporting analyses related to RQ1. Ineligible suites were discarded and did not proceed to the mutation testing stage. This filter ensures that the analysis reflects exclusively the ability of tests to detect faults, rather than structural or logical issues of the generated suites.

4.2.3 Mutation Testing.

Mutant Generation: For the generation of mutants in each class, a dedicated script was developed that runs PIT with the default set of mutation operators (DEFAULT)[3], producing a representative set of typical faults, as illustrated in Stage II, Step 9 of the experimental workflow (Figure 1). Mutant generation was restricted to the target method of each class by configuring the `<excludedMethods>` parameter via `pom.xml`, excluding all other methods from the mutation scope. This decision avoids distortions in the analyses, since the LLMs were instructed to generate tests only for the target method; mutants generated in other methods would inevitably remain uncovered, surviving not due to limitations of the test suites but because they lie outside the defined test scope. Furthermore, in the absence of test coverage, PIT does not select any tests to challenge the generated mutants, as it executes only the tests that cover the instruction at which each mutant is located [9]. Given this scenario, we defined a baseline test suite created manually, used exclusively to provide minimal coverage of the target method during mutant generation. Note that the baseline suite does not participate in the evaluations, only the suites generated by the LLMs are used in the mutation testing execution step.

Finally, the generated mutant set was fixed using the PIT history mechanism, ensuring that all LLM-generated suites were evaluated against the same set of mutants per class, thereby guaranteeing a fair comparison among the test suites. It should be emphasized that the number of mutants generated per class was not controlled in the experiment; it was determined solely by PIT, based on the structure of the target method and the mutation operators in the DEFAULT set. The resulting artifacts, including the mutation history (PIT history) and the corresponding reports, were stored in a directory dedicated to the experiment.

Execution of Test Suites on Mutants: With the set of mutants previously fixed via *pithistory*, mutation testing was performed, as illustrated in Stage II, Step 10 of the experimental workflow (Figure 1). To this end, a dedicated script identified, for each class, the eligible test suites compiled in the experiment directory, excluding the baseline suite used during the mutant generation step. For each identified suite, *PIT* was executed independently, reusing the previously established mutation history file (*pithistory*). The reports generated by *PIT* for each suite were stored in individual directories, enabling the extraction of metrics used in the evaluation step.

4.3 Evaluation

The evaluation stage, corresponding to Stage III of the experimental workflow (Figure 1), was structured around the formulated research questions, each addressed through specific metrics and procedures, as described below.

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

To answer RQ1, the eligibility of test suites was evaluated based on the combination of class, model, and prompt strategy, using structured reports automatically generated from the test suite validation artifacts, including quantitative records of eligible and ineligible suites. For each experimental combination, five independent executions were considered, and the Test Suite Eligibility Rate (TSER) was defined as:

$$\text{TSER} = \left(\frac{S_e}{S_t}\right) \times 100 \quad (1)$$

where S_e represents the number of eligible suites and S_t the total number of suites generated for that combination.

To answer RQ2, the eligible suites were evaluated individually using the Mutation Score (MS), which corresponds to the Mutation Coverage value reported by PIT from the test suite execution reports for mutants. MS is defined as:

$$\text{MS} = \left(\frac{M_k}{M_t}\right) \times 100 \quad (2)$$

where M_k represents the number of mutants killed by the test suite and M_t the total number of mutants generated for the respective class. Based on these reports, CSV artifacts were generated, containing, for each test suite, the counts of killed, survived, and uncovered mutants, which served as the basis for extracting the metric.

To answer RQ3, a qualitative analysis of surviving mutants was conducted based on the mutation testing reports generated for each eligible suite, following three complementary dimensions: (i) surviving mutants were grouped by mutation operator type, considering that multiple mutants may be associated with the same operator, allowing the identification of which operator types concentrated the largest number of surviving mutants, i.e., which categories of syntactic transformations presented the greatest difficulty for detection; (ii) observed patterns were aggregated across the set of analyzed classes, aiming to identify structural recurrences that support objective recommendations for test suite adjustments to improve defect detection capability; and (iii) mutants that survived in all mutation testing reports of their respective class test suites were highlighted, treated as candidates for equivalent mutants based on the observed systematic recurrence, without implying formal confirmation of semantic equivalence.

5 Results

5.1 RQ1: Test Suite Eligibility

To answer RQ1, we analyzed 1,200 generated test suites, of which 679 were eligible for mutation testing, yielding an overall eligibility rate of 56.58%. Among the 521 ineligible suites, 86 (7.17%) failed to compile and 435 (36.25%) failed at runtime. Table 2 summarizes the TSER for each prompting strategy, reporting mean and standard deviation ($\mu \pm \sigma$), median, and 95% confidence interval (CI) based on five independent executions per experimental combination.

The strategies showed similar mean eligibility rates, ranging from 47.23% (Few-Shot) to 60.85% (PoT), with high variability across all groups ($\sigma \approx 39\%$–40%). The medians reinforce this pattern: PoT reached 80%, suggesting more frequent high-eligibility executions, whereas Few-Shot recorded 40%, indicating lower consistency. The substantial overlap in the 95% confidence intervals suggests that the observed differences should be interpreted with caution, as they may reflect variability in the experimental combinations rather than systematic effects of the prompting strategies. Figure 3 shows the distribution of TSER by prompt strategy.

Table 2: Descriptive statistics of TSER per prompt strategy.

Strategy	TSER ($\mu \pm \sigma$)	Median (%)	95% CI
Zero-Shot	58.33 ± 39.32	70	[47.21 – 69.46]
One-Shot	60.00 ± 39.34	60	[47.47 – 70.03]
Few-Shot	47.23 ± 38.54	40	[35.29 – 57.21]
PoT	60.85 ± 40.64	80	[49.85 – 72.65]
CoC	59.57 ± 39.89	60	[46.91 – 69.76]

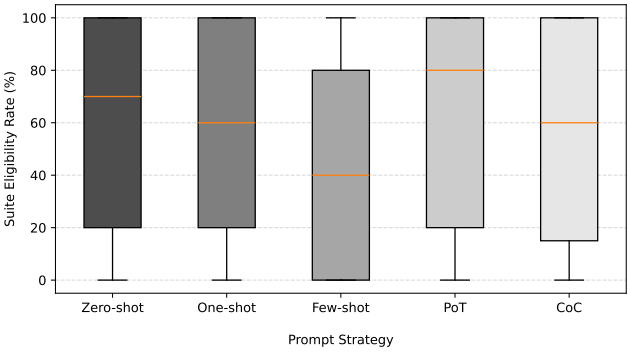

Figure 3: Distribution of TSER by prompt strategy.

As illustrated in Figure 3, regardless of the strategy adopted, eligibility rates vary widely between 0% and 100%, with substantial overlap among distributions and no explicit outliers. The PoT and Zero-Shot strategies exhibited the highest medians, whereas One-Shot and CoC recorded intermediate and closely related medians, suggesting similar behavior. In contrast, Few-Shot concentrates the lowest median values, indicating lower consistency in generating eligible test suites. The wide boxes across all strategies reflect high intra-strategy variability, reinforcing that test suite eligibility can vary considerably within the same prompting approach, with class–model combinations registering both null and maximum eligibility.

To formally assess the differences among the strategies, statistical tests were applied. The Shapiro–Wilk test rejected the normality hypothesis for all strategies ($p < 0.05$), with W statistics ranging from 0.80 to 0.86, motivating the use of the non-parametric Kruskal–Wallis test [38, 43]. The results showed no statistically significant difference among the prompt strategies ($H = 4.47$, $df = 4$, $p = 0.346$, $\alpha = 0.05$), with a negligible effect size [38] ($\eta^2_H = 0.002$), indicating that only 0.2% of the observed variance in TSER is associated with the prompt strategy. Thus, there is no evidence that more structured prompt strategies, such as PoT and CoC, yielded consistent gains in test suite eligibility.

Answer to RQ1. The prompt strategies did not exert a statistically significant effect on the eligibility rate of the generated test suites ($H = 4.47$, $p = 0.346$, $\eta^2_H = 0.002$). The high intra-strategy variability observed suggests that other factors, such as the language model used or the structural characteristics of the classes under test, may be related to instability in generating compilable and executable suites.

5.2 RQ2: Application of Mutation Testing

To answer RQ2, we considered the set of 679 test suites classified as eligible in the study, which together totaled 189 mutants, distributed across the 16 classes. Table 3 presents the descriptive statistics of the Mutation Score (MS) per prompt strategy, reporting the number of suites (n), mean, standard deviation (SD), median, first quartile (P25), third quartile (P75), minimum and maximum values, as well as the proportion of suites that achieved an MS equal to 100% (%MS=100).

Table 3: Descriptive statistics of MS per prompt strategy.

Strategy	n	Mean	SD	Median	P25	P75	Min	Max	%MS=100
Zero-Shot	140	86.28	23.08	100.00	75.00	100.00	15.80	100.00	65.70
One-Shot	142	84.88	24.13	100.00	65.20	100.00	15.80	100.00	66.20
Few-Shot	109	79.77	29.73	100.00	55.56	100.00	15.80	100.00	61.50
PoT	148	88.42	17.17	100.00	74.43	100.00	33.30	100.00	60.10
CoC	140	90.30	17.28	100.00	88.73	100.00	31.60	100.00	67.10
Total	679	86.24	22.54	100.00	72.73	100.00	15.80	100.00	64.20

The data in Table 3 show that all strategies presented a median equal to 100%, indicating that most suites were able to kill all mutants of the target method. Given the ceiling effect on the median and P75, the first quartile (P25) becomes the primary discriminating measure across groups. The CoC strategy achieved the best performance (P25 = 88.73%), indicating that at least 75% of its suites achieved an MS above this value, followed by Zero-Shot (75.00%) and PoT (74.43%). In contrast, Few-Shot recorded the lowest P25 (55.56%), a difference of 33.17 percentage points relative to CoC. This pattern is reinforced by the mean and standard deviation: CoC and PoT exhibited the highest means (90.30% and 88.42%) and the lowest standard deviations (17.28% and 17.17%), indicating better central performance and greater consistency, whereas Few-Shot recorded the lowest mean (79.77%) and the highest standard deviation (29.73%), with a larger proportion of suites achieving an MS below 50% (22.0%, compared to 2.1% for CoC).

As with RQ1, the Shapiro–Wilk test was applied to verify the normality assumption of the Mutation Score data. The results indicated p-values below 0.05 for all strategies (W between 0.62 and 0.69), rejecting the normality hypothesis and justifying the use of non-parametric methods. The Kruskal-Wallis test was then adopted to compare MS distributions across the five prompt strategies. The test revealed no statistically significant differences among the groups ($H = 4.223$; $df = 4$; $p = 0.377$; $\eta^2_H = 0.002$). This result indicates that, despite trends observed in the descriptive analysis, such as CoC's higher P25 (88.73%) than Few-Shot (55.56%), these differences are not sufficiently large to characterize a systematic effect of the prompt strategy on the mutant-detection capability of eligible suites.

Answer to RQ2. The prompt strategies did not present statistically significant differences regarding the Mutation Score values of the suites ($p = 0.377$). Although CoC and PoT exhibited the highest mean values (90.3% and 88.4%) and the lowest variability (SD of 17.28% and 17.17%) compared to the other strategies, the observed ceiling effect (median of 100% in all groups) and the lack of statistical significance indicate that the choice of strategy alone does not constitute a determining factor for the effectiveness of test suites from the mutation testing perspective.

5.3 RQ3: Surviving Mutants

Unlike the quantitative analyses performed in RQ1 and RQ2, addressing RQ3 required a qualitative approach based on inspecting the reports generated by *PIT*, aiming to infer structural adjustments in the test suites produced by the LLMs. Of the 189 mutants generated throughout the experiment, 43 remained alive in at least one of the eligible suites analyzed. The analysis was conducted by treating mutation operators as the primary unit of structural interpretation: operators represent categories of syntactic transformations applied to the original program, whereas mutants correspond to concrete instances of those transformations. Table 4 presents the total number of non-detection occurrences per operator, considering all eligible suites that exercised each surviving mutant.

Table 4: Impact of surviving mutation operators – total non-detection occurrences per operator.

Mutation Operator	Affected Suites	% of Total
ConditionalsBoundaryMutator	146	42.6
MathMutator	89	26.0
NegateConditionalsMutator	78	22.7
IncrementsMutator	15	4.4
VoidMethodCallMutator	9	2.6
BooleanTrueReturnValsMutator	6	1.7
Total	**343**	**100.0**

The data in Table 4 reveal a concentration of detection failures in three mutation operators. The *ConditionalsBoundaryMutator* alone accounted for 42.6% of the non-detection occurrences, followed by the *MathMutator* (26.0%) and the *NegateConditionalsMutator* (22.7%). Together, these three operators account for 91.3% of all non-detections, indicating that the generated suites face predominant difficulty in detecting mutations that alter boundary conditions, arithmetic operations, and conditional negation. In contrast, the *IncrementsMutator*, *VoidMethodCallMutator*, and *BooleanTrueReturnValsMutator* exhibited a reduced impact, accounting for less than 9% of the occurrences. This pattern suggests that the limitations of the suites are not uniformly distributed across the simulated fault types, but rather concentrate in specific mutation categories.

5.3.1 Recommendations for Test Suite Adjustments. Table 5 synthesizes the mutation operators identified as surviving in the *PIT* reports, the inferred faults, and the recommended adjustments, providing an interpretative guide for refining and improving the test suites, rather than a complete inventory of all observed occurrences.

To support the recommended adjustments in Table 5, the chain of evidence proposed by Wohlin et al. [43] was adopted as a systematic refinement procedure, consisting of: (i) identifying the surviving mutation operator and the affected classes; (ii) locating in the original code the mutation point associated with the operator; (iii) examining a non-detecting suite to identify the exercised scenarios and the missing condition; (iv) contrasting with a detecting suite to isolate the test case or assertion responsible for detection; and (v) incorporating the missing element into the fragile suite. It should be noted that, since *PIT* operates on bytecode, the exact syntactic transformation is not directly traceable in the source

Table 5: Mapping of operators, inferred faults, and recommended adjustments.

Operator	Inferred fault	Recommended adjustment
Conditionals-BoundaryMutator	Lack of test cases at the condition threshold value	Include test cases with values equal to the thresholds
MathMutator	Expected value mirrors the implementation, making the test insensitive to mutation	Adopt reference values computed independently from the code under test
NegateConditionals-Mutator	Lack of positive assertion for the negated branch of the condition	Explicitly cover both branches of boolean conditions
IncrementsMutator	No verification of the control variable progression	Verify the variable value across successive iterations
VoidMethodCall-Mutator	Lack of assertions about the result of the operation performed by the method	Include assertions about the object state after execution
BooleanTrueReturn-ValsMutator	Lack of a scenario with a `false` return	Include a test case that forces a negative return

code, being inferred based on the type of operator applied and the construct present at the indicated line.

For the two highest-impact operators, which together accounted for 68.6% of the non-detections, the aforementioned tracing procedure was applied. The *ConditionalsBoundaryMutator* (42.6%) was identified at a boundary comparison in a binary tree insertion method (`if (value < parent.data)`), in one of the study's classes. One of the suites generated for that class did not detect the mutant because it lacked a test case exercising the equality condition (`value == parent.data`), a scenario in which the inserted value exactly matches that of the parent node. A distinct suite, generated for the same class, detected the mutant by including a test case that inserted a value equal to the parent node's and verified the resulting insertion position, thereby eliminating the mutation. The *MathMutator* (26.0%), in turn, was identified in an area calculation method of another class in the study, whose expected value was obtained by delegating the calculation to the very implementation under test; when the mutant replaced multiplication with division, the *expected* value followed that change, and the test continued to pass. A distinct suite, generated for the same class, eliminated this fragility by adopting a reference value computed independently of the implementation.

5.3.2 Mutant Candidates for Equivalence. Two mutants survived in all eligible suites of their respective classes and were flagged as candidate equivalents according to the criterion defined in Section 4.3. Both are associated with the *ConditionalsBoundaryMutator*: one in the `Fraction` class (method `getReducedFraction`), surviving in all 22 suites of that class; another in `StringCompression` (method `compress`), surviving in all 25 suites. Although formal confirmation of equivalence is beyond the scope of this study, these cases represent a specific limitation to be considered when interpreting mutation score results.

Answer to RQ3. The analysis of surviving mutants enabled the inference of specific structural adjustments for six mutation operators, concentrated mainly in three categories: (i) inclusion of test cases at the boundary values of relational conditions (*ConditionalsBoundaryMutator*, 42.6% of non-detections); (ii) adoption of independent reference values in assertions (*MathMutator*, 26.0%); and (iii) explicit coverage of alternative branches in conditionals (*NegateConditionalsMutator*, 22.7%). The remaining operators (*IncrementsMutator*, *VoidMethodCallMutator*, and *BooleanTrueReturnValsMutator*) together accounted for less than 9% of the occurrences, with adjustments aimed at verifying loop progression, post-execution assertions, and coverage of negative returns.

6 Lessons Learned

The RQ1 results revealed that, even with full access to the target class's source code and the test scope restricted to a single method, 43.42% of the suites produced by the LLMs were ineligible, the vast majority of which failed at runtime (36.25%) rather than at compilation (7.17%). This pattern suggests that the models demonstrate reasonable capability to generate syntactically valid code but face difficulties in understanding the program's semantic contract, i.e., the expected behavior of the method under different input and state conditions. This observation raises a relevant question: source code alone appears to be insufficient as the sole input artifact for test generation. Specification documents, business rule descriptions, or annotations of expected behavior could provide the model with the semantic context that code does not explicitly express, potentially reducing the incidence of runtime failures and enhancing the model's ability to exercise less-obvious logical paths, such as boundary conditions.

Given this scenario, re-prompting strategies with feedback from compilation and execution error logs represent a promising path to maximize the set of suites assessable by mutation testing. In the conducted experiment, 521 suites were discarded before reaching the mutation testing evaluation stage, representing a significant volume of artifacts that could have been recovered through iterative correction cycles. In this direction, Konstantinou et al. [23] demonstrated that iterative feedback of error information to the model increased line coverage and mutant detection, showing that failure-driven re-prompting constitutes a promising strategy to increase the eligibility rate and, consequently, the investigative capacity of studies with experimental designs similar to that of this work.

7 Practical Implications

One of the most relevant contributions of this study lies in the observation that *prompt* strategies with structured reasoning, such as PoT and CoC, which incorporate elements of data flow analysis as a guiding mechanism for test generation, produced suites that showed more favorable performance trends in descriptive *Mutation Score* analyses. This finding suggests that testing teams may benefit from investing in the design of *prompts* grounded in classical software testing literature concepts, bringing the guidance provided to models closer to formal definitions such as *def-use*, *p-use*, and *c-use* associations, rather than purely descriptive instructions.

This direction directly connects to the issue of generated test quality. The surviving mutation operators identified in RQ3, especially the *ConditionalsBoundaryMutator* (42.6%), the *MathMutator* (26.0%), and the *NegateConditionalsMutator* (22.7%), can be used as

concrete inputs for refining generated suites, guiding specific structural adjustments. In this sense, the MUTGEN approach [42], which used surviving mutants as an iterative feedback mechanism for the model, represents an interesting complementary direction, in which the patterns identified in this study could be directly incorporated into the test suite generation process.

Finally, a balanced approach that combines test eligibility criteria, such as those adopted in this study, which restrict mutation testing application to compilable and executable suites, may offer a viable trade-off between operational feasibility and evaluative rigor in large-scale contexts.

8 Threats to Validity

Internal Validity: The primary threat to the internal validity of this study stems from the non-determinism of LLMs, which may introduce variability in results regardless of the *prompt* strategy used. To mitigate this effect, all executions were performed at zero temperature, and each {`class`, `model`, `strategy`} combination was run five times via independent API calls. Instrumentation threats were mitigated by using PIT and a fixed set of mutants per class, ensuring identical evaluation conditions. Finally, the generated artifacts were not modified throughout the experiment and were analyzed exactly as produced by the models, thereby preserving the integrity of the results.

External Validity: The relatively small scale of the study, comprising 16 classes and a single target method per class, limits the generalization of the results to larger and more heterogeneous codebases. The selected Java classes, although extracted from repositories well-established in the literature, do not represent the full diversity of industrial systems, particularly those with high coupling and complex architectures. Furthermore, the experiment was conducted in an academic context, using *single-pass* generation without reproducing typical elements of industrial environments, such as continuous integration and iterative refinement cycles. Finally, the LLMs evaluated correspond to specific versions, and were selected based on cost and feasibility criteria rather than state-of-the-art capability; more capable commercial models may exploit advanced prompting strategies such as PoT and CoC more effectively, potentially yielding gains not captured in this study, so that future versions of the models may yield different results.

Construct Validity: The quality of the tests was operationalized in two complementary dimensions: operational feasibility, measured by TSER, and defect detection effectiveness, measured by the *Mutation Score* [6, 11]. Although these metrics do not capture attributes such as readability or maintainability, the experiment was conducted at the method level, which favors greater scope control and comparability across treatments.

Conclusion Validity: The analytical power of the study was enhanced by a balanced experimental design, with an equal number of executions per {`class`, `model`, `strategy`} combination, totaling 1,200 generated test suites and 679 evaluated through mutation testing. Data normality was assessed using the Shapiro–Wilk test and, given the violation of the normality assumption, the nonparametric Kruskal–Wallis test was adopted. In addition to p-values, the analyses considered effect size (η^2_H) and 95% confidence intervals, thereby strengthening the robustness of the conclusions obtained [43].

9 Conclusion and Future Work

This paper presents an empirical study investigating how different *prompt* strategies influence the quality of unit tests generated by LLMs, considering both operational feasibility and effectiveness in defect detection via mutation testing. The results indicated that although no strategy stood out statistically, *prompts* based on structured reasoning, such as PoT and CoC, grounded in data-flow concepts, tend to produce more consistent test suites with better *Mutation Score* performance. This finding suggests that the way the instruction organizes the model's reasoning may affect the logical quality of the generated tests. Furthermore, the analysis of surviving mutants revealed recurrent weaknesses concentrated in three main aspects: absence of test cases to explore boundary conditions in conditionals (42.6%), assertions that delegate the computation of expected values to the code under test, rendering the tests insensitive to mutation (26.0%), and insufficient coverage of alternative branches (22.7%). Based on these results, objective recommendations were derived to refine the generated test suites.

The study used 16 Java classes with low method-level coupling in a single-pass generation scenario without iterative refinement, which limits the generalizability of the results to more complex industrial systems. As future work, the following stand out: (i) extending the PoT and CoC strategies to iterative generation scenarios with error log feedback; (ii) evaluating the semantic enrichment of *prompts* with specification artifacts, aiming to reduce runtime failures; and (iii) developing *prompts* targeted at mitigating the non-detection patterns identified in RQ3, converting the diagnosis produced in this study into more effective *prompting* strategies for the most recurrent mutation operators. In summary, the findings reinforce that the contribution of *prompt* engineering to LLM-based test automation is not limited to generating executable artifacts, but also includes producing logically more robust and relevant tests that detect program defects.

Artifact Availability

The replication package for this study is publicly available on Zenodo under a Creative Commons Attribution 4.0 International (CC BY 4.0) license: https://doi.org/10.5281/zenodo.21230015.

Acknowledgements

We want to thank CNPq (grant number 420406/2023-9) for partially funding this research. In this work, GenAI tools were used in two distinct contexts: first, as subjects of the empirical investigation (Section 4.1.4); and second, as writing support tools, with Claude (Anthropic), ChatGPT (OpenAI), and DeepSeek being used for spell and grammar checking, enhancement of textual cohesion and coherence, and formatting of tables in LaTeX. All content was reviewed and validated by the authors, who assume full responsibility for the accuracy and integrity of the work.

References

[1] 2024. Defects4J – version 3.0.1. https://github.com/rjust/defects4j

[2] Aldeida Aleti. 2023. Software Testing of Generative AI Systems: Challenges and Opportunities. In *Proceedings of the IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, Melbourne, Australia, 4–14. doi:10.1109/ICSE-FOSE59343.2023.00009

[3] Jothan Almeida. 2025. Prompt Engineering: A Comparative Study of Prompting Techniques in AI Language Models. In *Proceedings of the IEEE Integrated STEM Education Conference (ISEC)*. IEEE. doi:10.1109/ISEC64801.2025.11147384

[4] Khadija Almohsen, Fawzi Albalooshi, and Jafla Al Ammari. 2025. Exploring the Use of Generative AI in Automating Software Testing. In *Proceedings of the IEEE 4th International Conference on Computing and Machine Intelligence (ICMI)*. IEEE, 1–6. doi:10.1109/ICMI65310.2025.11139839

[5] Xavier Amatriain. 2024. Prompt Design and Engineering: Introduction and Advanced Methods. arXiv:2401.14423 [cs.SE] https://arxiv.org/abs/2401.14423

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

[7] Merve Astekin, Max Hort, and Leon Moonen. 2024. An Exploratory Study on How Non-Determinism in Large Language Models Affects Log Parsing. In *Proceedings of the ACM/IEEE 2nd International Workshop on Interpretability, Robustness, and Benchmarking in Neural Software Engineering*. ACM, Lisbon Portugal, 13–18. doi:10.1145/3643661.3643952

[8] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of Thoughts Prompting: Disentangling Computation from Reasoning for Numerical Reasoning Tasks. *Transactions on Machine Learning Research* (2023). doi:10.48550/arXiv.2211.12588 TMLR 2023.

[9] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, Saarbrücken, Germany, 449–452. doi:10.1145/2931037.2948707

[10] Tonmoy Debnath, Md Nurul Absar Siddiky, Muhammad Enayetur Rahman, Prosenjit Das, Antu Kumar Guha, Muhammad Rezaur Rahman, and H. M. Dipu Kabir. 2025. A Comprehensive Survey of Prompt Engineering Techniques in Large Language Models. (Oct. 2025). doi:10.36227/techrxiv.174140719.96375390/v2

[11] Márcio Eduardo Delamaro, José Carlos Maldonado, and Mario Jino. 2007. *Introdução ao Teste de Software*. Elsevier Editora, Rio de Janeiro, RJ, Brasil.

[12] Benedetta Donato, Leonardo Mariani, Daniela Micucci, and Oliviero Riganelli. 2025. Studying How Configurations Impact Code Generation in LLMs: The Case of ChatGPT. In *Proceedings of the 2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*. IEEE, 442–453. doi:10.1109/ICPC66645.2025.00055

[13] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarsky, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 1–23. doi:10.1109/ICSE-FoSE59343.2023.00008

[14] Sevdanur Genç, Mustafa Furkan Ceylan, and Ayhan İstanbullu. 2025. Software Unit Test Automation with LLM-Based Generative AI: Evaluating Test Quality through Code Coverage and Edge-Case Analysis. In *2025 10th International Conference on Computer Science and Engineering (UBMK)*. 242–247. doi:10.1109/UBMK67458.2025.11206953

[15] Robin Gröpler, Steffen Klepke, Jack Johns, Andreas Dreschinski, Klaus Schmid, Benedikt Dornauer, Eray Tüzün, Joost Noppen, Mohammad Reza Mousavi, Yongjian Tang, Johannes Viehmann, Selin Şirin Aslangül, Beum Seuk Lee, Adam Ziolkowski, and Eric Zie. 2025. The Future of Generative AI in Software Engineering: A Vision from Industry and Academia in the European GENIUS Project. In *Proceedings of the 2025 2nd IEEE/ACM International Conference on AI-powered Software (AIware)*. IEEE, 1–12. doi:10.1109/AIWARE69479.2025.00026

[16] Alexandru Guzu, Georgian Nicolae, Horia Cucu, and Corneliu Burileanu. 2025. Large Language Models for C Test Case Generation: A Comparative Analysis. *Electronics* 14, 11 (2025), 2284. doi:10.3390/electronics14112284

[17] Jon D. Hagar and Satoshi Masuda. 2024. Prompt Engineering Impacts to Software Test Architectures for Beginner to Experts. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 116–121. doi:10.1109/ICSTW60967.2024.00034

[18] Braxton Hall and Elisa Baniassad. 2022. Evaluating the Quality of Student-Written Software Tests with Curated Mutation Analysis. In *Proceedings of the 2022 ACM SIGPLAN International SPLASH-E Symposium (SPLASH-E '22)* (Auckland, New Zealand). ACM, New York, NY, USA, 24–34. doi:10.1145/3563767.3568132

[19] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, Hyderabad, India, 435–445. doi:10.1145/2568225.2568271

[20] iSEngLab. 2024. TestBench. https://github.com/iSEngLab/TestBench

[21] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. 2015. Syntactic Versus Semantic Similarity of Artificial and Real Faults in Mutation Testing Studies. *IEEE Transactions on Software Engineering* 41, 4 (2015), 367–382. doi:10.1109/TSE.2014.2367173

[22] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. 2002. Preliminary Guidelines for Empirical Research in Software Engineering. *IEEE Transactions on Software Engineering* 28, 8 (2002), 721–734. doi:10.1109/TSE.2002.1027796

[23] Michael Konstantinou, Renzo Degiovanni, Mark Harman, Mike Papadakis, and Jie M. Zhang. 2025. YATE: The Role of Test Repair in LLM-Based Unit Test Generation. arXiv:2507.18316 [cs.SE] doi:10.48550/arXiv.2507.18316

[24] Thomas Laurent and Anthony Ventresque. 2019. PIT-HOM: An Extension of Pitest for Higher Order Mutation Analysis. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2019)*. IEEE, 83–89. doi:10.1109/ICSTW.2019.00036

[25] Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. 2024. Chain of Code: Reasoning with a Language Model-Augmented Code Emulator. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*. PMLR. doi:10.48550/arXiv.2312.04474

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

[27] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. 2025. An Empirical Study of the Non-Determinism of ChatGPT in Code Generation. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–28. doi:10.1145/3697010

[28] Wendkûuni C. Ouédraogo, Kader Kaboré, Yewei Li, Jacques Klein, Haoye Tian, Anil Koyuncu, David Lo, and Tegawendé F. Bissyandé. 2026. Prompt Engineering in LLMs for Automated Unit Test Generation: A Large-Scale Study. *Empirical Software Engineering* 31, 103 (2026). doi:10.1007/s10664-026-10840-4

[29] Md Nakhla Rafi, An Ran Chen, Tse-Hsun (Peter) Chen, and Shaohua Wang. 2024. Exploring Data Cleanness in Defects4J and Its Influence on Fault Localization Efficiency. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. ACM, Lisbon Portugal, 386–387. doi:10.1145/3639478.3643125

[30] Md Nakhla Rafi, An Ran Chen, Tse-Hsun Peter Chen, and Shaohua Wang. 2025. Revisiting Defects4J for Fault Localization in Diverse Development Scenarios. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, Ottawa, ON, Canada, 63–75. doi:10.1109/MSR66628.2025.00022

[31] Rudolf Ramler, Philipp Straubinger, Reinhold Plosch, and Dietmar Winkler. 2025. Poster: Unit Testing Past vs. Present: Examining LLMs' Impact on Defect Detection and Efficiency. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 733–736. doi:10.1109/ICST62969.2025.10988960

[32] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM. *Proceedings of the ACM on Software Engineering (FSE)* 1, FSE (2024), 43:1–43:21. doi:10.1145/3643769

[33] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. 2024. A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications. arXiv:2402.07927 [cs.CL] https://arxiv.org/abs/2402.07927

[34] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–98. doi:10.1109/TSE.2023.3334955 Online published: 28 November 2023.

[35] Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, HyoJung Han, Sevien Schulhoff, Pranav Sandeep Dulepet, Saurav Vidyadhara, Dayeon Ki, Sweta Agrawal, Chau Pham, Gerson Kroiz, Feileen Li, Hudson Tao, Ashay Srivastava, Hevander Da Costa, Saloni Gupta, Megan L. Rogers, Inna Goncearenco, Giuseppe Sarli, Igor Galynker, Denis Peskoff, Marine Carpuat, Jules White, Shyamal Anadkat, Alexander Hoyle, and Philip Resnik. 2025. The Prompt Report: A Systematic Survey of Prompt Engineering Techniques. arXiv:2406.06608 [cs.CL] https://arxiv.org/abs/2406.06608

[36] Jay Ashok Shah and Garima Bajpai. 2025. Responsible Generative AI for Software Development Life Cycle. In *2025 IEEE World AI IoT Congress (AIIoT)*. IEEE, 0056–0061. doi:10.1109/AIIoT65859.2025.11105309

[37] Mukundraj Raman Sudarshan and Chandrashekhar Kumbhar. 2024. Comparative Analysis of Zero-shot, Few-shot, and One-shot Learning in Natural Language Processing. In *Proceedings of the International Conference on Augmented Reality, Intelligent Systems, and Industrial Automation (ARIIA)*. IEEE, Singapore. doi:10.1109/ARIIA63345.2024.11051380

[38] Maciej Tomczak and Ewa Tomczak. 2014. The need to report effect size estimates revisited. An overview of some recommended measures of effect size. *Trends in Sport Sciences* 21, 1 (2014), 19–25.

[39] Alexandra van der Spuy and Bernd Fischer. 2025. An Anatomy of 488 Faults from Defects4J Based on the Control- and Data-Flow Graph Representations of Programs. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering (EASE '25)*. ACM, Istanbul, Turkiye, 622–627.

doi:10.1145/3756681.3757025

[40] Oscar Luis Vera-Pérez, Martin Monperrus, and Benoit Baudry. 2018. Descartes: A PITest Engine to Detect Pseudo-Tested Methods. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, Montpellier, France, 878–881. doi:10.1145/3238147.3240474

[41] Jakub Walczak, Piotr Tomalak, and Artur Laskowski. 2025. Impact of Code Context and Prompting Strategies on Automated Unit Test Generation with Modern General-Purpose Large Language Models. arXiv:2507.14256 [cs.SE] doi:10.48550/arXiv.2507.14256

[42] Guancheng Wang, Qinghua Xu, Lionel Briand, and Kui Liu. 2026. Mutation-Guided Unit Test Generation with a Large Language Model. *IEEE Transactions on Software Engineering* (2026), 1–15. doi:10.1109/TSE.2026.3682975

[43] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer-Verlag Berlin Heidelberg. doi:10.1007/978-3-642-29044-2

[44] 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)*. ACM, New York, NY, USA, 1607–1619. doi:10.1145/3691620.3695529

[45] Peng Zhang, Yang Wang, Xutong Liu, Zeyu Lu, Yibiao Yang, Yanhui Li, Lin Chen, Ziyuan Wang, Chang ai Sun, Xiao Yu, and Yuming Zhou. 2024. Assessing Effectiveness of Test Suites: What Do We Know and What Should We Do? *ACM Transactions on Software Engineering and Methodology* 33, 4 (2024), 86:1–86:32. doi:10.1145/3635713

[46] Quanjun Zhang, Ye Shang, Chunrong Fang, Siqi Gu, Jianyi Zhou, and Zhenyu Chen. 2024. TestBench: Evaluating Class-Level Test Case Generation Capability of Large Language Models. doi:10.48550/arXiv.2409.17561 arXiv:2409.17561 [cs].

[47] Zhen Zheng, Kun Ning, Qiao Zhong, Junjie Chen, Wei Chen, Liang Guo, Wen Wang, and Yanhui Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *ACM Transactions on Software Engineering and Methodology* 33, 4 (2024), 1–45.