LLM-Based Test Generation from Bytecode

An Empirical Comparison with EvoSuite and the Effect of Example Test Artifacts

Julian Louis Overmars

Supervisors: Sebastian Proksch, Cathrine Paulsen

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: Julian Louis Overmars
Final project course: CSE3000 Research Project
Thesis committee: Sebastian Proksch, Cathrine Paulsen, Soham Chakraborty

Abstract

Automated test generation for third-party libraries is essential for detecting behavioral changes introduced by updates. Existing tools like EvoSuite operate on bytecode, but achieve only moderate mutation scores, leaving room for improvement. Recent work has shown that LLMs can outperform EvoSuite on mutation score when given source code, but it is unclear whether these results hold when only bytecode is available. Additionally, LLM-generated tests suffer from hallucinations and low compilation rates, and it is unknown whether providing example tests can mitigate these issues. This paper investigates whether LLMs can generate effective test suites from bytecode alone, and whether adding test artifacts (source code or decompiled bytecode) improves compilation rates and overall test quality. I compare LLM-generated tests with EvoSuite across five Java libraries, evaluating line, branch, and method coverage, as well as mutation score. The results show that LLMs outperform EvoSuite on coverage for three of five libraries and achieve higher per-class mutation scores, though they struggle slightly more with very large classes. Adding example tests improves branch coverage across some libraries but decreases method coverage, with little difference between source code and bytecode examples. Test artifacts did not consistently improve compilation rates or overall quality. These findings suggest that LLM-based test generation on bytecode is a viable alternative to EvoSuite, and can even outperform it on certain libraries.

1 Introduction

Modern software heavily relies on third-party libraries, with studies estimating that 70-90% of applications consist of open-source components [12]. However, these applications expect certain behavior from these libraries, so when they are updated to fix bugs or security risks, their behavior can change, which potentially creates issues for the applications using them. In a study by Gyori et al. [8], they found that out of 408 popular Java projects, almost half were not able to update their dependencies to the latest version. Tests can detect behavioral changes in libraries when they are updated, helping developers detect changes that break the application. However, existing tests are often not available [8; 16], and writing such tests manually is very costly [10].

Automated test generation serves as a solution to this problem because it minimizes the cost of creating a test suite. Research has shown that LLMs can provide test suites that show improvements over existing tools in metrics like readability or code coverage; however, they face challenges like compilation errors due to the LLM hallucinating. Additionally, it is often assumed that context given to the LLM is the source code [3]. However, how LLMs perform when only given bytecode as context is still unclear, and bytecode has the advantage of always being available while source code is not.

Existing tools like EvoSuite address the issue of a lack of available source code, since it operates on bytecode directly. However, EvoSuite achieved an average of 77% line coverage, 71% branch coverage, and 53% mutation score per class in the SBST 2021 Tool Competition [20], leaving room for improvement. A recent paper titled Test Wars has found that LLM-generated test suites are similar to or even slightly better than EvoSuite in mutation score, despite having lower coverages [2], suggesting LLMs produce more meaningful tests. These findings raise the question of whether these results hold when LLMs are given only a bytecode representation rather than source code, an area that has not been explored yet in the field of LLM-based test generation. Additionally, LLM-generated test suites are known to suffer from hallucination of non-existent APIs [3], which can be mitigated by providing additional example tests as context.

This paper investigates the ability of LLMs to generate test suites based on a bytecode representation given by javap, a tool that extracts information out of raw bytecode. With the bytecode representation, it attempts to replicate the findings of Test Wars and see whether LLMs perform better on mutation score than EvoSuite. On top of that, it tries to solve the common issue of LLM-generated test suites not compiling due to hallucination by providing available example test cases, either provided as source code or by decompiling the compiled tests. This highlights the current knowledge gap; it is unclear whether LLMs can still provide high-quality test suites when provided with these different artifacts. Understanding whether LLM-generated tests remain effective when only bytecode is available is essential for assessing whether they are a viable alternative to tools like EvoSuite for libraries when source code is not available. At the same time, if providing example test artifacts can reduce hallucination and improve compilation rates, it would address one of the biggest barriers to adopting LLM-based test generation at scale.

This work aims to answer the question: "To what extent can LLMs provide an effective test suite for Java libraries when only the bytecode and test artifacts are available?" To answer this, the work is divided into 3 sub-questions.

- RQ1: How do LLM-generated tests on bytecode compare to EvoSuite?
- RQ2: Do example tests improve LLM-generated test quality, and does the type of example (source vs bytecode) matter?

RQ1 investigates whether the suggestion from Test Wars [2], they state that LLMs can potentially achieve higher mutation scores than EvoSuite despite lower coverage. In addition to mutation score, this question also compares line, branch, and method coverage to provide a complete picture of test quality. RQ2 examines whether adding test artifacts to the LLM prompts increases the generated code quality. The LLM is provided with either test source code or compiled test code, which are compared to determine which test artifact leads to the highest quality test suites.

For clarity, when the three configurations are compared, they are referred to throughout this paper as BC (bytecode only), BTB (bytecode + test bytecode), and BTS (bytecode + test source code).

This paper uses a structured pipeline to assess the quality of the generated test suites. The pipeline converts bytecode into a human-readable representation using javap. Test bytecode is decompiled before being added to the LLM prompt, whereas test source code is included directly. These artifacts are sent to the LLM when it is prompted to generate a test suite for a single method of a target class. The generated test suites are checked for compilation and test pass rates; if they fail, the LLM is re-prompted in a conversational style until success or until a maximum of 2 retry attempts is reached. Finally, the test suite is evaluated using the metrics described above.

The results showed that LLM-generated tests outperformed EvoSuite on coverage for three of five libraries, with improvements of 21-49%. For mutation score, the LLM achieved higher per-class averages than EvoSuite, but performed slightly worse on large classes with many mutants, resulting in aggregate scores that slightly favored the LLM. Adding example tests improved branch coverage on some libraries but decreased method coverage, and the choice between source code and bytecode examples made little difference.

The paper is organized as follows. Section 2 discusses related work, including EvoSuite, LLM-based test generation, and the Test Wars findings. Section 3 presents the methodology and experimental setup in detail, including the test generation pipeline, artifact processing, evaluation metrics, and motivation for the research questions. Section 4 reports and compares the experimental results for RQ1 (LLM vs EvoSuite on bytecode) and RQ2 (impact of example tests). Section 5 discusses ethical considerations and reproducibility. Section 6 interprets the results and summarizes the main findings. Finally, Section 7 concludes the work and proposes recommendations for future work.

2 Related Work

2.1 EvoSuite

EvoSuite is a search-based unit test generation tool that operates on bytecode directly. It uses a genetic algorithm that iteratively produces solutions that improve toward coverage targets [5]. It can generate tests for a target class completely automated, targeting metrics like line coverage, branch coverage, and mutation score [6]. Since the process is automated, EvoSuite runs its tests in a sandbox and detects and removes malicious tests [5]. In the Search-Based Software Testing (SBST) tool competition, EvoSuite is often the winner, or close to first place [20; 6]. In the results of the SBST 2021 competition, EvoSuite won with an average of 77% line coverage, 71% branch coverage, and 53% mutation score per class [20], which leaves major room for improvement, especially in the mutation score. Therefore, I use EvoSuite as the baseline against which LLM-generated tests are compared.

2.2 LLM-Based Test Generation

LLM-based test generation is on the rise, with many studies coming out on the topic as models become increasingly better [22]. One of the biggest advantages of this approach is that it can write more readable, correct, and usable tests; the LLMs follow common code practices when generating tests [3]. But they also face challenges like inconsistent performance, requiring significant computational resources, struggling to achieve high coverage, and generating tests with compilation errors due to hallucination invalidating the entire test suite [3; 21]. Studies by Siddiq et al. and Tang et al. even show LLMs performing worse than EvoSuite [17; 18]; however, Godage et al. [7] evaluated different models and showed they outperformed existing tools and that Claude 3.5 scored the best on the metrics. These contradictions likely stem from the recency of LLM-based test generation research, so there is not yet a clear methodology that is proven to work best. For example, LLMs are very sensitive to prompting; Chowdhury et al. showed that advanced prompting techniques can massively increase performance [4]. Thus, despite some negative results, it remains unclear whether LLMs are worse overall, and further investigation is needed. Most papers assume source code to be available and use it as context to give to the LLM [3], but, in practice, this is not always the case, for example, for third-party libraries, source code might not be available [16]. Whether LLMs still perform well when only bytecode is available needs further research.

2.3 LLM versus EvoSuite: Test Wars

Test Wars is a recent paper by Abdullin et al., a collaboration between JetBrains and TU Delft, comparing TestSpark (an LLM-based test generation tool) to EvoSuite [2]. They gave the LLM source code as context and evaluated the test suites on line coverage, branch coverage, and mutation score. The findings show that the LLM-generated tests achieved lower line and branch coverage than EvoSuite; however, TestSpark performed better in terms of median mutation score, but comparable on average. They suggest this is due to the LLMs potentially understanding the code better and creating more meaningful tests. In this paper, the experiment uses the same metrics to see if similar patterns hold in the new context of bytecode instead of source code. Test Wars also finds that the compilation rate of TestSpark is about 58% while EvoSuite and Kex got close to 100%. If the LLM gets extra context showing how methods should be invoked, it might cause the compilation rate to go up.

2.4 Bytecode Representation

Since LLMs require human-readable input and bytecode is not, a human-readable representation is needed to prompt them. There are tools for this purpose; however, the representation cannot fully recover all the information that was present in the source code. Comments, whitespace, and unreachable code are stripped during compilation and do not appear in any representation.

I use javap [14], the disassembler tool that comes with the JDK, to obtain a human-readable representation of compiled classes. Depending on the options used, javap prints class structure, including method signatures, fields, and bytecode instructions. For every method and field, javap prints the full type names (e.g., java.lang.String rather than just String). The tool can output anything from basic information, such as only public signatures, to very detailed output, such as all stack instructions. Unlike a decompiler, javap does not reconstruct

source code but exposes the high-level class blueprint and low-level metadata per method.

An alternative would be a decompiler like Fernflower [11], which attempts to reconstruct Java source code from bytecode. Fernflower reconstructs Java code directly from bytecode; while it does not contain more information than the bytecode itself, it tries to infer high-level structures from it. Where javap can print every bytecode instruction, Fernflower tries to combine the instructions into source code, hiding some low-level information but producing output closer to actual source code. However, I chose javap for two reasons: it runs reliably on all class files, and preliminary experiments showed the LLM performed better on javap output than Fernflower for this task.

3 Methodology

This section contains the experimental setup, which explains what the pipeline for achieving the results looks like and includes the details to replicate the research. Then the evaluation metrics are explained and justified. Finally, the goal of the research questions is specified and motivated.

3.1 Experimental Setup

A visualization of the pipeline is shown in Figure 1. The Java libraries are manually extracted; I picked those that have test sources available. The pipeline downloads the dependencies and then iterates over each class in the library. I use javap to get a human-readable text representation of the class bytecode, which can be used as input for the LLM. A prompt is created using the text representation of the entire class, instructing the LLM to generate a test suite for a specific method within the target class. This prompt is sent to the Deepseek v4 Flash LLM hosted on OpenRouter. The test suite in the response is run and checked if it is valid, meaning it compiles and runs. For invalid test suites (e.g., due to compilation errors, failing tests, or runtime errors), the LLM is re-prompted in a conversation-like style: I combine the original prompt, the LLM's response, the error message, and a new prompt instructing the LLM to repair its code. Retrying is limited to two times; preliminary tests showed that more did not increase the coverage significantly. If the test code is valid, the test is saved in a folder of test suites that pass. Once the LLM has attempted to generate a test suite for all methods, JaCoCo [9] is run on the test suites for that class, and the results are saved. The results include: the number of methods for which the LLM could generate valid tests and the line and branch coverage obtained from the test suites.

Pipeline for Generating Tests

The pipeline is adapted from a colleague's repository [1]. The branch I used for getting the final results is *"julian_result_branch_final"*. All results and code are also available to download on Zenodo [15]. The dataset used for the experiments is also available in the repository under the data folder. The repository contains 930 of the most popular Java libraries based on download statistics. For all experiments, I selected libraries that have test artifacts publicly available (both test source code and compiled test bytecode). This allows comparison between the configurations in RQ1 (bytecode only) and RQ2 (with example tests).

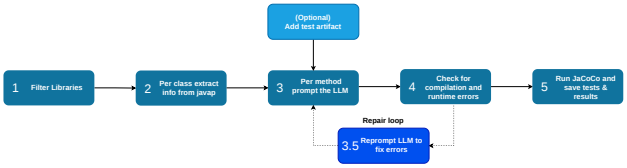

Figure 1: Overview of the test generation pipeline. For each method in a class, the LLM receives javap bytecode output and, optionally, a test artifact, then generates a test suite. If a test is invalid, the LLM is re-prompted up to two times. Valid tests are saved, and JaCoCo computes coverage after all methods are processed.

After a compiled library is downloaded as a JAR, the pipeline processes a maximum of 100 classes per library. Due to time constraints, I prioritized testing multiple libraries over fully testing a few large libraries, especially because results vary significantly per library. All classes are listed and shuffled in a random order with a fixed seed of 42 for reproducibility. All classes are considered for testing, including abstract classes, interfaces, and enums. However, if an abstract class or interface contains no testable static methods, it is skipped entirely.

One prompt is sent for each method in a class. In preliminary experiments where the LLM was prompted to generate a test suite for an entire class at once, compilation rates were very low. If any single test failed to compile or threw an error, the entire test suite was discarded. Prompting per method yields better results because methods the LLM cannot handle are simply skipped, keeping only valid tests. This also ensures that classes with many methods are tested properly, since the LLM is likely to fail for at least one method when generating for the entire class.

The prompt template (*few_shot.txt*) is available in the repository. The prompt first specifies output constraints requiring only Java code. Placing constraints at the start of the prompt follows findings that LLMs attend most strongly to information at the beginning of their input context [13]. Then it receives the javap output of the compiled class and is asked to generate a test suite for a specific method. javap produces a human-readable bytecode representation listing method signatures, bytecode instructions, and field references.

The flags used when running javap are: -classpath, -c, and -public. This output is put into the prompt and sent to the LLM. The experiments used javap from JDK 21. The model I used is Deepseek v4 Flash, accessed via OpenRouter. The returned test suite is placed in its own project and compiled and run using Maven version 3.9.12 to check validity.

If the generated tests are invalid, the LLM is re-prompted conversationally. The conversation history is included in the new prompt, along with the error message and instructions to fix the mistakes in the previously provided code. The LLM gets a maximum of 2 retries to fix compilation errors, and another 2 tries to fix runtime errors. Preliminary results showed that 2 retries significantly increased the valid test rate, but more than 2 did not yield further improvement, making additional retries not worth the extra time.

Valid tests are saved to a folder per class. Once all methods in a class have been processed, all valid test suites for that class are executed. Maven builds the project, and JaCoCo [9] computes line and branch coverage. The experiments use JaCoCo version 0.8.11. Using the JaCoCo reports, a script (also available in the repository) records the covered lines and total lines for every targeted class. It then calculates line and branch coverage only for the classes that were considered for testing, since some classes were skipped, and I tested a maximum of 100 classes per library.

Test Artifact Addition

RQ2 investigates whether adding test artifacts to the LLM input increases test quality. In Figure 1, the test artifact addition to the pipeline is depicted. The pipeline is similar to the one described above, but with extra context added to the LLM prompt. The prompt template for RQ2 is in *few_shot_tests.txt*. The pipeline does not automatically download test JARs or test sources from Maven Central; I manually add them to the project. When the code runs, it iterates over classes as usual. It is quite difficult to automate the process of finding relevant tests for a class, so I decided to use a simple approach: if a test class name matches the name of the class to be tested, that test class is used as the test representation. Since test classes can be very large and most of them are not relevant for every method, only the basic setup and the tests that call the target method are given to the LLM. This approach found a matching test class for more than 50% of classes.

Pipeline for Getting Mutation Score

To calculate the mutation score, I use PIT version 1.15.3. I manually add the library source code to a folder. For the LLM-generated tests, the valid tests are saved per class. A script (available on Zenodo [15]) copies the source code of the target class to the class folder, compiles the tests using Maven, and runs PIT on the tests to compute the line coverage and mutation score. Results are gathered per class. Because test suites are generated per method, all valid method-level test suites for a class are combined and run together to compute the class-level mutation score. For the tests generated by EvoSuite, the script is a little different; instead of running Maven per class, all tests and sources are compiled at once, and then PIT runs on each class individually. However, since EvoSuite and PIT can have different dependency requirements, I sometimes had to download the class sources pre-compiled and switch between Java 8 and 21 to make different libraries work. But different Java versions likely do not affect coverage or mutation scores.

I did not manage to get PIT to work on all classes. Some classes returned a line coverage and mutation score of 0, even though tests were available and clearly called the target class. Therefore, I deleted all classes returning 0% mutation score from the results. This most likely happened because the script failed to find the tests, the tests did not compile correctly, or they produced runtime errors.

3.2 Evaluation Metrics

To assess test suite quality, I use some of the metrics from Test Wars [2], specifically those for evaluating test effectiveness rather than metrics regarding the target class. The main finding from Test Wars that this work builds upon is that LLM-generated tests can achieve higher mutation scores even when their line coverage is lower.

- **Code coverages**
  - **Line coverage;** it is the most basic metric for test completeness, but does not capture logical complexity. A high line coverage shows the test covers every aspect of a class, but does not indicate test thoroughness.
  - **Branch coverage;** it is a commonly used metric to evaluate code coverage because it is a stronger indicator of test thoroughness than line coverage. High branch coverage indicates that methods are invoked multiple times with different inputs, testing various expected behaviors.
  - **Method coverage;** if a method is never called, tests cannot detect bugs in it, regardless of their quality. High method coverage shows that different methods are tested, but not how thoroughly.
- **Mutation score;** evaluates test effectiveness by seeding small faults (mutants) into the code and checking whether the tests detect them. A higher mutation score means the tests are more sensitive to behavioral changes.

Together, these metrics can show the true quality of the test suite. Even though they are not independent of each other (if one coverage metric is high, others tend to also be high), it is important to evaluate all these metrics to get the full picture. Coverage metrics show which parts of the code are covered, while mutation score assesses whether the tests' assertions are meaningful.

3.3 Research Question Motivation

The following paragraphs motivate the research questions, how I approach them, and the vision behind them.

RQ1: How do LLM-generated tests on bytecode compare to EvoSuite?

Motivation; Test Wars [2] showed that LLMs can achieve higher mutation scores than EvoSuite when given source code. However, source code is not always available for third-party libraries. It remains unknown whether LLMs can replicate this result when given only a bytecode representation.

Approach; To answer RQ1, I feed the LLM only the javap output of the target class and generate tests per method. I then compare the resulting test suites against EvoSuite on their quality using metrics described in the section above (line coverage, branch coverage, method coverage, mutation score).

Expected contribution; If the LLM matches or exceeds EvoSuite on bytecode, this demonstrates that source code is not necessary for effective LLM-based test generation, making the approach applicable to a wider range of libraries and a suitable replacement for EvoSuite.

RQ2: Do example tests improve LLM-generated test quality, and does the type of example (source vs bytecode) matter?

Motivation; LLM-generated tests are known to suffer from hallucination, producing non-existent method calls, causing

compilation errors [3]. When test artifacts are available (e.g., from the library's own test suite), they could serve as examples showing the LLM how to properly call methods, what variables they can use, and copy the setup logic. However, it is unclear whether adding such examples actually helps, and whether source code examples are necessary or if decompiled bytecode examples suffice.

Approach; I extend the RQ1 pipeline by adding test examples to the prompt. Two configurations are tested: (1) bytecode + test (decompiled) bytecode (BTB), and (2) bytecode + test source code (BTS). Both are compared against the bytecode-only (BC) baseline using the same quality metrics. If hallucination is mitigated by this approach, method coverage should increase compared to the BC baseline. The other metrics are used to assess any additional changes.

Expected contribution; If examples improve method coverage, this provides a practical way to mitigate hallucination when source code is available. Overall test quality could also increase if the LLM gains a deeper understanding of the targeted method. If decompiled bytecode examples work as well as source examples, test artifacts alone are sufficient, further reducing the need for source code access.

4 Results

This section compares four experimental approaches: EvoSuite and three LLM-based configurations: bytecode-only (BC), bytecode + test bytecode (BTB) and bytecode + test source code (BTS). The test suites are evaluated on line coverage, branch coverage, method coverage, class coverage, and mutation score. All differences reported in this section are in percentage points (PP) unless otherwise noted.

4.1 RQ1: LLM vs EvoSuite on Bytecode

To answer this question, I compare tests generated by the LLM using only bytecode context against tests generated by EvoSuite. The experiment was run on five libraries. Collections and lang3 are large libraries, each with more than 100 testable classes. For these libraries, the pipeline processed a random subset of 100 classes, of which about 15% were not testable. EvoSuite's coverage results consider the entire library. CSV and fileupload contain 8 and 23 testable classes, respectively. Parameter is a small library with only 2 classes, and the LLM failed to generate any tests, whereas EvoSuite managed to produce some.

The results of the line and method coverage are similar to the branch coverage. Branch coverage is the focus since it is a stronger indicator of test thoroughness. In the table below are the branch coverages obtained for the 5 libraries:

Library	LLM Branch (%)	EvoSuite Branch (%)
collections	76.24	80.96
lang3	81.25	42.23
csv	56.09	6.84
fileupload	71.89	50.22
parameter	0.00	18.75

For lang3, csv, and fileupload, the LLM outperforms EvoSuite on branch coverage by a lot, with differences of 29.8, 49.25, and 21.67 PP, respectively. On the remaining two libraries, the LLM scores 4.72 PP lower on collections and 18.75 PP lower on parameter. The smaller libraries, csv and parameter, introduce more variance, making these differences less reliable than those on collections, fileupload, and lang3.

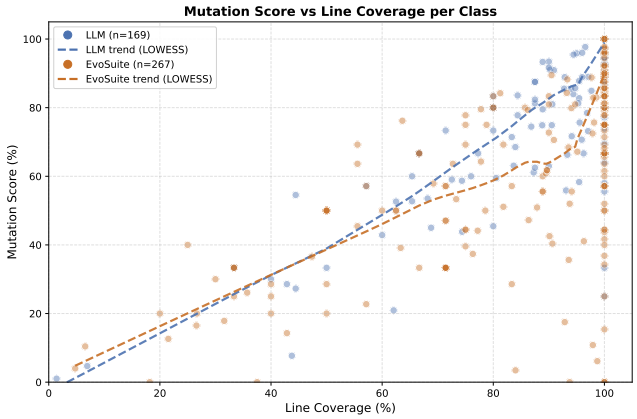

Figure 2: Comparison of the classes EvoSuite and the LLM could generate tests for. The x-axis shows the line coverage, and the y-axis the mutation score. LLM (bytecode only) performs similarly to EvoSuite at lower line coverages, but outperforms EvoSuite on average for classes with higher line coverage.

Line coverage followed the same pattern as branch coverage across all libraries, with no unexpected deviations. Method coverage showed smaller relative differences between EvoSuite and the LLM than branch coverage, but the overall trend remained consistent. The fact that there are no major changes confirms that the coverages are not skewed, and so I can rely on the branch coverages only for making comparisons.

Figure 2 compares mutation scores of the LLM and EvoSuite on the libraries collections, lang3, csv, and fileupload. All of the points in the graph represent a class that had a test suite I managed to run PIT on. Unfortunately, the EvoSuite results represent a subset of the total test classes, and this subset is not random since the average line coverage of the classes I managed to run their tests for was 87.4%, which is substantially higher than the library coverage. Since mutation score and line coverage are not independent of each other, to keep a fair comparison, they are plotted against each other with the line coverage on the x-axis and the mutation score on the y-axis. The dotted trend lines show the average mutation score at each line coverage level. Most classes are centered around the higher end of line coverage. In the bottom right, most classes are from EvoSuite, meaning they have test suites that cover many lines but do not thoroughly test behavior. The trend lines show that for classes with high line coverage, the LLM achieves higher mutation scores than EvoSuite. The table below shows the mutation score statistics for all classes where PIT successfully ran for LLM and EvoSuite test suites. The mean, median, standard deviation, and average line coverage are taken from the results of all individual classes. The aggregate mutation score is the number of killed mutations over all classes divided by the total number of mutations.

Mutation Score	LLM	EvoSuite
Mean (%)	82.0	70.1
Median (%)	90.3	77.8
Standard Deviation (%)	22.4	28.5
Aggregate (%)	72.0	61.8
Average Line Coverage (%)	90.6	87.4

To ensure a fairer comparison, the table below presents the same statistics for the subset of 103 classes for which both EvoSuite and the LLM successfully generated tests, and PIT ran. The average line coverage for the classes was similar

Mutation Score	LLM	EvoSuite
Mean (%)	84.5	69.5
Median (%)	97.7	79.3
Standard Deviation (%)	22.5	28.1
Aggregate (%)	68.8	64.9
Average Line Coverage (%)	91.2	88.0

4.2 RQ2: Impact of Example Tests

This sub-question compares only the LLM approaches: bytecode only (BC) versus bytecode + test bytecode (BTB) and versus bytecode + test source code (BTS). I use the same metrics as before. I compare these on four libraries: lang3, csv, servlet, and fileupload.

An important note: because I only considered classes that have a matching test class for the example test configurations, the number of classes differs from the first RQ. For the experiments that had to include an example test suite, the pipeline processed 60 classes for lang3, 7 classes for csv, 2 for servlet, and 7 for fileupload. The classes I ran the bytecode-only approach for are the same as in RQ 1 for lang3, csv, and fileupload. The results from servlet are on 21 classes and were not compared in RQ1 because I did not find any EvoSuite tests for the library.

The branch coverage results are shown in the table below.

Library	BC (%)	BTB (%)	BTS (%)
lang3	81.25	77.59	78.11
csv	56.09	70.00	71.48
servlet	10.91	0.00	6.45
fileupload	71.89	90.00	84.62

For lang3, the library with the most tested classes, BC outperformed BTB and BTS by about 3.5 PP. For servlet, BTB scored 0.0% branch coverage despite having 10.91% line coverage, meaning the LLM managed to write tests but did not test any branches; the variance for servlet is higher due to BTB and BTS having only 2 test classes. In branch coverage for servlet, BC outperformed BTB by 10.91 PP and BTS by 4.5 PP. For csv, BTB, and BTS outperformed BC by 14 PP and 15.5 PP, respectively. For file upload, BTB scored 18 PP higher than BC on branch coverage, and BTS scored 13 PP higher.

Just like in RQ1, line coverage follows the same pattern as branch coverage across all libraries. Except for servlet, as mentioned earlier, where BTB scored 10.91% line coverage, and BTS scored 9.64%. However, the method coverages do not follow a similar pattern. The table below shows that even though the branch coverage for csv and fileupload was higher on BTB and BTS than BC, the method coverages across these libraries are lower for BTB and BTS than BC.

Library	BC (%)	BTB (%)	BTS (%)
lang3	76.9	79.1	78.2
csv	77.8	70.4	66.7
servlet	19.4	9.7	19.4
fileupload	72.8	57.0	59.6

I ran PIT on 59 classes for the test classes generated by the BTB and BTS approach. The mutation score statistics are shown in the table below, and the BC results are repeated for clearer comparison.

Mutation Score	BC	BTB	BTS
Mean (%)	77.1	76.0	77.0
Median (%)	82.8	83.3	84.2
Standard Deviation (%)	25.5	25.9	25.2
Aggregate (%)	70.9	48.2	66.2
Average Line Coverage (%)	89.5	84.1	88.0

The large difference in aggregate scores results entirely from one class with 232 methods, where BTB performed unusually poorly. When excluding this outlier, BTB has an aggregate score of 70.5% and BTS 67.5%. For comparison, BC's mutation scores on only the classes that BTB and BTS could test were also computed. BC achieved a mean of 77.1%, a median of 82.8%, a standard deviation of 25.5%, and an aggregate mutation score of 70.9% on an average line coverage of 89.5%.

5 Responsible Research

The study involves no human participants or personal data, meaning there are no privacy or consent concerns. However, due to the use of LLMs and public software artifacts, there are still ethical concerns that must be considered.

One limitation of LLM-based systems is that they may generate hallucinated or incorrect outputs. In this experiment, the tests are not deployed, so it is not an issue. However, the goal of the research is to come closer to using LLMs in this context in practice. Therefore, it is important to know the risk of letting an LLM generate code and to run it blindly. Generated code could be harmful, for example, they could delete important files if a file-deleting method is tested. It is safer to run the tests in a controlled sandbox. If harmful tests end up being deployed anyway, it raises the question of who is responsible for the issue, as is often the case in AI-related issues. That is why it is important to never remove the human out of the loop. AI outputs human-readable test [3], so someone should review the tests before they are deployed, making them responsible for any malicious tests.

Another concern is the possibility that some of the evaluated libraries were included in the training data of the LLM. This could provide an unfair advantage and influence the results. However, since this research uses bytecode and a javap representation of it, it is unlikely that the LLM has seen this exact code before. It probably still provides a small advantage because the LLM has seen code with the same behavior. Unfortunately, because the exact training data of most commercial LLMs is not publicly known, this risk cannot be fully mitigated. This limitation is therefore a threat to validity.

The experiment is designed to be reproducible. The pipeline is mostly automated, and the code is reported and

available on GitHub [1] and the results on Zenodo [15]. The dataset consists of open-source Java libraries, and all library versions are reported in the dataset. However, since LLMs are designed to generate different responses to the same input every time, they are nondeterministic and pose a threat to the reproducibility of the research. Therefore, a seed is set to make the output as deterministic as possible, but slight differences might still occur. Finally, the metrics and how they are calculated are explained in the methodology, including the testing frameworks with their versions. This means the entire pipeline is deterministic except for the test generation. For each prompt to the LLM, the output is saved and is available on Zenodo [15]. So there should be no need to reproduce the exact experiment anyway. But it is still important for the experiment to be reproducible, so the experiment can be redone with more time and a better LLM model. Additionally, versions are important in case major bugs are revealed later on, which might pose a threat to the validity of this research.

6 Discussion

6.1 RQ1: LLM vs EvoSuite on Bytecode

Since the line coverage and method coverage are very similar to the branch coverage, I will focus on the branch coverage for the coverage results. The LLM-based approach on bytecode only (BC) achieved substantially higher branch coverage than the EvoSuite tests on 3 libraries. But EvoSuite outperformed the LLM on 2 other libraries. However, out of the libraries that EvoSuite outperformed the LLM, for one library, the difference is quite small, and the other library contains only two classes, so the LLM's failure to generate tests for them could be due to variance. For the other 3 libraries, the LLM outperformed EvoSuite with a big difference, also for lang3, which is a big library. Out of these libraries, it is clear that LLM-generated tests on bytecode can perform much better than EvoSuite tests. In the cases where EvoSuite performed better, the advantage was either marginal or likely due to variance, suggesting that the LLM approach is at least competitive with, and possibly superior to, EvoSuite.

A hypothesis stated earlier in the report, based on the findings of Test Wars [2], was that LLM-generated tests achieve a higher mutation score than EvoSuite, even when line coverage is similar. The results strongly support this hypothesis when considering per-class performance: the LLM-generated tests show a notably higher mean and median mutation score compared to EvoSuite, despite having similar line coverages. It should be noted, however, that this comparison is based only on the subset of classes for which the approaches generated compilable tests and PIT executed successfully. Since line coverage was higher than the library line coverage, this subset is not fully representative of all classes in the subject libraries. Nonetheless, given this subset, the narrative changes dramatically when examining the aggregate mutation scores across all classes. Here, the difference becomes much smaller. After manually inspecting the scores per class, the reason for this result becomes clear: the LLM performs very well on smaller classes, which results in high average and median scores. However, on larger classes with a higher number of mutations, the LLM's performance drops a lot. EvoSuite achieves lower scores on small classes on average, but is more consistent across bigger class sizes. This shows the LLM-generated tests perform very well on average since most classes do not contain many mutations, but does not perform well on big classes with many mutations. EvoSuite has a lower mutation score on average, but is more consistent across different class sizes.

6.2 RQ2: Impact of Example Tests

Across the 4 tested libraries for which I evaluated the branch coverage, BTB and BTS outperformed BC on two libraries with a large margin. However, these libraries contained only a few classes. BC outperformed BTB and BTS on two other libraries, one big library for which BC performs better by a marginal difference, and one very small library that all approaches scored badly on, and could be due to variance. From these results, it is unclear whether example test artifacts actually improve branch coverage. On the only large library, BC achieved the highest branch coverage; however, it could hint that for some libraries, like csv and fileupload, the test examples make a big difference, since for those libraries, BTB and BTS scored much higher on branch coverage.

Another hypothesis stated earlier in the report was that example tests should help improve compilation rates because the LLM knows how to invoke methods, what fields to use, and how to copy the setup from the example tests. However, the method coverage for three of the four libraries seems to be the opposite of the branch coverage results. The libraries where BTB and BTS got more branch coverage had much lower method coverage. And the library where BC scored higher in branch coverage had a marginally lower method coverage, whereas the last library for which all approaches performed badly did not show anything interesting. It is clear from these results that the method coverage does not increase from using test examples, since it either performs similarly to BC or worse.

BTB and BTS performed about equally on branch and method coverage, except for BTB performing slightly better on fileupload. However, this can very likely be due to variance, since the difference is small and the number of classes in the library is small. Comparing the mutation scores, BC performed similarly to BTB and BTS on mean and median. Between BTB and BTS, there are only marginal differences except for the aggregate mutation score. This difference came solely due to a big class with 232 methods, where BTB scored very badly, while BC scored similarly to BTS on mutation score. This may be due to variance or because the decompiled test bytecode confuses the LLM on very large classes. Since the mean and median mutation scores of BC dropped when comparing only the classes that BTB and BTS generated tests for, this suggests that the classes tested by developers are more often complex, and BC's mutation scores were inflated due to simpler classes that developers did not test.

6.3 Implications of the Results and Future Work

The results of RQ1 demonstrate that LLM-based test generation remains effective even when only bytecode is available, which provides less context than source code. It confirms the

suggestion from Test Wars [2] that LLMs outperform EvoSuite on mutation score, but in the case when the LLM is given only bytecode. However, since this paper explores new ground, it is not clear what the best way of representing bytecode is. More advanced techniques to filter out irrelevant information from javap output, or the use of javap and different decompilers to see which one performs best, are left to be explored. Once these techniques are better understood, it would be very interesting to compare bytecode approaches to source code approaches to see whether the lower context impacts test suite quality.

The results of RQ2 show that adding test artifacts, regardless of type, did not consistently increase compilation rates or solve the hallucination issues commonly reported in LLM-based test generation [3]. While these findings show that adding test artifacts does not simply increase the compilation rate, the approaches did achieve better branch coverage on some libraries, despite having lower method coverage. It could be the case that test artifacts only help with more complex libraries, and it could also be that the extra context often only confuses the LLM, a phenomenon called "context distraction" where performance drops when irrelevant information is included [19].

6.4 Threats to Validity

This paper evaluated a limited number of libraries. I acknowledge this limitation and therefore focus on large, consistent differences rather than small variations. The small number of libraries limits the generalizability of the results. The improvements observed on some libraries may not hold for others. However, since the goal is to demonstrate feasibility rather than generalizability, this does not undermine the main finding: LLMs can outperform EvoSuite on bytecode.

Another threat to the mutation score comparison of the LLM-based approach vs EvoSuite is that the only classes I managed to run PIT on did not fully represent the average test suite generated by EvoSuite. I have mitigated this by mapping the line coverage and mutation score together, and the LLM showed better mutation scores. I do not think this is a major threat because the fact that only classes with high coverage could run for EvoSuite should benefit EvoSuite when looking at average mutation scores.

Finally, an issue with the approach used for adding test artifacts is that classes without a corresponding test class were skipped. This means the LLM only wrote tests for classes that the developers also wrote tests for. It could be that the results are skewed by classes that are harder to test; developers may have written tests only for complex classes. Conversely, it could also mean that the classes tested by BTB and BTS are confirmed to be testable because developer-written tests exist. This could lead to an unfair comparison between BC and the test artifact approaches. To mitigate this, I manually checked the results per class. Almost all classes that BTB and BTS managed to test were also handled by BC. On the subset where mutation scores could be compared across all three configurations, BC achieved similar scores to BTB and BTS, suggesting the filtering did not introduce a systematic bias. More advanced techniques to extract information from test artifacts and a qualitative analysis of what classes are being tested would be interesting to explore in future work.

7 Conclusions

Automated test generation for third-party libraries is challenging when source code is unavailable. LLMs have shown promise on source code, but it was unclear whether they could generate effective tests from bytecode alone. While EvoSuite operates on bytecode, its performance leaves room for improvement. Test Wars [2] observed that LLM-generated tests can outperform EvoSuite in terms of mutation score. This paper investigates whether this observation holds when only bytecode is available. A common issue in LLM-based test generation is hallucination, which causes low compilation rates. It also investigates whether adding test artifacts increases compilation rates and overall test quality.

RQ1: How do LLM-generated tests on bytecode compare to EvoSuite? For line and branch coverage, the LLM-based approach on bytecode substantially outperformed EvoSuite on three of five libraries, with improvements ranging from 21% to 49%. On the two libraries where EvoSuite performed better, its advantage was either marginal (4.7%) or on a library with only two classes, making the result unreliable due to variance. This shows the LLM approach performs better on some libraries and is otherwise close to what EvoSuite achieved.

For the mutation score, the results partially support the Test Wars hypothesis. On a per-class basis, the LLM achieved a higher mean and median mutation score than EvoSuite. This comparison is based on a subset of classes with high line coverage, which is not fully representative of all classes in the libraries. However, within this subset, the line coverages of the LLM and EvoSuite were very close, so the comparison is still meaningful. When examining aggregate mutation scores across all classes, the difference was much smaller. Manual inspection revealed that the LLM performs well on small classes with few mutations, inflating its average and median scores, but struggles on large classes with many mutations. EvoSuite is more consistent across class sizes, resulting in similar aggregate mutation scores despite lower per-class averages. This suggests that while LLM-generated tests can outperform EvoSuite on many classes, they are less reliable on complex classes with many mutants.

Overall, these results show that LLM-generated tests on bytecode are a viable alternative to EvoSuite, and can even outperform it on certain libraries.

RQ2: Do example tests improve LLM-generated tests, and does the type of example matter? For RQ2, the results are mixed. Example tests improved branch coverage on two small libraries but did not help on the only large library, suggesting their benefit may depend on library size or complexity. The hypothesis that examples would improve compilation rates was not supported; method coverage either stayed similar or decreased when examples were added. It might be that examples ensure more branch coverage because the LLM understands complex methods better, but in general, the extra context may confuse the LLM. Comparing the two artifact types, BTB and BTS performed almost identically on most metrics, with the only notable difference being an ag-

gregate mutation score outlier caused by one very large class, where BTB performed poorly. Overall, example tests do not consistently improve mutation score; they score similarly to when no test artifacts were added. Finally, the choice between source code and bytecode examples appears to make little difference.

References

[1] J. L. Overmars A. Głodek. Research-Project. https://github.com/Anna51279/Research-Project, 2026. GitHub repository, accessed June 18, 2026.

[2] Azat Abdullin, Pouria Derakhshanfar, and Annibale Panichella. Test wars: A comparative study of sbst, symbolic execution, and llm-based approaches to unit test generation. In *Proceedings of the IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 221–232. IEEE, 2025.

[3] Arda Celik and Qusay Mahmoud. A review of large language models for automated test case generation. *Machine Learning and Knowledge Extraction*, 7:97, 09 2025.

[4] S. Roy Chowdhury, Giriprasad Sridhara, Anirudh K. Raghavan, Joy Bose, Saurabh Mazumdar, Harsh Singh, Senthil B. Sugumaran, and Ricardo Britto. Static program analysis guided llm based unit test generation. In *Proceedings of the 8th International Conference on Data Science and Management of Data (12th ACM IKDD CODS and 30th COMAD)*, pages 279–283. Association for Computing Machinery, 2024.

[5] Gordon Fraser and Andrea Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology*, 24(2):8:1–8:42, 2014.

[6] Gordon Fraser and Andrea Arcuri. Evosuite at the sbst 2016 tool competition. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, SBST '16, pages 33–36, New York, NY, USA, 2016. Association for Computing Machinery.

[7] Tharindu Godage, Sivaraj Nimishan, Shanmuganathan Vasanthapriyan, Vigneshwaran Palanisamy, Charles Joseph, and Selvarajah Thuseethan. Evaluating the effectiveness of large language models in automated unit test generation. In *2025 5th International Conference on Advanced Research in Computing (ICARC)*, pages 1–6, 2025.

[8] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. Evaluating regression test selection opportunities in a very large open-source ecosystem. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*, pages 689–700. ACM, 2018.

[9] JaCoCo Team. Jacoco java code coverage library. https://www.jacoco.org/jacoco/, 2026. Accessed: 2026-06-20.

[10] Kush Jain and Claire Le Goues. Testforge: Feedback-driven, agentic test suite generation. *arXiv preprint arXiv:2503.14713*, 2025.

[11] JetBrains. Fernflower java decompiler, 2026. Accessed: 2026-06-16.

[12] Linux Foundation. A summary of census ii: Open source software application libraries the world depends on, 2022. Accessed: 2026-04-21.

[13] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.

[14] Oracle Corporation. *javap*, 2015. Java Platform, Standard Edition 8 Tools Reference. Accessed: 2026-06-16.

[15] J. L. Overmars. Zip for replication of llm-based test generation from bytecode. https://doi.org/10.5281/zenodo.20772587, 2026.

[16] Cathrine Paulsen and Sebastian Proksch. Marco: Compatible version ranges in maven. In *Proceedings - 2025 IEEE International Conference on Software Maintenance and Evolution, ICSME 2025*, Proceedings - 2025 IEEE International Conference on Software Maintenance and Evolution, ICSME 2025, pages 910–914, United States, 2025. IEEE. 41st IEEE International Conference on Software Maintenance and Evolution, ICSME 2025 ; Conference date: 07-09-2025 Through 12-09-2025.

[17] Mohammed Latif Siddiq, José C. da Silva Santos, Raihan H. Tanvir, Nazmus Ulfat, Fahim Al Rifat, and Vitor Carvalho Lopes. Using large language models to generate junit tests: An empirical study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, pages 313–322, New York, NY, USA, 2024. Association for Computing Machinery.

[18] Yuchen Tang, Zhenyu Liu, Zixiang Zhou, and Xiang Luo. Chatgpt vs sbst: A comparative assessment of unit test suite generation. *CoRR*, abs/2307.00588, 2023.

[19] Nikhil Verma. Active context compression: Autonomous memory management in llm agents. *arXiv preprint arXiv:2601.07190*, 2026.

[20] Sebastian Vogl, Sebastian Schweikl, Gordon Fraser, Andrea Arcuri, José Campos, and Annibale Panichella. Evosuite at the sbst 2021 tool competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 28–29. IEEE, 2021.

[21] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *arXiv preprint arXiv:2307.07221*, 2023.

[22] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen

Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.