A Comparative Study on the Efficiency of EvoSuite and GPT-3.5-Turbo for Unit Test Generation

Minggang Xie Chartchai Doungsa-ard Passakorn Phannachitta
College of Arts, Media and Technology, Chiang Mai University
Chiang Mai, Thailand
xmg2004@outlook.com

***Abstract*—Automated unit testing is essential for software quality, yet evaluating the true efficiency of generation tools remains challenging. This paper compares the test generation efficiency of a traditional search-based tool (EvoSuite) and an LLM (GPT-3.5-Turbo) on 15 Java files across varying structural complexities. We evaluate efficiency through statement/branch coverage, test pass rates, and the impact of iterative prompting. Quantitatively, EvoSuite demonstrates superior initial efficiency, achieving high coverage and nearly perfect pass rates, whereas GPT-3.5-Turbo requires multiple iterations to close the structural gap. However, our qualitative analysis of failed assertions reveals that this efficiency comparison masks a "pass rate illusion." While EvoSuite relies heavily on defensive assertions for runtime stability, iteratively refined GPT-3.5-Turbo tests attempt to verify actual semantic correctness, despite exhibiting more Expected Result Errors. These findings suggest that evaluating test generation efficiency must go beyond traditional coverage and pass rate metrics to include assertion quality.**

***Index Terms*—Unit Test Generation, Large Language Models, GPT-3.5-Turbo, EvoSuite, Search-Based Software Testing, Assertion Quality**

I. Introduction

Unit testing is a critical practice for ensuring software reliability, but manually creating comprehensive test suites is notoriously labor-intensive [1], [2]. To reduce this developer burden, automated unit test generation has become a vital necessity in software engineering.

Historically, Search-Based Software Testing (SBST) tools like EvoSuite [1] have dominated this field. These tools treat test generation as an optimization problem to maximize structural coverage [2]. However, traditional search-based methods have notable drawbacks: the generated tests are often difficult for developers to comprehend [3]. More importantly, because they generate assertions based on existing execution paths rather than intended specifications, they act primarily as regression safeguards to prevent unexpected crashes rather than true validators of business logic.

Recently, Large Language Models (LLMs) such as GPT-3.5-Turbo have shown immense potential in software engineering [4]. Unlike SBST tools, LLMs can infer developer intent from code semantics and naming conventions. This enables them to generate human-readable assertions that attempt to evaluate actual input-output relationships, mimicking human reasoning.

To address these gaps, this paper presents a comprehensive empirical comparison of EvoSuite and GPT-3.5-Turbo for automated unit test generation. Using a curated dataset of 15 Java files, we evaluate both approaches in terms of structural coverage, test pass rates, and the effects of iterative prompt refinement. To move beyond superficial metrics, we further introduce a root-cause classification framework, including Expected Result Error, Actual Result Error, Condition Error, and Exception Error, to analyze assertion failures and assess their underlying quality. Through this analysis, we aim to provide deeper insights into the effectiveness of different test generation paradigms. In particular, we identify a critical phenomenon termed the "Pass Rate Illusion," where the high pass rates achieved by traditional SBST tools can be misleading, as they mainly reflect the absence of runtime failures rather than true validation of program logic. In contrast, our findings suggest that iteratively refined LLM-based approaches, despite producing lower pass rates, are more capable of evaluating intended business logic, highlighting the importance of considering assertion quality alongside traditional efficiency metrics.

II. Background and Related Work

A. Search-Based Software Testing (SBST)

Search-based software testing (SBST) techniques frame testing as an optimization problem for generating unit test cases. The goal of SBST is to generate effective test suites that improve code coverage and efficiently reveal program errors by using algorithms to traverse the problem space [2].

EvoSuite is an automated test generation tool which utilizes SBST. It takes a Java class or method as input, uses search-based algorithms to create a test suite meeting specified criteria (e.g., code or branch coverage), and evaluates test fitness. Through iterative processes of variation, selection, and optimization, EvoSuite generates JUnit test cases [1]. While EvoSuite is highly efficient in generating test cases, its assertions are primarily designed to prevent unexpected crashes or runtime errors, rather than to verify the correctness of computed results. This fundamentally limits its utility in verifying actual business logic.

B. LLM Advancements in Software Engineering

Beyond daily use, large language models are increasingly applied in software engineering tasks, including code generation and summarization [3]. These models can also facilitate the generation of unit test cases, streamlining software validation processes. To comprehensively evaluate their utility, it is essential to examine two key aspects: first, their **Test Generation Capabilities**, which allow them to reason about code semantics and infer expected behaviors like human developers; and second, their **Current Limitations**, which highlight the ongoing challenges they face, such as predicting incorrect outcomes or missing environmental setups.

1) Test Generation Capabilities: Unlike traditional SBST tools, LLMs exhibit behavior that resembles that of a human developer who tries to reason about the intended functionality of the program and compute expected results accordingly. Rather than merely ensuring that the code runs without errors, they increasingly focus on verifying the correctness of input-output relationships. Furthermore, when comparing the initial test cases generated by ChatGPT with those produced after three rounds of prompt refinement, we can see a clear improvement in coverage and overall effectiveness [5].

2) Current Limitations: Despite these impressive capabilities, LLM-based test generation still faces distinct limitations. A primary challenge is that the implementation is correct, but the AI predicts an incorrect expected result. We classify this as an Expected Result Error (ERE). Additionally, there are instances where the logic of the test is fine, but the AI forgets to prepare the necessary data or environment before testing (like adding data to a database first), leading to Actual Result Errors (ARE). These limitations indicate that while LLMs are powerful, their initial outputs often require structural refinement and iterative prompting to achieve high reliability [4].

III. Methodology

To comprehensively compare the performance of EvoSuite and GPT-3.5-Turbo, we established a structured experimental methodology encompassing code categorization, data collection, prompt design, and multifaceted evaluation metrics.

A. Code Structure Categorization

To investigate how structural complexity affects test generation, this study introduces a self-defined classification scheme derived from the structural characteristics observed in the dataset. Based on these observations, the Java source files were grouped into three representative structural categories to better analyze the specific challenges encountered by automated test generation tools.

- **Category 1 (Monolithic logic):** The code logic is entirely concentrated at the entry point of the program (e.g., the `main` method). There are no custom methods or object state management, and all logic is executed sequentially.
- **Category 2 (Static methods and utility classes):** The code is composed primarily of static methods. Although the code resides within a class, the class itself is not instantiated as an object but instead serves mainly as a container for utility functions.
- **Category 3 (Strict object-oriented design):** The code defines a class containing private fields and instance methods. Data is encapsulated within the object and can only be accessed or modified through defined methods.

B. Data Collection

To construct our dataset, we sourced 15 representative Python scripts from the open-source GitHub repository, LLM-nirvana [6]. These scripts were manually reimplemented in Java to establish a controlled environment compatible with our target test generation tools. The resulting 15 Java files were then distributed across the three defined structural categories. Our selection criteria focused on ensuring diverse algorithmic logic and varying degrees of Cyclomatic Complexity. Based on our dataset analysis, the total Cyclomatic Complexity for Category 1 is 37, Category 2 is 68, and Category 3 reaches 110. This distribution provides a robust baseline for evaluating the test generation tools under different complexity thresholds.

C. Prompt Design for GPT-3.5-Turbo

To ensure reproducibility and effectively guide the LLM, we designed specific prompts using a role-playing and constraint-based strategy.

1) Initial Prompt: For the initial test generation, we provided the following prompt to establish the persona and strict output requirements:

> *You are an expert Java tester. Generate a JUnit 4 test class for '{class_name}' with package and class names exactly matching the target settings. Use standard JUnit imports, maximize branch coverage, and output only complete compilable Java test code for '{java_code}'.*

2) Iterative Prompt with Feedback: To address compilation errors, failed assertions, and uncovered branches from the first attempt, we utilized the following iterative prompt:

> *Iteratively improve the test class for '{class_name}' by fixing all reported compilation/runtime/assertion issues and adding tests for uncovered branches. Keep 'package {package_name};' and class name '{test_class_name}' exactly unchanged, and output only the full compilable Java test class for '{java_code}'.*

D. Evaluation Metrics

To go beyond superficial pass rates, the generated test suites were evaluated using the following metrics:

- **Structural Coverage:** We calculated both statement coverage and branch coverage to quantify the exploration of execution paths.
- **Test Pass Rate:** Measured as the percentage of successfully executed tests.
- **Failed Assertion Classification:** To conduct a root cause analysis of test failures, we categorized failed assertions into four types:

1) *Expected Result Error (ERE):* The implementation is correct, but the AI predicts an incorrect expected result.
2) *Actual Result Error (ARE):* The test logic is fine, but the AI forgets to prepare the necessary data or environment before testing. As a result, the actual result it gets is wrong or empty.
3) *Condition Error (CE):* The test fails because the true/false judgment is wrong. Even though the source code works well, the AI writes a bad test condition.
4) *Exception Error (EE):* The test waits for an error (exception) that never happens.

IV. Experimental Design and Research Questions

To comprehensively evaluate test generation capabilities, we conducted parallel experiments on the 15 Java files. For the baseline, we ran EvoSuite using its default search-based configuration. Specifically, the search budget was set to 60 seconds per class to ensure a fair and standardized execution environment. For GPT-3.5-Turbo, we first collected test suites generated from the initial prompt. We then executed these tests to collect compilation errors and failed assertions, feeding this feedback back to the LLM for three consecutive refinement iterations. Finally, we calculated statement and branch coverage using JaCoCo, recorded test pass rates, and conducted a manual root-cause classification (ERE, ARE, CE, EE) for all failed assertions. Based on this experimental setup, this study addresses the following three research questions (RQs):

- **RQ1 (Initial Performance):** How does the initial test generation performance of GPT-3.5-Turbo compare with the traditional SBST tool, EvoSuite, in terms of structural coverage and pass rates?
- **RQ2 (Iterative Improvement):** To what extent can iterative prompt refinements improve the coverage and test generation capabilities of GPT-3.5-Turbo?
- **RQ3 (Assertion Quality):** Beyond superficial pass rates, how do the two tools compare in terms of actual assertion quality and their ability to verify intended business logic?

V. Results

A. RQ1: Initial Performance Comparison

Table I presents the initial structural coverage and test pass rates for both test generation tools. In the initial scenario, EvoSuite demonstrated exceptional efficiency, achieving an average statement coverage of 96.6% and an average branch coverage of 90.2% across all code categories, accompanied by a perfect 100% test pass rate. In contrast, the initial tests generated by GPT-3.5-Turbo yielded significantly lower metrics, with an average statement coverage of 63.9%, a branch coverage of 66.3%, and an initial test pass rate of only 66.1%.

This contrast highlights the efficiency of search-based algorithms in path exploration and runtime stability under the initial setting. These results suggest that EvoSuite significantly outperforms GPT-3.5-Turbo in initial efficiency, achieving superior coverage and pass rates due to its immediate algorithmic exploration of execution paths.

B. RQ2: Performance Comparison After Prompt Update

Following three rounds of iterative prompt updates containing compilation errors, failed assertions, and uncovered branches, Table I shows a substantial increase in GPT-3.5-Turbo's performance. The average statement coverage for GPT-3.5-Turbo rose to 80.6%, and its branch coverage reached 87.0%, closely rivaling the performance of EvoSuite. However, despite this significant leap of approximately 20% in structural coverage, the overall test pass rate only marginally increased from 66.1% to 67.6%.

Iterative feedback closes most of the structural coverage gap, but the pass rate remains nearly unchanged. This indicates that while iteration substantially improves overall structural coverage—making the LLM's test suites more comprehensive—it does not significantly improve the overall pass rate, as the generated tests shift from syntax errors to deeper logical challenges.

TABLE I
AVERAGE STATEMENT AND BRANCH COVERAGE FOR EVOSUITE AND ITERATED GPT-3.5-TURBO ACROSS ALL CODE CATEGORIES.

Category	Avg Statement Coverage		Avg Branch Coverage	
	EvoSuite	GPT-3.5 (Iter)	EvoSuite	GPT-3.5 (Iter)
Category 1	100.0%	80.2%	96.9%	78.8%
Category 2	100.0%	77.9%	91.8%	90.7%
Category 3	93.0%	82.6%	86.4%	89.1%
Overall	96.6%	80.6%	90.2%	87.0%

C. RQ3: Assertion Quality and Pass Rate Illusion

The 100% pass rate of EvoSuite compared to the approximately 67% pass rate of GPT-3.5-Turbo requires a detailed breakdown of assertion failures. In our evaluation, EvoSuite generated exactly zero failed assertions. Conversely, Table II details the failure classifications for GPT-3.5-Turbo, showing that the iteration process doubled the number of Expected Result Errors (ERE) from 9 to 18.

This rise in EREs reveals a fundamental difference in test generation behavior. Rather than just writing tests that compile and pass, GPT-3.5-Turbo actively attempts to compute and assert the expected functional outcomes. This mimics human reasoning but naturally leads to more semantic mismatches and assertion failures when the LLM incorrectly predicts the business logic outcome.

At the same time, the increased ERE count also exposes a key limitation of LLM-based testing: when only source code context is provided and detailed external requirements are unavailable, the model is prone to hallucination and may infer incorrect business expectations. This suggests that future prompt design should incorporate explicit software specifications to better constrain expected-value generation.

TABLE II
FAILED ASSERTION CLASSIFICATION FOR GPT-3.5-TURBO BEFORE AND AFTER THREE PROMPT-REFINEMENT ITERATIONS.

GPT Config	Failed Assertion Classification				Total
	ERE	ARE	CE	EE	
Initial	9	0	3	3	15
After 3 Iterations	18	0	9	9	36

This underlying behavior is further corroborated by analyzing the generated assertion types presented in Table III. EvoSuite generated 30 instances of the `fail()` method. In contrast, the iterated GPT-3.5-Turbo generated 105 `assertEquals` statements.

EvoSuite makes extensive use of defensive mechanisms, where assertions act merely as safeguards against unexpected runtime exceptions. This ensures robustness and inflates the pass rate but ignores the intended specifications. On the other hand, GPT's heavy reliance on `assertEquals` prioritizes the actual verification of input-output correctness. These findings suggest that relying solely on pass rates to determine efficiency can be highly misleading; GPT-generated assertions are semantically richer and attempt meaningful validation despite exhibiting more failures.

TABLE III
DISTRIBUTION OF ASSERTION TYPES GENERATED BY EVOSUITE AND GPT-3.5-TURBO IN INITIAL AND ITERATED SETTINGS.

Assert Type	EvoSuite	GPT-3.5-Turbo	
		Initial	Iterated
`assertEquals`	25	50	105
`assertTrue`	10	15	21
`assertFalse`	11	9	10
`fail()`	30	5	16
`assertThrows`	0	0	3

VI. Discussion

Regarding efficiency, EvoSuite rapidly achieves near-perfect zero-shot coverage. Conversely, GPT-3.5-Turbo requires iterative feedback to close this gap. While iteration improves structural coverage, the pass rate stagnates. This is because the prompt provides only syntactic and structural feedback. Without explicit software specifications, the LLM lacks the ground-truth business rules to resolve semantic mismatches. This implies that structural iterations alone cannot overcome logical verification challenges.

Furthermore, quantitative metrics mask the "Pass Rate Illusion." EvoSuite's near-perfect pass rate is artificially inflated by defensive mechanisms (e.g., `fail()`) that prevent crashes but fail to verify actual computed results. In contrast, GPT-3.5-Turbo exhibits a higher failure rate driven by Expected Result Errors (ERE). This implies that the LLM actively attempts to simulate human-like validation by independently predicting expected outcomes. When its reasoning deviates from undocumented business logic, an ERE occurs. Thus, efficiency evaluations must incorporate assertion quality.

While these findings offer critical insights, this study is subject to several threats to validity. Regarding external validity, our 15-file dataset, though structurally diverse, may not fully capture the intricacies of large-scale industrial systems, requiring broader validation in future work.

In terms of internal validity, the manual classification of failed assertions introduces a risk of subjective bias. To mitigate this threat, we rigorously adhered to consistent decision criteria throughout the root-cause analysis.

Finally, concerning construct validity, our experimental setup—contrasting EvoSuite's 60-second budget with GPT-3.5-Turbo's iterations—intentionally prioritizes evaluating final assertion quality over strict computational equivalence. Furthermore, the limited sample size necessitates presenting observable descriptive trends rather than formal statistical significance.

VII. Conclusion and Future Work

This study evaluated the unit test generation efficiency of EvoSuite and GPT-3.5-Turbo. While EvoSuite dominates in initial efficiency—rapidly achieving near-perfect structural coverage and a 100% pass rate—GPT-3.5-Turbo requires multiple iterations to achieve comparable structural results.

Crucially, relying solely on these metrics creates a "Pass Rate Illusion." EvoSuite's perfect pass rate is maintained by defensive assertions that ensure runtime stability but fail to verify business logic. Conversely, GPT-3.5-Turbo's tests, despite yielding more Expected Result Errors, actively reason about functional correctness, offering greater practical value for fault detection. Therefore, future efficiency evaluations must integrate assertion quality and root-cause failure analysis.

Future work will explore the scalability of this approach on complex industrial projects and investigate a synergistic framework: utilizing GPT-3.5-Turbo for logic-oriented assertions alongside EvoSuite for structural coverage and runtime stability.

References

[1] G. Fraser and A. Arcuri, "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*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 416–419. [Online]. Available: https://doi.org/10.1145/2025113.2025179

[2] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–12.

[3] S. Bhatia, T. Gandhi, D. Kumar, and P. Jalote, "Unit test generation using generative ai : A comparative performance analysis of autogeneration tools," in *Proceedings of the 1st International Workshop on Large Language Models for Code*, ser. LLM4Code '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 54–61. [Online]. Available: https://doi.org/10.1145/3643795.3648396

[4] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng, “No more manual tests? evaluating and improving chatgpt for unit test generation,” 2024. [Online]. Available: https://arxiv.org/abs/2305.04207

[5] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, “Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 919–931.

[6] Rey-2001, “Llm-nirvana: A repository of python scripts,” https://github.com/Rey-2001/LLM-nirvana, 2024.