An Empirical Study of Assertion Generation Strategies for LLM-Based Test Oracles

Vanesa Mitseva

Supervisor(s): Annibale Panichella, Mitchell Olsthoorn

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Vanesa Mitseva
Final project course: CSE3000 Research Project
Thesis committee: Annibale Panichella, Mitchell Olsthoorn, Alexios Voulimeneas

Abstract

Unit test assertions are essential for detecting software faults, yet writing them remains costly and time-consuming. Large Language Models (LLMs) offer a promising way to automate assertion generation. However, prior work has primarily focused on generating assertions that closely mimic human-written ones. Because this represents only one possible generation strategy, the impact of alternative approaches on overall quality remains poorly understood. This paper presents an empirical study evaluating four distinct generation strategies: Assertion Generation, which was proposed and evaluated in prior work, alongside Assertion Augmentation, Blind Augmentation, and Chain-of-Thought Generation. Using GPT-oss 20b as the underlying model, we evaluate these strategies on 811 test oracles from 10 open-source projects in the GitBug-Java benchmark. We assess the generated assertions in terms of correctness, fault-detection capability, and textual similarity to developer-written assertions. Our results show that the choice of generation strategy strongly influences performance. Assertion Augmentation performs best overall, achieving the highest compilation rate, execution validity, and mutation score. Meanwhile, Chain-of-Thought Generation detects the highest proportion of real bugs, and standalone Assertion Generation yields results most similar to developer-written tests. Overall, the findings demonstrate that providing LLMs with existing developer-written assertions substantially improves the quality and effectiveness of generated test oracles.

Keywords

Assertion Generation Strategies, Software Testing, Mutation Testing, Large Language Models

1 Introduction

Software testing is essential to ensure software quality and reliability. In particular, unit tests help developers verify program behaviour at a fine-grained level and detect faults early in the development process. However, writing and maintaining tests is both time-consuming and expensive. Today, about 16% of engineering effort is devoted to testing activities [3]. One of the difficulties in this process is the *test oracle problem*, namely determining whether the observed behaviour of a program is correct [1]. In unit testing, these oracles are typically expressed through assertions.

To reduce the cost of manual testing, researchers have proposed automated test-generation techniques such as Search-Based Software Testing (SBST). Tools like EvoSuite are able to automatically generate unit tests together with assertions by observing runtime behaviour and recording it as expected output [4]. Although this enables large-scale automation, the generated assertions are often weak or overly tied to a specific execution and may fail to capture the intended semantics of the program. As a result, developers frequently need to inspect and refine such assertions manually [23].

Recent advances in Large Language Models (LLMs) have opened new possibilities for automated assertion generation. LLMs have shown promising results across several software-engineering tasks, including code generation, summarisation, and automated test generation [16]. Previous work by Molinelli et al. has shown that LLM-generated assertions can achieve mutation scores comparable to developer-written ones, although their effectiveness depends strongly on the prompt design and the context provided to the model [10]. Primbs et al., on the other hand, have demonstrated that models fine-tuned specifically for assertion generation outperform general-purpose LLMs on targeted benchmarks [14]. These findings suggest that the way assertions are generated may significantly affect their quality.

Despite recent progress in LLM-based assertion generation, most existing work has focused on comparing models or improving benchmark performance. Existing approaches typically follow an *Assertion Generation* setting, in which the model generates assertions from scratch for a given test method [10; 14]. In contrast, the influence of the generation strategy itself remains largely unexplored. This paper investigates four assertion generation strategies: *Assertion Generation*, *Assertion Augmentation*, *Blind Augmentation*, and *Chain-of-Thought Generation*. While Assertion Generation reconstructs missing assertions, Assertion Augmentation generates additional assertions in the presence of existing developer-written oracles, Blind Augmentation adds assertions without access to the original oracle, and Chain-of-Thought Generation [19] encourages explicit reasoning before producing assertions.

In this study, we investigate three key dimensions of assertion quality. First, we assess whether the generated assertions are syntactically and semantically correct by measuring their ability to compile and execute successfully. Second, we examine their fault-detection capability using both mutation testing and real historical bugs. Finally, we analyse how closely the generated assertions resemble developer-written assertions using exact-match and textual-similarity metrics.

The evaluation is conducted on GitBug-Java, a benchmark of real-world Java bugs from open-source repositories introduced in 2023 and later [18]. We assess the generated assertions using compilation rate, execution validity, mutation score, and bug-detection rate. Textual similarity to developer-written assertions is measured using BLEU, ChrF, and CodeBLEU.

Assertion Augmentation achieves the highest compilation rate (74.5%), execution validity (87.7%), and acceptance rate (78.4%) and produces the largest average mutation-score improvement over the original tests (+1.1%). Chain-of-Thought Generation detects the largest number of real bugs from the dataset. Assertion Generation produces the most textually similar assertions (exact-match accuracy: 36.4%), although this result may partly reflect benchmark contamination. Across all strategies, hallucinations account for approximately 30% of compilation failures.

The contributions of this paper are:

- An empirical comparison of four LLM-based assertion generation strategies on a real-world Java benchmark, evaluated across correctness, fault detection, and textual similarity.
- Evidence that Chain-of-Thought reasoning improves real-bug detection, identifying the most historical defects in the GitBug-Java dataset.

- Quantitative evidence that Assertion Augmentation produces the most syntactically and semantically correct assertions, with the largest mutation-score improvement over the original tests.
- Evidence that textual similarity to developer-written assertions is a poor indicator of fault-detection effectiveness, consistent with prior findings in the literature [17].

The remainder of this paper is structured as follows. Section 2 presents background and related work. Section 3 describes the methodology, the dataset, and the assertion generation strategies. Additionally, we present the experimental setup and evaluation metrics. Section 4 reports and discusses the experimental results. Section 5 outlines threats to validity, and Section 6 covers responsible research practices. Finally, Section 7 concludes the paper and outlines directions for future work.

2 Background & Related Work

Automated assertion generation lies at the intersection of software testing, test oracle generation, and Large Language Models (LLMs). To motivate our work, we first discuss Search-Based Software Testing (SBST) and the long-standing oracle problem (2.1). We then introduce mutation testing, which is widely used to evaluate the effectiveness of test assertions (2.2). Finally, we review recent LLM-based approaches for assertion generation and highlight the lack of research on how different assertion generation strategies influence assertion quality (2.3).

2.1 Search-Based Software Testing and the Oracle Problem

Search-Based Software Testing (SBST) formulates test generation as an optimisation problem and applies metaheuristic algorithms to automatically generate test inputs that maximise coverage or expose defects. Tools such as EvoSuite automatically synthesise unit tests and assertions for Java programs using search-based techniques [4].

However, despite their effectiveness in generating executable tests, SBST approaches do not solve the *test oracle problem*, i.e., determining whether the observed behaviour of a program under test is correct [1; 4]. In unit testing, test oracles are realised as assertions that validate runtime behaviour. Automated tools such as EvoSuite generate assertions by recording observed executions and treating them as correct behaviour [4]. While this enables fully automated test generation, the resulting assertions may be weak, overly specific, or incapable of detecting semantic faults, often requiring manual inspection and refinement [23].

Because assertion quality cannot be adequately measured through structural coverage alone, mutation testing is commonly used to evaluate the effectiveness of generated assertions [8].

2.2 Mutation Testing for Assertion Evaluation

Mutation testing evaluates the effectiveness of a test suite by systematically introducing small syntactic modifications, known as mutants, into the program under test and checking whether the assertions detect the resulting behavioural changes [6]. A mutant is considered killed if at least one assertion fails when executing the mutated program. The resulting mutation score is computed as the ratio of mutants killed to the total number of mutants generated.

Research has shown that mutation score correlates more strongly with real fault detection than traditional metrics such as line or branch coverage, which can often be satisfied by weak or trivial assertions [8; 22]. Consequently, mutation testing has become one of the standard approaches for evaluating assertion quality in automated testing research.

Several frameworks support mutation analysis for Java programs, among which PIT (PItest) is one of the most widely adopted due to its efficiency and integration with modern Java tooling [2]. In this work, we use PIT to evaluate the fault-detection effectiveness of generated assertions.

2.3 LLM-Based Assertion Generation

General LLM-Based Test Generation

Large Language Models (LLMs) have demonstrated strong performance across software engineering tasks, including code generation, summarisation, and automated test generation [16]. Recently, Molinelli et al. have shown that LLMs are capable of generating syntactically correct and executable unit tests, although their effectiveness depends heavily on prompt design and the amount of contextual information provided to the model [10].

While earlier work treated assertions as part of full test-case generation, more recent efforts have framed assertion generation as a standalone task for foundational models to address directly.

Assertion Generation Approaches

Recent studies demonstrate that LLM-generated assertions can achieve mutation scores comparable to human-written assertions under certain conditions [10]. In particular, providing contextual information such as the focal method, surrounding test code, and method invocations has been shown to substantially influence assertion quality.

At the same time, some models, specifically fine-tuned for assertion generation, have been proposed. AsserT5, for instance, outperforms general-purpose LLMs on dedicated benchmarks, suggesting that the quality of generated assertions is strongly influenced by how the generation task is formulated and optimized [14].

However, existing work primarily focuses on model architectures, benchmark performance, and prompt engineering, while the influence of the assertion generation strategy itself remains largely unexplored. This work addresses this gap through an empirical comparison of different assertion generation strategies and their impact on assertion effectiveness.

3 Methodology

This section describes the methodology used to evaluate the influence of different assertion generation strategies on the quality of LLM-generated test oracles. Figure 1 presents an overview of the experimental pipeline.

We structured this study around three research questions investigating assertion correctness, fault-detection capability, and similarity to developer-written assertions. To answer these

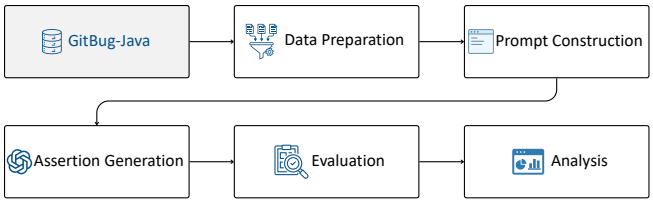

Figure 1: Overview of the experimental pipeline.

questions, we construct two evaluation datasets, apply four assertion generation strategies using the same LLM, and evaluate the generated assertions using a common set of metrics. Finally, we describe the experimental environment and inference configuration used to ensure reproducibility.

3.1 Research Questions

RQ1. *To what extent does the generation strategy affect the syntactical and runtime correctness of the generated assertions?*

This question evaluates whether different strategies produce assertions that successfully compile and execute within the original test suite.

RQ2. *How do different strategies compare in their ability to detect software faults?*

This question investigates whether certain strategies generate stronger assertions capable of distinguishing correct from faulty behaviour. The evaluation considers both mutation testing and detection of real historical bugs from GitBug-Java [18].

RQ3. *How closely do LLM-generated assertions resemble developer-written assertions?*

This question examines whether generated assertions are similar to human-written oracles at both textual and structural levels, and whether such similarity correlates with assertion effectiveness.

3.2 Datasets

GitBug-Java

This study uses GitBug-Java [18], a benchmark of real Java bugs extracted from recent bug-fixing commits in open-source GitHub repositories. Each entry contains both the buggy and fixed program versions, together with the corresponding test modifications introduced by developers during the bug-fixing process.

GitBug-Java was selected for two reasons. First, it enables the evaluation of fault-detection capability on real software defects. Since both buggy and corrected versions of each program are available, generated assertions can be executed on both revisions to determine whether they successfully distinguish faulty from intended behaviour [7; 18].

Second, GitBug-Java consists of relatively recent bug-fixing commits, reducing the likelihood of benchmark contamination compared to older and more widely used datasets. This is particularly important when evaluating LLM-generated assertions, as prior exposure to benchmark examples may artificially inflate performance [5; 24].

Bug Detection Benchmark

GitBug-Java is used exclusively to evaluate the real-bug detection capability of the generated assertions. To construct this benchmark, the raw dataset undergoes a filtering process similar to the one used by Silva et al. [18].

We only use PASS_PASS entries, where both the production code and the test suite were modified as part of the bug fix. These entries provide a developer-written oracle corresponding to the corrected behaviour and therefore allow generated assertions to be evaluated against real defects.

To ensure reproducibility, we additionally require entries to have a successful final continuous-integration execution. Finally, only entries whose test patches introduce new assertions or modify existing ones are retained. This guarantees that each selected bug is associated with an explicit oracle change that can be used for evaluation.

The resulting benchmark contains 96 bug-fixing commits from 38 repositories. For each entry, generated assertions are executed on both the buggy and fixed revisions. A bug is considered detected if the generated assertion passes on the fixed version and fails on the buggy version.

Evaluation Dataset Construction

While GitBug-Java is suitable for evaluating real-bug detection, it contains a limited number of test modifications and therefore provides only a small number of evaluation instances for assessing assertion correctness and mutation-testing effectiveness. To obtain a larger and more representative set of test methods, we additionally mined tests from repositories contained in GitBug-Java.

The mining procedure is inspired by the methodology of Molinelli et al. [10]. First, we selected a subset of repositories from GitBug-Java and cloned them locally. Test files were then identified automatically, and JUnit test methods were extracted. Only methods containing at least one developer-written assertion were retained. Finally, we discarded methods that could not be compiled or executed within their original project context.

The final dataset consists of 404 test methods, containing 811 assertions, gathered from 10 repositories. For each test method, the original assertions were used to construct the inputs required by the four generation strategies described in Section 3.3. The generated assertions were then evaluated using the correctness, mutation-testing, and similarity metrics introduced in Section 3.4.

3.3 Assertion Generation Strategies

We performed all assertion generation experiments using the same Large-Language Model, *GPT-oss 20b cloud*, to isolate the effect of the generation strategy from differences in model capabilities. To ensure a fair comparison, every strategy was provided with exactly the same context: the focal method, the test method, the source class under test, and the additional contextual information required for assertion generation. Keeping this context identical across all strategies ensures that the generation strategy itself remains the only controlled experimental variable, so that any observed differences in the results can be attributed to the strategy rather than to variations in the information available to the model.

Assertion Generation
This strategy, inspired by the experimental setup of Molinelli et al. [10], represents the baseline generation setting. The developer-written assertions in the test method are removed and replaced with placeholders before the test is provided to the model. Rather than regenerating all assertions at once, each removed assertion is generated individually and in sequence, one position at a time. This setup evaluates the model's ability to construct assertions directly from the available test context without relying on developer-written oracles. The generated assertions are inserted back into the original test method and evaluated using the metrics defined in Section 3.4.

Assertion Augmentation
In the augmentation setting, the complete original test method, including the developer-written assertions, is provided to the model. Rather than replacing existing assertions, the model generates only additional ones intended to complement the current oracle. This strategy evaluates whether exposing the model to human-written assertions improves assertion quality and fault detection while preserving the original developer intent. We motivate this strategy with recent work, conducted by Khandaker et al., showing that additional contextual information can improve LLM-based oracle generation [9]. In this setting, the existing developer-written assertions themselves serve as contextual guidance for the model.

Blind Augmentation
Blind Augmentation combines elements of generation and augmentation. Original assertions are removed from the test method and replaced with placeholders, but instead of reconstructing them, the model is instructed to generate new assertions intended to strengthen the test. The objective of this strategy is to prevent direct imitation of developer assertions while preserving the surrounding test structure. This allows evaluation of whether limiting oracle information encourages more diverse and fault-sensitive assertions.

Chain-of-Thought Generation
The final strategy introduces explicit intermediate reasoning into the assertion generation process. Chain-of-Thought prompting has been shown to improve performance on reasoning-intensive tasks by eliciting step-by-step intermediate representations [19]. In this setting, the model first reasons about the expected behaviour of the focal method and the purpose of the test before generating the replacement assertions. This strategy evaluates whether explicit reasoning improves assertion correctness and fault-detection capability.

3.4 Evaluation Metrics

Our evaluation protocol builds upon the evaluation methodologies proposed by Molinelli et al. [10] and Primbs et al. [14]. In particular, we adopt mutation-testing and textual-similarity metrics from prior work, while additionally evaluating bug-detection capability on GitBug-Java [18] and inference time to assess the practical cost of each strategy.

We organised the evaluation metrics by research question, covering the syntactic and runtime correctness of the generated assertions (RQ1), their fault-detection capability (RQ2), and their resemblance to developer-written assertions (RQ3).

RQ1: Syntactic and Runtime Correctness

- **Compilation Rate:** The proportion of test methods that compile successfully after adding the generated assertions. Because an assertion that does not compile can neither be executed nor detect faults, this metric captures the most basic prerequisite for usability and quantifies the syntactic validity of a strategy's output.
- **Execution Validity:** The proportion of all test methods that execute without runtime error after inserting the generated assertions. This metric is measured unconditionally - it includes both compiled and non-compiled test methods in the denominator, giving a global view of execution success. A usable oracle must not raise false alarms on correct code, so this metric measures whether the assertions encode behaviour that is genuinely true of the program under test.
- **Acceptance Rate:** Among test methods that compile successfully, the proportion that also executes without a runtime error. By conditioning on compilation, this metric isolates runtime correctness from syntactic failures, showing how often a syntactically valid assertion is also behaviourally consistent with the program under test.
- **Error Types Distribution:** The breakdown of rejected assertions by category (e.g., hallucinated APIs, type mismatches). This distribution exposes the dominant failure modes.
- **Average Inference Time:** The average time required per prompt. This complementary metric is used to evaluate the practical efficiency of each strategy and identify potential trade-offs between assertion quality and generation cost.

RQ2: Fault-Detection Capability

- **Mutation Score:** The proportion of mutants that the test detects (kills) out of all generated mutants. Mutation analysis is a well-established proxy for fault-detection ability, so the score directly measures whether the generated assertions strengthen a test's capacity to distinguish correct from faulty behaviour.
- **Bug Detection Rate (GitBug-Java):** The percentage of real bugs from GitBug-Java for which a generated assertion passes on the fixed version and fails on the buggy version. Unlike the mutation score, which relies on synthetic faults, this metric measures the detection of real, historical defects and therefore provides the most direct evidence of practical fault-finding capability.

RQ3: Resemblance to Developer-Written Assertions

- **Exact-Match Accuracy:** The proportion of generated oracles that are textually identical to a developer-written assertion for the same test. It is a strict, unambiguous measure of how closely a strategy reproduces human oracles, although it necessarily undercounts assertions that are semantically equivalent but textually different.
- **BLEU, ChrF, and CodeBLEU:** Following the use of NLG-based textual-similarity metrics for neural test-oracle generation [17], these three metrics quantify how

Table 1: Syntactic and runtime correctness metrics by strategy.

Strategy	Comp. Rate	Acc. Rate	Exec. Validity
Generation	58.3%	71.5%	70.2%
Augmentation	**74.5%**	**78.4%**	**87.7%**
Blind	65.6%	72.8%	77.2%
Thinking	59.7%	71.0%	75.4%

closely the generated assertions resemble the developer-written ones. BLEU [12] measures n-gram precision overlap and awards partial credit where exact-match is too strict. ChrF [13] computes an F-score over character n-grams, which is more robust to small lexical differences in identifiers and tokenisation than word-based BLEU. Lastly, CodeBLEU [15] extends BLEU with weighted keyword matching and similarity over the abstract syntax tree and data flow, capturing syntactic and semantic structure that token-level metrics ignore.

3.5 Experimental Setup

To ensure reproducibility, we fixed the random seed to 42 for both dataset sampling and LLM inference. All experiments were conducted on a personal laptop equipped with an AMD Ryzen 7 7735HS processor (8 cores at 3.2 GHz), 16 GB RAM, and Windows 11.

Inference was performed using *GPT-oss 20b cloud*, a 20.9-billion-parameter model served through Ollama and quantised to MXFP4 precision. We fixed the context window to 8,192 tokens. Mutation analysis was performed using PIT version 1.15.3 with the DEFAULTS mutator set.

We executed all pipeline scripts using Python 3.12.1, while Java 21.0.2 was used for compilation and test execution. Key libraries included pandas 2.2.3, numpy 2.2.5, ollama 0.4.8, and litellm 1.70.0. A complete `requirements.txt` file is included in the replication package.

4 Empirical Results

4.1 RQ1: Syntactic and Runtime Correctness

Results

Table 1 outlines the syntactic and runtime correctness across the four generation strategies. *Assertion Augmentation* achieved the highest performance across all metrics, leading in compilation rate (74.5%), acceptance rate (78.4%), and execution validity (87.7%). In contrast, the baseline *Assertion Generation* strategy performed the worst overall, failing to compile in 41.7% of cases and yielding the lowest execution validity (70.2%).

The distribution of generated assertion types was broadly consistent across all strategies. `assertEquals` and `assertThat` together accounted for between 71.2% and 76.5% of generated assertions, closely matching the developer baseline. However, *Assertion Augmentation* produced the highest proportion of `assertThat` assertions (32.4%), compared to 25.9% for *Assertion Generation*, 28.8% for *Blind Augmentation*, and 26.8% for *Chain-of-Thought Generation*.

Regarding computational overhead, *Assertion Augmentation* exhibited the lowest latency among method-level approaches, requiring 5.50 seconds per prompt on average, compared to *Blind Augmentation* (5.74s) and *Chain-of-Thought Generation* (6.43s). Although *Assertion Generation* recorded a lower nominal latency of 3.88 seconds per prompt, this metric is not directly comparable, as the strategy operates on individual assertions rather than complete test methods.

Discussion

The results demonstrate that the quality of the generated assertions is heavily influenced by the amount and type of context provided to the model. *Assertion Augmentation* achieved the highest compilation rate, acceptance rate, and execution validity, suggesting that providing the LLM with existing developer-written assertions reduces the likelihood of generating invalid or incorrect assertions.

The assertion-type distribution provides further evidence for this explanation. *Assertion Augmentation* generated the highest proportion of `assertThat` assertions (32.4%), exceeding both the developer baseline (24.9%) and all other strategies. Since `assertThat` assertions typically require more complex matcher expressions and are therefore more susceptible to syntax errors and hallucinated APIs, their successful use suggests that exposure to developer-written assertions helps the model infer the correct assertion structure. This may partially explain why *Assertion Augmentation* achieved the highest compilation rate despite producing a comparatively large number of complex assertions.

In contrast, the baseline *Assertion Generation* strategy performed poorly across all correctness metrics. By generating assertions in isolation, the model more frequently produces assertions that are syntactically invalid and fail during compilation.

Another notable finding is that approximately 30% of compilation failures were caused by hallucinations. References to non-existent methods, variables, and APIs therefore remain a major source of failure across all strategies, even when additional contextual information is provided [20].

Conclusion RQ1. The choice of generation strategy substantially influences assertion correctness. *Assertion Augmentation* produced the most reliable assertions, achieving the highest compilation rate, acceptance rate, and execution validity. Providing developer-written assertions as context appears particularly beneficial when generating more complex assertion forms, while hallucinated APIs remain a persistent source of failure across all strategies.

4.2 RQ2: Fault-Detection Capability

Results

Table 2 summarizes the fault-detection performance of the four assertion generation strategies. *Assertion Augmentation* achieved the highest mutation score among the evaluated generation strategies (67.3%) and produced the largest average mutation-score improvement over the original tests (+1.1%). On the other hand, *Chain-of-Thought Generation* detected the largest number of real bugs (57.1%) out of the ones selected from the GitBug-Java dataset.

Table 2: Comparison of assertion generation strategies, based on fault-detection capability.

Strategy	Mut. Score	Delta	Bug Detection
Original	66.2%	–	–
Generation	58.2%	-8.0%	38.5%
Augmentation	**67.3%**	**+1.1%**	46.7%
Blind	57.6%	-8.6%	30.8%
Thinking	59.1%	-7.1%	**57.1%**

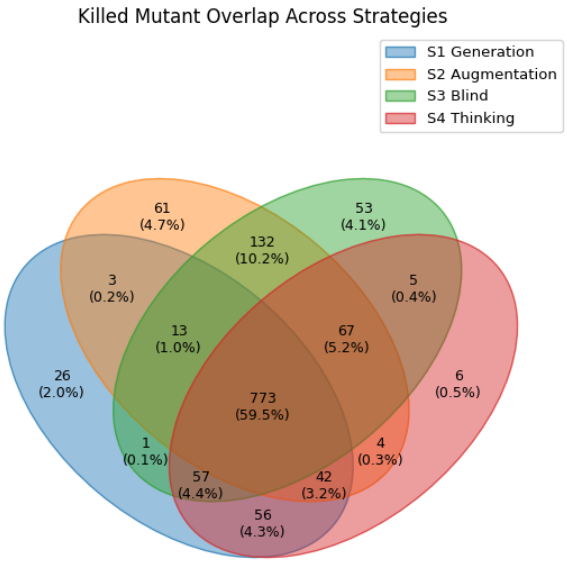

Figure 2: Killed-mutant Venn diagram across the four generation strategies.

Figure 2 shows the killed-mutant overlap across strategies. Of 1,299 unique mutants killed in total, 773 (59.5%) are killed by all four strategies, forming a shared core that no individual strategy exclusively contributes to. The largest intersection beyond this core belongs to *Assertion Augmentation* and *Blind Augmentation* together, which jointly account for 132 (10.2%) additional mutants. Among strategy-exclusive kills, *Assertion Augmentation* contributes the most (61), followed by *Blind Augmentation* (53).

Discussion

According to the average mutation-score improvement (Delta Δ), *Assertion Augmentation* achieved the strongest improvement (+1.1%), substantially outperforming the other strategies, which, on average, decrease the baseline mutation score. The overlap analysis in Figure 2 supports this finding, as *Assertion Augmentation* contributes the most strategy-exclusive kills, confirming that it targets mutants, the other strategies miss.

A similar trend is observed for real-bug detection. *Chain-of-Thought Generation* detected the highest proportion of historical defects (57.1%), followed by *Assertion Augmentation* (46.7%). These results suggest that strategies providing either richer contextual information (*Augmentation*) or explicit reasoning guidance (*Chain-of-Thought*) produce assertions that are more effective at strengthening fault detection [21].

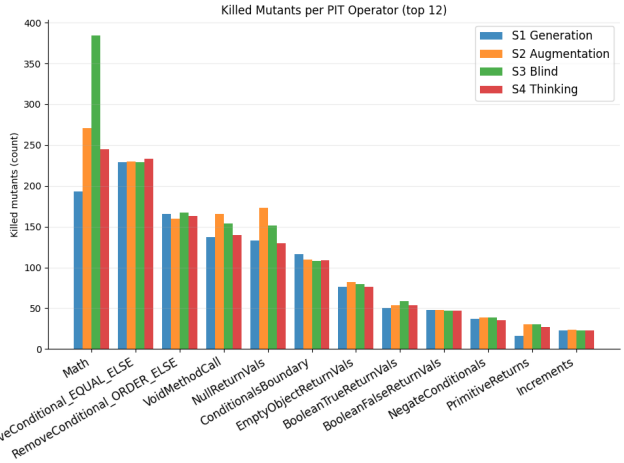

Figure 3: Number of mutants killed per PIT operator for each strategy.

The opposing rankings of *Chain-of-Thought Generation* on mutation score (-7.1% delta) and real-bug detection (57.1%) call for closer examination. The overlap data in Figure 2 reveals that *Chain-of-Thought Generation* contributes only 6 strategy-exclusive mutant kills, far fewer than any other strategy. Yet it detects the most real bugs. This tension reflects a known limitation of mutation testing as a proxy for real fault detection: the correspondence between synthetic and real faults is imperfect and depends on the nature of the faults, the mutation operators applied, and the size of the dataset [7; 11].

The assertion-type distribution offers a complementary explanation. Unlike the other strategies, *Chain-of-Thought* generated substantially more `assertThrows` assertions (7.6% versus 3.1–3.8%). Exception assertions verify that the code raises the correct error under specific conditions, making them sensitive to disruptions in the execution path that PIT's standard operators do not typically exercise. As shown in Figure 3, the operator composition of kills is broadly similar across all four strategies, confirming that performance differences arise not from targeting different mutation types, but from the specificity and reliability with which each strategy targets them. One notable exception is *Blind Augmentation*, which kills substantially more Math mutants in absolute terms than the other strategies. This advantage does not translate into a positive per-test delta. A plausible explanation is that *Blind Augmentation*, lacking access to developer-written assertions, tends to generate generic value-checking oracles that are naturally sensitive to arithmetic substitutions but insufficiently specific to strengthen detection across other mutation types, resulting in a net negative delta overall.

Conclusion RQ2. *Assertion Augmentation* produced the largest improvement in mutation score over the original tests, while *Chain-of-Thought Generation* detected the largest number of real bugs. These findings indicate that generation strategies that either utilize existing developer-written assertions or encourage explicit reasoning yield the greatest gains in fault-detection capability.

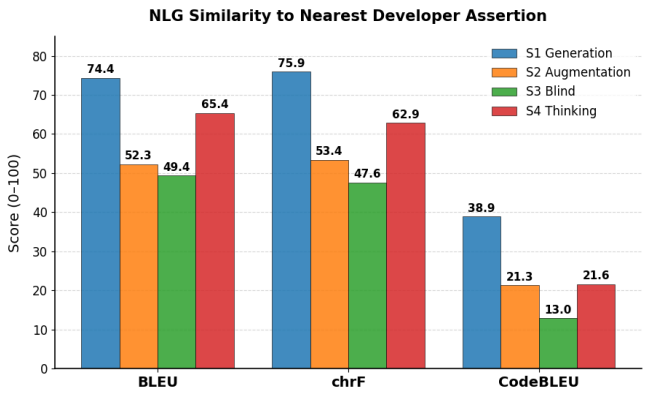

Figure 4: NLG similarity to the nearest developer-written assertion for each generation strategy, measured by BLEU, ChrF, and CodeBLEU.

4.3 RQ3: Resemblance to Developer-Written Assertions

Results

Comparison of generated assertions to developer-written ones shows that *Assertion Generation* achieves the highest exact-match accuracy (36.4%), followed by *Chain-of-Thought Generation* (26.2%), *Blind Augmentation* (4.8%), and *Assertion Augmentation* (2.7%). Figure 4 presents the BLEU, ChrF, and CodeBLEU scores computed against the most similar developer-written assertion for each generated assertion. Across all three metrics, *Assertion Generation* achieved the highest similarity scores (BLEU: 74.4, ChrF: 75.9, CodeBLEU: 38.9), followed by *Chain-of-Thought Generation*, while the augmentation-based strategies consistently produced the lowest similarity values.

Discussion

The results are largely explained by the objectives of the individual strategies. *Assertion Generation* and *Chain-of-Thought Generation* are explicitly tasked with replacing removed assertions and, therefore, aim to reproduce the behaviour encoded by the original developer-written oracle. Consequently, they are expected to achieve the highest similarity scores.

In contrast, *Assertion Augmentation* and *Blind Augmentation* are instructed to generate additional assertions rather than recreate existing ones. Their purpose is to complement the developer oracle with new checks, making low textual similarity an expected outcome rather than a weakness. A low similarity score in these settings, therefore, indicates that the model is generating different assertions, not necessarily lower-quality ones.

The results from RQ2 support this interpretation. Although *Assertion Generation* achieved the highest similarity scores, it did not achieve the strongest fault-detection performance. In contrast, *Assertion Augmentation* had a positive mutation-score improvement despite producing assertions that differed substantially from the developer-written ones. This suggests that similarity to developer assertions and fault-detection effectiveness capture different aspects of assertion quality.

Conclusion RQ3. *Assertion Generation* produces assertions that most closely resemble developer-written ones, which is expected given its objective of reconstructing missing assertions. The augmentation strategies generate substantially different assertions by design, as they are intended to complement rather than reproduce the developer oracle. Similarity metrics, therefore, measure how closely a strategy mimics developer-written assertions, but do not necessarily reflect fault-detection effectiveness.

5 Threats to Validity

In this section, we outline the main threats that may affect the validity of our findings, including threats related to internal (5.1), construct (5.2), external (5.3), and conclusion (5.4) validity.

5.1 Internal Validity

Data Leakage

Although we use the recently introduced GitBug-Java dataset [18] to reduce leakage risk, it is still possible that *GPT-oss 20b* was exposed to similar code patterns during training. This could slightly influence the generated assertions.

As a rough indicator, the Assertion Generation strategy achieves 36.4% accuracy, suggesting that large-scale memorization is unlikely, although limited overlap with similar repositories cannot be fully excluded.

Notably, Assertion Augmentation and Blind Augmentation are the strategies least susceptible to data leakage: since they generate additional assertions beyond those already written by developers, the generated oracles are not present in any training corpus by definition.

PIT Failure Handling

Some PIT runs fail or require fallback execution, resulting in missing or slightly inconsistent mutation scores. To mitigate this, the pipeline applies automatic retries and expands the mutation scope when no mutants are found.

5.2 Construct Validity

Semantic Equivalence

Exact-match accuracy does not capture cases where different assertions express the same underlying behaviour, such as checking emptiness via a boolean condition versus comparing a size to zero. To account for this, we also use BLEU, ChrF, and CodeBLEU as approximate similarity measures. However, these metrics are still limited in capturing whether two assertions are truly equivalent from a semantic perspective.

Ground Truth

We treat human-written assertions as ground truth when evaluating generated assertions, assuming they correctly capture the intended behaviour of the code. However, this assumption may not always hold, as developer-written tests can be incomplete or overly specific. For this reason, we additionally evaluate mutation scores as a more behaviour-oriented complement to direct comparison with the reference assertions.

5.3 External Validity

Dataset Scope

Our experiments are conducted on the GitBug-Java dataset [18] using Java unit tests and PIT mutation testing [2]. Consequently, the results may not generalize to other programming languages, testing frameworks, or industrial codebases with different testing practices.

Model Scope

This study evaluates a single model, *GPT-oss 20b cloud*. Other LLMs may exhibit different assertion generation behaviour, prompting sensitivity, or mutation testing performance, limiting the generalizability of the observed results across model families.

5.4 Conclusion Validity

Non-Determinism

LLM inference is inherently non-deterministic. To reduce the impact of generation variability, all experiments were executed with a fixed random seed (42) and repeated six times. We report the average results across runs. Although this substantially improves the stability of the measurements, some variability may still remain due to factors beyond seed control.

Sample Size

Although the evaluation includes multiple projects and test oracles, the sample size remains relatively small, which may limit the study's statistical power. As a result, subtle differences between generation strategies may not be reliably detected.

6 Responsible Research

Data licensing. All datasets and third-party tools used in this study are openly licensed. GitBug-Java is published under the MIT License. The inference model, *GPT-oss 20b*, was accessed under its standard usage terms. The mutation-testing framework PIT is distributed under the Apache License 2.0.

Reproducibility. To support replication, we make the complete experimental pipeline available, including all dataset preprocessing scripts, prompt templates, inference scripts, mutation-testing configurations, and analysis notebooks. The random seed was fixed to 42 throughout all experiments. Software versions and hardware specifications are reported in Section 3.5. The replication package is available at https://github.com/VanesaM18/assertion-generation-strategies.

Societal impact and limitations. This work investigates how LLMs can assist developers in generating unit-test assertions. Such tools have the potential to reduce the effort required to write and maintain tests, allowing software engineers and testers to focus on higher-level validation tasks, such as designing test scenarios, identifying requirements, and reviewing generated outputs. However, the results of this study also demonstrate that automatically generated assertions remain far from fully reliable. Compilation failures, execution failures, and hallucinated references occur across all evaluated strategies, indicating that human oversight is still necessary. Consequently, the techniques studied in this paper should be viewed as developer-assistance tools rather than replacements for software testing professionals.

AI tool usage. AI language tools were used to assist with drafting and editing parts of this paper. All AI-generated content was reviewed, verified, and substantially revised by the author. No text was included without manual review. All factual claims, numerical results, and citations were independently verified by the author.

7 Conclusion & Future Work

This paper investigated how different assertion generation strategies influence the quality of LLM-generated test assertions. We compared four strategies, namely *Assertion Generation, Assertion Augmentation, Blind Augmentation, and Chain-of-Thought Generation,* using 404 Java test methods from 10 repositories in the GitBug-Java benchmark.

The results show that the choice of generation strategy has a substantial impact on assertion quality. *Assertion Augmentation* produced the most reliable assertions, achieving the highest compilation rate (74.5%), acceptance rate (78.4%), execution validity (87.7%), and mutation score (67.3%). Its successful generation of more complex assertion forms, such as `assertThat` expressions, suggests that exposure to developer-written assertions helps the model infer correct assertion structure. It also delivered the largest average improvement in mutation score (+1.1%) and the most strategy-exclusive mutant kills (61), indicating the greatest overall benefit when strengthening existing test suites.

Chain-of-Thought Generation detected the highest proportion of historical bugs (57.1%), despite achieving a negative mutation-score improvement (-7.1%) and contributing the fewest strategy-exclusive mutant kills (6). Its strong bug-detection performance is partly explained by its greater use of `assertThrows` assertions, which are sensitive to control-flow disruptions that PIT's standard mutation operators do not typically exercise. Together, these findings suggest that mutation score and real-bug detection capture complementary aspects of assertion quality, and that neither metric alone is sufficient to evaluate a generation strategy.

Assertion Generation produced assertions that were most similar to developer-written ones according to all textual similarity metrics. However, this did not translate into stronger fault detection, indicating that textual similarity alone is not a reliable measure of assertion quality. Across all strategies, approximately 30% of compilation failures were caused by hallucinations, highlighting a persistent limitation of LLM-based assertion generation.

Overall, *Assertion Augmentation* provides the strongest balance between correctness and fault-detection effectiveness, while *Chain-of-Thought Generation* is the most effective strategy when the primary objective is detecting real software defects.

Future work should investigate whether these findings generalise across different LLMs and model sizes. Another promising direction is the development of hybrid strategies that combine the contextual information available in *Assertion Augmentation* with the explicit reasoning encouraged by *Chain-of-Thought Generation*. Additionally, future work should examine the gap between mutation-score improvement and real-bug detection for LLM-generated assertions, as the results

suggest that standard mutation operators may not adequately capture the fault classes that LLM-based strategies are most effective at detecting. Finally, future evaluations could incorporate semantic equivalence techniques to better assess whether the generated assertions capture the same intent as developer-written oracles, regardless of syntactic differences.

All artefacts associated with this study are publicly available at https://github.com/VanesaM18/assertion-generation-strategies.

Acknowledgments

We are thankful to Professor Annibale Panichella and Professor Mitchell Olsthoorn of the Software Engineering Research Group at TU Delft for their guidance and feedback throughout this project.

References

[1] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. doi:10.1109/TSE.2014.2372785

[2] 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* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 449–452. doi:10.1145/2931037.2948707

[3] Ermira Daka and Gordon Fraser. 2014. A Survey on Unit Testing Practices and Problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 201–211. doi:10.1109/ISSRE.2014.11 ISSN: 2332-6549.

[4] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. doi:10.1145/2025113.2025179

[5] Jacob Haimes, Cenny Wenner, Kunvar Thaman, Vassil Tashev, Clement Neo, Esben Kran, and Jason Schreiber. 2024. Benchmark Inflation: Revealing LLM Performance Gaps Using Retro-Holdouts. arXiv:2410.09247 [cs.LG] https://arxiv.org/abs/2410.09247

[6] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Softw. Eng.* 37, 5 (Sept. 2011), 649–678. doi:10.1109/TSE.2010.62

[7] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. doi:10.1145/2610384.2628055

[8] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 654–665. doi:10.1145/2635868.2635929

[9] Shaker Mahmud Khandaker, Fitsum Kifetew, Davide Prandi, and Angelo Susi. 2025. AugmenTest: Enhancing Tests with LLM-Driven Oracles. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 279–289. doi:10.1109/ICST62969.2025.10988926

[10] Davide Molinelli, Luca Di Grazia, Alberto Martin-Lopez, Michael D. Ernst, and Mauro Pezzè. 2025. Do LLMs Generate Useful Test Oracles? An Empirical Study with an Unbiased Dataset. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 278–290. doi:10.1109/ASE63991.2025.00031 ISSN: 2643-1572.

[11] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2018. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 537–548. doi:10.1145/3180155.3180183

[12] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (Philadelphia, Pennsylvania) *(ACL '02)*. Association for Computational Linguistics, USA, 311–318. doi:10.3115/1073083.1073135

[13] Maja Popović. 2015. chrF: character n-gram F-score for automatic MT evaluation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, Ondřej Bojar, Rajan Chatterjee, Christian Federmann, Barry Haddow, Chris Hokamp, Matthias Huck, Varvara Logacheva, and Pavel Pecina (Eds.). Association for Computational Linguistics, Lisbon, Portugal, 392–395. doi:10.18653/v1/W15-3049

[14] Severin Primbs, Benedikt Fein, and Gordon Fraser. 2025. AsserT5: Test Assertion Generation Using a Fine-Tuned Code Language Model. In *2025 IEEE/ACM International Conference on Automation of Software Test (AST)*. 12–23. doi:10.1109/AST66626.2025.00008

[15] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method

for Automatic Evaluation of Code Synthesis. *CoRR* abs/2009.10297 (2020). arXiv:2009.10297 https://arxiv.org/abs/2009.10297

[16] 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), 85–105. doi:10.1109/TSE.2023.3334955

[17] Jiho Shin, Hadi Hemmati, Moshi Wei, and Song Wang. 2024. Assessing Evaluation Metrics for Neural Test Oracle Generation. *IEEE Transactions on Software Engineering* 50, 9 (2024), 2337–2349. doi:10.1109/TSE.2024.3433463

[18] André Silva, Nuno Saavedra, and Martin Monperrus. 2024. GitBug-Java: A Reproducible Benchmark of Recent Java Bugs. In *Proceedings of the 21st International Conference on Mining Software Repositories* (Lisbon, Portugal) *(MSR '24)*. Association for Computing Machinery, New York, NY, USA, 118–122. doi:10.1145/3643991.3644884

[19] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1800, 14 pages.

[20] Qinghua Xu, Guancheng Wang, Lionel Briand, and Kui Liu. 2026. Hallucination to Consensus: Multi-Agent LLMs for End-to-End JUnit Test Generation. *ACM Trans. Softw. Eng. Methodol.* (March 2026). doi:10.1145/3803418 Just Accepted.

[21] Junwei Zhang, Xing Hu, Xin Xia, Shing-Chi Cheung, and Shanping Li. 2026. Automated Unit Test Generation via Chain-of-Thought Prompt and Reinforcement Learning from Coverage Feedback. *ACM Trans. Softw. Eng. Methodol.* 35, 4, Article 92 (March 2026), 30 pages. doi:10.1145/3745765

[22] 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 Trans. Softw. Eng. Methodol.* 33, 4, Article 86 (April 2024), 32 pages. doi:10.1145/3635713

[23] Yucheng Zhang and Ali Mesbah. 2015. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 214–224. doi:10.1145/2786805.2786858

[24] Kun Zhou, Yutao Zhu, Zhipeng Chen, Wentong Chen, Wayne Xin Zhao, Xu Chen, Yankai Lin, Ji-Rong Wen, and Jiawei Han. 2023. Don't Make Your LLM an Evaluation Benchmark Cheater. arXiv:2311.01964 [cs.CL] https://arxiv.org/abs/2311.01964