

HUMAN AND LLM-GENERATED TESTS FOR JAVA: A PRELIMINARY STUDY USING MUTATION TESTING

Pham Thi Thuong*, Ngo Thi Lan Phuong

TNU - University of Information and Communication Technology

ARTICLE INFO		ABSTRACT
Received:	12/5/2026	Recently, automated testing using Large Language Models (LLMs) has emerged as a promising approach to improving software test suites. However, the actual benefits of combining LLM-generated tests with human-written tests in terms of mutation testing remain an open issue. This paper focuses on addressing this issue. We experimented on a Java project using Maven, with three test case generation configurations for the project: (1) Human-written test cases (HumanTest), (2) LLM-generated test cases (LLMTest), and (3) a combination of human and LLM (HumanLLMTest+). Using the PIT mutation testing tool, we evaluated the three configurations with line coverage, mutation coverage, test strength, and execution time. The results showed that the HumanLLMTest+ was more efficient than the other approaches in terms of mutation points, while ensuring that execution time did not increase significantly. Furthermore, we found that LLMTest should be used as a test enhancement agent, supplementing for human-written test suites.
Revised:	29/6/2026	
Published:	30/6/2026	
KEYWORDS		
Large Language Models		
LLM-generated tests		
Mutation Testing		
Unit Testing		
Software Testing		

KIỂM THỬ DO CON NGƯỜI VÀ LLM TẠO RA CHO JAVA: MỘT NGHIÊN CỨU BAN ĐẦU SỬ DỤNG KIỂM THỬ ĐỘT BIẾN

Phạm Thị Thương*, Ngô Thị Lan Phương

Trường Đại học Công nghệ thông tin và Truyền thông – ĐH Thái Nguyên

THÔNG TIN BÀI BÁO		TÓM TẮT
Ngày nhận bài:	12/5/2026	Gần đây, kiểm thử tự động sử dụng các mô hình ngôn ngữ lớn (LLMs) đã nổi lên như một phương pháp đầy hứa hẹn để cải thiện bộ kiểm thử phần mềm. Tuy nhiên, lợi ích thực sự của việc kết hợp các bài kiểm thử do LLMs tạo ra với các bài kiểm thử do con người viết về mặt kiểm thử đột biến vẫn là một vấn đề chưa được giải quyết. Bài báo này tập trung vào việc giải quyết vấn đề này. Chúng tôi đã tiến hành thử nghiệm trên một dự án Java sử dụng Maven, với ba cấu hình tạo trường hợp kiểm thử cho dự án: (1) Các trường hợp kiểm thử do con người viết (HumanTest), (2) Các trường hợp kiểm thử do LLMs tạo ra (LLMTest), và (3) Sự kết hợp giữa con người và LLMs (HumanLLMTest+). Sử dụng công cụ kiểm thử đột biến PIT, chúng tôi đã đánh giá ba cấu hình này về độ phủ dòng lệnh, độ phủ đột biến, độ mạnh của kiểm thử và thời gian thực thi. Kết quả cho thấy HumanLLMTest+ hiệu quả hơn đáng kể so với các cấu hình khác về số điểm đột biến, đồng thời đảm bảo thời gian thực thi không tăng lên đáng kể. Hơn nữa, chúng tôi nhận thấy rằng LLMTest nên được sử dụng như một tác nhân tăng cường kiểm thử, bổ sung cho các bộ kiểm thử do con người viết.
Ngày hoàn thiện:	29/6/2026	
Ngày đăng:	30/6/2026	
TỪ KHÓA		
Mô hình ngôn ngữ lớn		
Các ca kiểm thử sinh bởi LLMs		
Kiểm thử đột biến		
Kiểm thử đơn vị		
Kiểm thử phần mềm		

DOI: <https://doi.org/10.34238/tnu-jst.15725>

* Corresponding author. Email: ptthuong@ictu.edu.vn

1. Introduction

LLMs are currently being widely studied for the automated test case generation problem. There is growing industry evidence showing that LLM-generated tests can significantly improve test suites and are being accepted in practice. Recent studies at Meta have shown that LLM-based tools can automatically extend or improve human-written unit tests. Furthermore, mutation-guided LLM-based test generation methods show that LLMs can be used to synthesize realistic and contextual mutants, thereby generating tests that aim to “kill” these simulated bugs, thereby enhancing the ability to resist privacy or security vulnerabilities [1]-[3].

However, to our knowledge so far, most current research on LLM-generated tests to supplement human-written tests focuses on coverage measures (e.g., line or branch coverage) or qualitative assessments of engineer acceptance. Direct empirical comparisons of mutation testing efficiency between LLM-generated and human-written test cases remain relatively rare. In particular, it is unclear whether LLM-generated test cases detect different types of errors, whether they reveal new types of vulnerabilities in the source code, or simply re-examine behaviors already covered by human-written tests [1], [2], [4]. These gaps are noteworthy because mutation testing provides a more nuanced measure of test quality than mere structural coverage metrics. For example, two test cases with comparable coverage may differ significantly in their ability to detect injected errors.

Mutation testing evaluates the quality of the test suite by introducing small syntax changes (mutants) into the program and checking whether the test suite can distinguish the modified program from the original program. A mutant is considered killed if the test cases detect it, i.e., the program detects a difference in behavior; otherwise, if it is not detected, the mutant is considered to survive. The mutation score is defined as the ratio of the number of killed mutants to the total number of mutants injected into the program [5].

Recent studies on LLM-based mutation testing have been moving toward evaluating test cases generated by LLM mutation techniques [2]-[4]. However, these evaluations often consider LLM-generated test cases independently, without a thorough analysis or systematic description of how LLMs complement, compete with, or extend the error detection capabilities of manual tests.

This paper aims to address one of the issues mentioned above; specifically, we focus on answering the following three main questions:

- RQ1: How does an LLM-only test suite compare to a human-written baseline in terms of mutation score and coverage?
- RQ2: Does combining LLM-generated tests with human-written tests improve mutation score, and at what computational cost?
- RQ3: Which types of mutants are killed by LLM-generated tests compared to human-written tests?

In this study, we used PIT to inject and measure mutation points. PIT is a widely used mutation testing tool for Java, supporting many different mutator groups, including conditional mutators, return value mutators, mathematical mutators, and null/void mutators. In addition, PIT also reports metrics such as line coverage, mutation coverage, and test strength for data analysis [6].

Some recent studies on exploiting LLMs for the problem of generating unit tests, property-based tests, and often evaluate test performance through coverage or error detection capability. However, coverage measures such as line coverage or branch coverage mainly reflect the degree of structural execution of the program, and do not always correlate strongly with the error detection performance of the test suite [7], [8]. Therefore, in this study, we combine traditional coverage measures and mutation scores to more comprehensively assess the quality of test cases generated by LLMs and the configurations tested.

The main contributions of this paper:

- An experimental comparison of three configurations: **HumanTest**, **LLMTest**, and **HumanLLMTest+** on the same Java Maven project, using the PIT mutation testing engine.

- A systematic analysis of experimental results to show that LLM-generated tests significantly improve mutation scores without increasing line coverage and execution time.
- Recommendations on integrating LLMs as supplementary agents to enhance the power of human-generated test suites.

2. Our Approach

2.1. Data Set

In this preliminary study, we tested generating test cases for a Maven project, whose source code repository is located on GitHub at the URL: <https://github.com/PhamThuongBlog/my-app.git>. This is a simple library management code repository with three main classes: Book, LoanService, and App. The project is small enough to allow manual testing of mutants and test cases, but still complex enough to represent the control flow, state changes, and boundary conditions necessary for test quality assessment.

2.2. Experimental Setup

We experimented with three test configurations as follows:

1. Human baseline (called **HumanTest***): Includes manually written JUnit tests, focusing on core functional scenarios (happy paths and some negative cases), representing typical programmer test cases.
2. LLM-only (called **LLMTest**): Includes test classes generated entirely by a genetic LLM model, without any human-written tests. These tests are generated based on a structured prompt that requires the model to analyze uncovered branches and missing asserts, then generate corresponding JUnit 5 tests.
3. Human plus LLMs (called **HumanLLMTest+**): A combination of Human and LLM-only test classes, where LLM-generated tests are added to the existing human-written tests.

We use a generic LLM with the ability to generate test cases. The generated test cases are manually checked for successful compilation (test cases generated by the LLMs will be manually checked by humans to see if they can be compiled), then integrated into the Maven test suite as classes such as *BookLLMTest*, *LoanServiceLLMTest*, and similar classes.

The experimental setup applied to these test configurations as shown in Table 1.

Table 1. Experimental setup

Tool	Description
Build system	Apache Maven 3.9.9 (Java 17.0.16; aligned with project configuration)
Mutation tool	PIT 1.16.0 configured via the pitest-maven plugin
Test framework	JUnit 5
Environment	CPU: modern multi-core processor (8 cores) Memory: ≥ 8 GB OS: Ubuntu Linux on WSL2
PIT configuration	Target classes: com.mycompany.app.* (Book, LoanService, App).
LLM model	codellama:7b-code

Note: Each configuration is executed separately, with PIT runtime recorded via the generated HTML report and console logs.

2.3. LLM Prompt

We use a general LLM model with the following prompt template to generate tests:

You are given a Java class and its existing JUnit 5 tests.

Your task:

1. *Analyze uncovered branches and missing assertions.*
2. *Generate additional JUnit 5 tests that:*

- Kill surviving PIT mutants*
- Cover boundary conditions and null cases*

3. Do NOT duplicate existing tests.

4. Each test must be compilable and deterministic.

Return only Java test code.

The generated tests are manually checked by humans to see if they can be compiled and integrated into the Maven test suite as BookLLMTest, LoanServiceLLMTest, ...

2.4. Evaluation Metrics

In order to evaluate the effectiveness of the experimental configurations, we use the evaluation metrics provided by the PIT Mutation Testing tool, including: Line Coverage, Mutation Coverage, Mutants Killed, Test Strength, and PIT Runtime.

2.3.1. Line Coverage Metric

Line Coverage measures the percentage of lines of code executed by test cases. This metric represents the level of code coverage of the test suite. The formula is as follows:

$$\text{Line Coverage} = \frac{\text{Covered Lines}}{\text{Total Lines}} \times 100\% \quad (1)$$

Where: *Covered Lines* is the number of lines of code executed by the test, *Total Lines* is the total number of lines of code.

2.3.2. Mutation Coverage metric

Mutation Coverage measures the percentage of mutants detected by test cases out of the total number of mutants generated. This is an important metric for evaluating the error detection capability of a test suite. The formula is as follows:

$$\text{Total Lines} = \frac{\text{Killed Mutants}}{\text{Total Mutants}} \times 10\% \quad (2)$$

Where: *Killed Mutants* is the number of mutants killed, and *Total Mutants* is the total number of mutants born.

2.3.3 Mutants Killed metric

Mutants Killed indicates the number of mutants eliminated during mutation testing. This metric directly reflects the number of simulation errors detected by the testing system. The formula is as follows:

$$\text{Mutants Killed} = \text{Total Mutants} - \text{Survived Mutants} \quad (3)$$

Where: *Survived Mutants* is the number of mutants not detected after running the test case.

2.3.4. Test Strength metric

Test Strength measures the ratio of mutants killed to the number of mutants actually executed by the test case. This metric is used to evaluate the quality and strength of the test suite. The formula is as follows:

$$\text{Test Strength} = \frac{\text{Killed Mutants}}{\text{Covered Mutants}} \times 100\% \quad (4)$$

Where: *Covered Mutants* is the number of mutants executed by the test case.

2.3.5. PIT Runtime metric

PIT Runtime represents the total execution time of mutation testing. This metric is used to evaluate the computational cost and performance of the experimental configuration. The formula is as follows:

$$PIT Runtime = End Time - Start Time \quad (5)$$

Where: *Start Time* is the time when PIT starts running, *End Time* is the time when mutation testing ends.

3. Results and Discussion

3.1. Experimental Results

3.1.1. Overall Mutation and Coverage Metrics

Table 2 summarizes the key metrics for the three configurations (extracted from PIT reports: index.html and mutations.csv).

Table 2. Mutation & Coverage Metrics

Metric	HumanTest	LLMTest	HumanLLMTest+
Line coverage	77% (36/47)	74% (35/47)	83% (39/47)
Mutation coverage	64% (23/36)	47% (17/36)	75% (27/36)
Mutants killed	23	17	27
Test strength	85%	65%	96%
PIT runtime	~9 s	~2 s	~10 s

From the experimental results, we observe that **LLMTest** achieved 74% of line coverage (35/47 lines), 47% mutation coverage (17/36 mutants), and 65% test strength. These values are lower than **HumanTest**, which achieved 77% of line coverage (36/47), 64% mutation coverage, and 85% test strength. This indicates that, when used independently, tests generated by **LLMTest** are still not as effective as manually written tests in terms of error detection.

A deeper analysis of the mutant set revealed that **HumanTest** killed 23 different mutants, while test cases generated by **LLMTest** killed an additional 4 mutants that the Human test set did not detect. Therefore, the combined **HumanLLMTest+** configuration gives a total of 27/36 mutants were killed, corresponding to a mutation score of 75%. This result shows that LLM-generated tests have a clear complementary value to the human-written test set: although **LLMTest** alone is less effective than **HumanTest**, when integrated, **LLMTests** show promising improvements in the test set's error detection capabilities.

Another noteworthy point is that the execution time of **HumanLLMTest+** is about 10 seconds, therefore a negligible difference compared to **HumanTest** is about one second, while mutation score increased from 64% to 75%. This suggests that the execution time cost of integration is relatively small compared to the benefits gained in error detection. In other words, while **LLMTest** is less efficient on its own, when used as an enhancement layer for **HumanTest**, it improves test quality at almost the same time cost.

3.1.2. Class-Level Analysis

From the experimental results, under the **HumanTest**, PIT reports the class-wise mutation scores summarized in Table 3.

From the experimental results shown in Table 2 and Table 3, we observed a gap between line coverage (77%) and mutation coverage (64%) with the HumanTest configuration. This indicates that a significant number of injected errors remain undetected by the test suite, despite the relatively wide scope of code execution (77%). The "survived mutants" (meaning "no-coverage mutants") suggest that many code locations were not touched by any test cases. Therefore, it has been demonstrated that the human-generated test suite is still weak.

Table 3. Class-Level Mutation Scores (HumanTest)

Class	Total mutants	Killed	Survived	No coverage	Mutation score
Book	19	10	1	8	52.6%
LoanService	12	9	2	1	75.0%
App	5	4	1	0	80.0%
Overall	36	23	4	9	64.0%

3.1.3. Survived Mutants and Missing Tests (Human Baseline)

We will systematically analyze the mutants that survived HumanTest, to identify the missing tests and areas that need further design to "kill" them.

1) Book.java: Incompletely specified accessor methods

A mutant survived in the *Book.getIsbn()* method associated with the transformation operator: *EmptyObjectReturnValsMutator*. The root cause is that no tests explicitly assert the return value of *getIsbn()*: ISBNs are only used to initialize objects, not to verify behavior.

This suggests a missing test that kills this mutant is:

```
@Test
void getIsbn_shouldReturnExactValue() {
    Book book = new Book("ISBN-1", "T", "A", LocalDate.now(), 1);
    assertEquals("ISBN-1", book.getIsbn());
}
```

This shows that even seemingly simple getter methods can hide errors if not explicitly asserted.

2) LoanService.java: Boundary Cases and Null Values

Two surviving mutants were observed in the *LoanService* class, namely:

(1) Constructor (*ConditionalsBoundaryMutator*) - the constructor imposes the condition *defaultLoanDays > 0*, however no tests cover the cases *defaultLoanDays == 0* or *< 0*.

Missing test:

```
@Test
void constructor_withZero_shouldThrow() {
    assertThrows(IllegalArgumentException.class,
        () -> new LoanService(0));
}
```

(2) *isOverdue()* (*BooleanTrueReturnValsMutator*) - This method returns false when *due == null*, representing the state "no borrowing has occurred," but there are no test cases for this scenario.

Missing test:

```
@Test
void isOverdue_whenNoLoan_shouldReturnFalse() {
    LoanService service = new LoanService(7);
    Book book = new Book("I1", "T", "A", LocalDate.now(), 1);
    assertFalse(service.isOverdue(book, "u1", LocalDate.now()));
}
```

The above analysis suggests that mutants often survive due to missing testing for boundary cases and null-related scenarios, which humans overlook.

3.1.4. LLMTest

From Table 2, we see that **LLMTest** achieved 74% of line coverage (35/47 lines), 47% mutation coverage (17/36 mutants), and 65% test strength, with a PIT runtime of approximately 2 seconds. Thus, structural coverage is relatively high; however, mutation coverage is much lower

than **HumanTest** (64%). This indicates limitations in the error detection capabilities of the test suite generated by **LLMTest**.

Table 4 shows mutants killed by **LLMTest** that **HumanTest** missed.

Table 4. *Mutants killed by LLMTest that HumanTest missed.*

Class	Method	Mutator	Description
Book	getIsbn()	EmptyObjectReturnVals	Human does not test for empty-string.
Book	returnBook()	MathMutator	Missing state update test
LoanService	<init>	ConditionalsBoundary	Missing test boundary constructor
LoanService	isOverdue()	BooleanTrueReturn	LLMTest generates negative logic coverage tests.

From experimental results shown in this table, we see that **LLMTest** is often powerful when designing test cases for: boundary; boolean return; state mutation; constructor contracts because it kills a significant subset of mutants belonging to these groups. However, a considerable number of mutants related to boundary semantics, time logic (e.g., date-based overdue checks), and the removal of side effects in void methods still survive under the LLM-only configuration. This behavior may be related to the probabilistic generation mechanism of LLMs: when operating independently, LLM-generated tests tend to focus on surface specifications and local invariants, and often struggle to fully capture the specific business processes and real-time behavior encoded in the business logic. Therefore, in this context, LLM-only tests cannot yet replace human-written tests. Instead, their true value emerges when combined with human-built tests, as demonstrated in the **HumanLLMTest+** configuration.

3.1.5. Effect of LLM-Generated Tests (HumanLLMTest+)

After adding LLM-generated tests, PIT reports: Line coverage: 83% (39/47), mutation coverage: 75% (27/36), and test strength: 96%. Therefore, LLM-generated tests primarily kill mutants that were previously either survived or no-coverage under Human tests, especially in the categories as shown in Table 4. Furthermore, from Table 2, we also find that the improvement in mutation coverage does not stem from a significant increase in line coverage. Line coverage remains almost unchanged, at approximately 83%. Meanwhile, mutation coverage increased from 64% (**HumanTest**) to 75% (**HumanLLMTest+**). Therefore, tests generated by **LLMTest** primarily improve behavioral completeness rather than expanding structural coverage.

To better understand which types of error benefit most from LLM-generated tests, we grouped mutants by mutator type. Experimental results show that LLM-generated tests significantly improve mutant kill rates for the following groups: (1) conditional boundary mutators; (2) return-value mutators; and (3) math mutators. Conversely, mutants involving time logic and complex state transitions remain a major challenge, where human-written tests generally perform better. This suggests a "specialization pattern": human-written tests tend to better capture business processes and time constraints in the application domain, while LLM-generated tests are particularly effective for local invariants and boundary contracts.

3.2. Discussion

Based on the experimental results, we can answer the following three research questions:

RQ1 – LLMTest vs. HumanTest in terms of mutation score and coverage? The **LLMTest** configuration achieved line coverage comparable to the **HumanTest** configuration, but mutation coverage was significantly lower (47% vs. 64%). This indicates that structural coverage, when used alone, does not accurately reflect the power of error detection. Therefore, test cases generated by **LLMTest**, when running independently, still fall short of the quality of manual testing in terms of mutation score and test strength.

RQ2 – What are the benefits and costs of **HumanLLMTest+** compared to individual configurations? When combining LLM-generated tests with **HumanTest**, the mutation score increases, while line coverage remains virtually unchanged and PIT runtime increases only slightly. Thus, **HumanLLMTest+** offers significant error detection benefits at a relatively low computational cost, suggesting that **LLMTest** is best suited as an enhancement layer, rather than a complete replacement for human-written tests.

RQ3 – Which type of mutant does LLM “kill” better? Analysis of mutants killed by **LLMTest** shows that **LLMTest** is particularly effective with constructor constraints, return value invariants, and simple state updates/arithmetic expressions. Conversely, mutants related to complex business processes and manual test-time logic are generally handled better. This gives **HumanLLMTest+** superiority over each individual configuration in both mutation score and test robustness.

4. Conclusion and Future Work

This study demonstrated the impact of LLM-generated tests on a Java Maven test project. Compared to the human-written test configuration (**HumanTest**), the combined **HumanLLMTest+** test suite significantly improved mutation scores without incurring substantial additional execution time costs. LLM-generated tests were particularly effective at “killing” mutations related to boundary conditions, boolean return values, and state updates – which are often missed by human-written tests – thereby highlighting the role of **LLMTest** as a test enhancement agent for human testing. Additionally, **LLMTest** should be used to suggest tests for boundary cases, getters, setters, constructor constraints, and return-value assertions.

This study focuses on a small to medium-sized project with the aim of thoroughly researching and analyzing the test case generation process and test generation efficiency of a representative LLM model, *codellama:7b-code*. In the future, we will continue to expand the research for more code repositories projects, on larger and more diverse systems. Simultaneously, we will conduct experiments on various LLM models to obtain more objective and reliable assessments of the test case generation capabilities of LLM models.

REFERENCES

- [1] N. Alshahwan *et al.*, “Automated unit test improvement using large language models at meta,” in *Companion Proc. 32nd ACM Int. Conf. Foundations of Software Engineering*, 2024, pp. 185-196.
- [2] M. Harman *et al.*, “Mutation-guided llm-based test generation at meta,” in *Proc. 33rd ACM Int. Conf. Foundations of Software Engineering*, 2025, pp. 180-191.
- [3] F. Tip, J. Bell, and M. Schäfer, “Llmorpheus: Mutation testing using large language models,” *IEEE Trans. Software Engineering*, vol. 51, pp. 1645-1665, 2025.
- [4] P. F. M. Cabral *et al.*, “Evaluating LLM-generated unit tests with mutation testing: ChatGPT vs DeepSeek,” in *Proceedings of the 24th Brazilian Symposium on Software Quality (SBQS 2025)*, 2025, pp. 120-130.
- [5] M. T. Valente, “Software Engineering: A Modern Approach,” 2024. [Online]. Available at: <https://softengbook.org/>. [Accessed Apr. 10, 2026].
- [6] S. T. E. Abbas *et al.*, “Investigation on java mutation testing tools,” *JOIV: Int. J. Informatics Visualization*, vol. 6, pp. 455-462, 2022.
- [7] R. Yang, X. Xu, and R. Wang, “TestLoter: A logic-driven framework for automated unit test generation and error repair using large language models,” *Journal of Computer Languages*, vol. 84, 2025, Art. no. 101348.
- [8] J. A. Pizzorno and E. D. Berger, “Coverup: Effective high coverage test generation for python,” *Proc. ACM Software Engineering*, vol. 2, pp. 2897-2919, 2025.