LLM-Based Unit Test Generation Without Source Code

An Empirical Evaluation of Bytecode Representations, Prompt Engineering, Model Selection, and Temperature Settings

Anna Glodek

Supervisor(s): 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: Anna Glodek
Final project course: CSE3000 Research Project
Thesis committee: Sebastian Proksch, Cathrine Paulsen, Soham Chakraborty

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Automated unit test generation is often used to reduce the manual effort required to create and maintain software tests. Recently, Large Language Models have shown promising results generating unit tests directly from source code; however, existing work assumes its availability, which is not always the case. Many third-party libraries are distributed only as compiled artifacts, making source-code-based test generation difficult or impossible. Generating tests from compiled software could help developers evaluate the behavioural compatibility of dependency updates without access to the original codebase, but it remains unclear how well LLMs perform when only bytecode is accessible.

This research investigates LLM-based unit test generation using only bytecode. I developed an automated pipeline that generates, compiles, executes, and evaluates JUnit tests for Java libraries from disassembled and decompiled bytecode. I use it to study how model choice, representation, prompting, and temperature affect compilation, execution, and coverage. I also evaluated the best configuration with iterative prompting and compared it against EvoSuite.

Across 50 Java libraries, the best configuration achieved 89.5% compilation and 83.6% execution success. Few-shot prompting and higher temperatures gave the strongest single-pass results, while iterative prompting nearly doubled coverage. Compared with EvoSuite, the approach produced usable tests for all 19 evaluated libraries, while EvoSuite succeeded on 9 but achieved higher coverage where it succeeded. These results suggest that bytecode-based LLM test generation is promising when source code is unavailable.

1 Introduction

Modern software development depends heavily on third-party libraries, with studies estimating that 70–90% of modern systems consist of open-source components [1]. While software reuse accelerates development and reduces costs, it also introduces maintenance challenges: keeping dependencies up-to-date is essential, as newer releases often contain bug fixes, security patches, and performance improvements, but current dependency management tools provide limited guarantees on behavior compatibility. As a result, developers are often responsible for manually validating dependency updates to ensure they do not introduce breaking changes. Studies show that developers remain suspicious of automated tools such as Dependabot, frequently intervening manually and in some cases abandoning them entirely due to recurring breakages [2], highlighting a clear need for more reliable, automated methods to assess the safety of dependency updates.

Testing is a commonly used strategy to decide whether a dependency update is safe. Developers may rely on client-side tests or, when available, library-provided test suites to detect behavioral differences across versions, but this approach is limited by the availability and quality of tests. Client tests frequently fail to cover all relevant functionality, while library tests and even source code are often unavailable, leaving developers uncertain when updating dependencies.

Automated test generation offers a potential solution to this problem. Search-based tools such as EvoSuite can generate tests directly from bytecode, enabling testing even when source code is unavailable [3]. More recently, Large Language Models have demonstrated promising performance in automated unit test generation, with LLM-generated tests shown to achieve results comparable to traditional approaches based on symbolic execution and search-based testing [4]. Existing work, however, focuses primarily on scenarios where source code is available, leaving little empirical evidence on LLM-based test generation when only compiled bytecode is accessible.

Recent advances in Large Language Models suggest they may be well suited to this challenge. Modern LLMs have demonstrated strong capabilities across a range of code-related tasks, including code generation, summarisation, and cross-language synthesis [5]. They can also operate flexibly across varied input representations, making them a promising candidate for settings where only compiled artifacts are available [6]. If effective, LLM-based test generation could provide developers with an automated mechanism to produce tests for third-party libraries without access to source code, thus improving confidence in dependency updates.

To evaluate whether these potential advantages translate into effective test generation from bytecode, this paper investigates the effectiveness of LLM-based unit test generation for Java libraries using bytecode-based program representations. This investigation is guided by the following research questions:

- **RQ1:** How do different language models compare in their ability to generate unit tests from bytecode?
- **RQ2:** Does decompiled bytecode provide advantages over disassembled bytecode as an input representation?
- **RQ3:** How do different prompting strategies influence test quality?
- **RQ4:** How does temperature setting affect test quality?
- **RQ5:** How much can iterative prompting improve overall test coverage, and how does the resulting approach compare to EvoSuite?

To answer these questions, I developed an automated evaluation pipeline to generate, compile, execute, and assess JUnit tests for Java libraries, evaluating test quality using compilation success, execution success, and code coverage metrics across a dataset of Java libraries.

The results show that model selection has a substantial impact on generation effectiveness, with the strongest models, Gemma-4 31B and DeepSeek-V4 Flash, achieving compilation rates around 68% and execution rates around 48%, clearly ahead of the remaining models. Disassembled bytecode proved more reliable than decompiled bytecode, achieving higher compilation and execution rates as well. Few-shot prompting achieved the strongest overall coverage, while a

constrained zero-shot prompt achieved the highest compilation rate. Temperature had only a limited effect within the evaluated range, with 0.4 producing the best coverage. Iterative repair prompting produced the largest single improvement in the study, nearly doubling coverage relative to the non-iterative baseline. The final tested configuration generated usable tests for all 19 libraries attempted, compared to 9 for EvoSuite, though EvoSuite achieved higher coverage on the libraries where it succeeded.

The contributions of this paper are:

- Empirical evidence on the effectiveness of LLM-based unit test generation from bytecode, including a comparison of language models, program representations, prompting strategies, and temperature settings.
- An open automated evaluation pipeline for generating, compiling, executing, and measuring the quality of JUnit tests for Java libraries using bytecode-based inputs.
- A dataset of generated tests and evaluation results to support future research on LLM-based software testing.

The pipeline implementation, generated tests, and full experimental results are publicly available on Zenodo at https://doi.org/10.5281/zenodo.20774237.

The remainder of this paper is structured as follows. Section 2 provides background information and discusses related work on automated test generation, Large Language Models, and program representations. Section 3 describes the methodology used in this study. Section 4 presents and analyzes the experimental results. Section 5 discusses responsible research considerations, including responsibility and reproducibility. Section 6 discusses the findings and their implications for LLM-based test generation. Finally, Section 7 concludes the paper and outlines directions for future work.

2 Background and Related Work

This section provides background on automated unit test generation, Large Language Models, and program representations used in software testing, as well as the influence model configurations may have. It reviews prior work relevant to LLM-based test generation and identifies the research gaps addressed by this study.

2.1 Automated Unit Test Generation

Automated unit test generation has traditionally been addressed using techniques such as random testing, symbolic execution, and search-based software testing. One of the most widely used tools is EvoSuite, which generates tests directly from Java bytecode using evolutionary algorithms to optimize objectives such as code coverage [3]. More recently, Large Language Models have emerged as an alternative approach, with prior studies showing that they can generate compilable and executable unit tests directly from source code and achieve competitive performance on coverage metrics [7; 8]. These results have motivated increasing interest in understanding the factors that influence LLM-based test generation performance.

2.2 Large Language Models for Unit Test Generation

Large Language Models have been applied across a range of software engineering tasks, including code completion, bug fixing, code summarization, and test generation [9]. Models such as ChatGPT can generate compilable and executable unit tests directly from source code, demonstrating the potential of LLMs as an alternative to traditional test-generation approaches [7].

Despite these promising results, the quality of generated tests remains highly variable, with common issues including compilation failures, hallucinated method calls, missing imports, and limited code coverage [7; 10]. Researchers have proposed various techniques to address this: for example, ChatUniTest combines LLM-based test generation with iterative refinement strategies to increase test effectiveness and coverage [10]. Currently, less is known about improvement strategies in settings where only lower-level program representations are accessible.

2.3 Program Representations for Test Generation

Most existing work on LLM-based unit test generation assumes that source code is available as input to the model [7; 10; 11], so relatively little attention has been given to how alternative program representations affect performance.

Recent research suggests LLMs are capable of reasoning about substantially lower-level representations. LLMs evaluated on binary code understanding tasks, including function name recovery and summarization managed to extract meaningful semantic information directly from binary representations [6]. Similarly work on LLM-based decompilation has shown that language models can reconstruct high-level source code from low-level representations [12]. These findings suggest source code may not be strictly necessary for effective software engineering tasks.

However, unlike this prior work, which focuses primarily on reverse engineering, decompilation, and binary analysis, the suitability of lower-level representations specifically for unit test generation remains unclear, with little empirical evidence comparing disassembled bytecode and decompiled source code as inputs for LLM-based test generation.

2.4 Influence of Model Configuration

Model configuration can also significantly influence the quality of LLM-generated tests. Recent empirical studies report substantial variation in performance across language models, with stronger models generally achieving higher compilation success and coverage at the cost of increased computational requirements [8], likely arising from differences in architecture, training data, parameter count, and code-specific fine-tuning.

Generation parameters may also affect test quality. One of the most commonly used parameters is temperature, which controls the randomness of token selection during generation. Lower temperatures typically produce more deterministic outputs, whereas higher temperatures encourage greater output diversity. While the effects of temperature have been

studied in code generation tasks, with prior work finding optimal performance below a temperature of 0.5 [13], comparatively little empirical evidence exists regarding temperature's influence on automated test generation using bytecode. Understanding how model selection and generation parameters affect test quality is therefore important for identifying effective configurations for LLM-based testing systems.

2.5 Research Gap

While prior work has explored model selection, generation parameters, and lower-level program representations across various software engineering tasks [8; 6; 12], their combined impact on bytecode-based unit test generation remains unexplored. To address this gap, this study investigates unit test generation in the absence of source code, comparing disassembled bytecode and decompiled source code as program representations, and evaluating the influence of model selection, prompting strategies, and temperature settings.

3 Methodology

This section describes the experimental design, dataset, model choice, and automated test-generation pipeline used in this study. The objective is to evaluate the effectiveness of Large Language Models for automated Java unit test generation. To do this, I built an automated pipeline that generates JUnit 4 test suites for classes extracted from external Java libraries. The pipeline then evaluates these test suites using compilation success, execution success, and code coverage. By applying the same generation and evaluation process to all experimental conditions, the methodology enables controlled comparisons between language models, program representations, prompting strategies, and temperature settings.

3.1 Experimental Setup

Dataset and Program Representations The dataset used in this study is a subset of 50 Java libraries drawn from the top1000_artifact_availability.csv dataset. This dataset was originally drawn from the 1000 most popular Java libraries on libraries.io, narrowed to 930 after removing libraries not found on Maven Central or lacking version-level popularity data on deps.dev. A representative version was then selected for each based on recency and popularity from deps.dev.

To obtain a manageable benchmark within the available computational budget, the first 50 libraries in the dataset were selected. Because the underlying dataset was curated using popularity-based selection criteria, the resulting subset contains widely used libraries spanning a variety of domains, API designs, and implementation styles.

For each library, the corresponding compiled JAR artifact was downloaded and processed automatically. Classes that could not be meaningfully tested in isolation – interfaces, abstract classes, enumerations, and inner classes – were excluded, since their inclusion could introduce variability unrelated to the research questions and require class-specific prompting strategies. The remaining classes formed the dataset used throughout the evaluation.

Three program representations were evaluated: detailed disassembled bytecode produced using `javap` [14] (`javap -c -p -s`), containing method signatures, bytecode instructions, and private members; a simplified public API view (`javap -public -c`); and decompiled source code produced by the Vineflower decompiler [15]. These representations provide different levels of information about the same program allowing us to investigate whether test generation benefits more from source-like structure, detailed implementation information, or simply access to a class's public interface.

Table 1: Dataset summary after class filtering

Metric	Value
Libraries	50
Total classes in JARs	10,084
Testable classes	4,766
Interfaces filtered	1,612
Abstract classes filtered	686
Enumerations filtered	117
Inner classes filtered	2,903
Average testable classes per library	95.3
Median testable classes per library	46.5

Table 1 summarizes the resulting dataset after filtering. Across the 50 selected libraries, the downloaded JAR files contained 10,084 classes in total, of which 4,766 remained after excluding interfaces, abstract classes, enumerations, and inner classes.

Language Models Five Large Language Models were selected for evaluation: DeepSeek-V4-Flash, Gemma-4-31b-it, Qwen3 8B, DeepSeek-Coder 6.7B, and DeepSeek-Coder-V2 16B, varying considerably in parameter count, release date, and training objective (Table 2). DeepSeek-V4-Flash and Gemma-4-31B-it were selected because they are similarly recent, high-performing models from different model families, enabling an examination of whether comparable benchmark performance translates to similar effectiveness in bytecode-based unit test generation. They were also included to compare against the smaller, older models in the selection, in order to assess how much improvement more recent and capable models offer for this task.

Qwen3 8B was chosen as a compact open-source model that runs efficiently on consumer hardware via Ollama and has previously shown strong coding capabilities despite its size. DeepSeek-Coder 6.7B and DeepSeek-Coder-V2 16B were included as code-specialized models, allowing the effect of scale and architectural improvements to be examined within a single model family.

Together, these models vary in size, specialization, and deployment dates, enabling a broader assessment of the factors influencing bytecode-based unit test generation performance.

Table 2: Evaluated language models.

Model	Parameters	Release Date	Type
DeepSeek-Coder 6.7B	6.7B	Nov. 2023	Code-specialized
DeepSeek-Coder-V2 16B	16B	Jun. 2024	Code-specialized
Qwen3 8B	8B	Apr. 2025	General-purpose
DeepSeek-V4-Flash	284B (13B active)	Apr. 2026	General-purpose
Gemma 4 31B	31B	Apr. 2026	General-purpose

Experimental Variables The independent variables are program representation, language model, prompting strategy

and sampling temperature. The dependent variables are compilation success rate, execution success rate, test coverage, token consumption, and generation throughput. Compilation and execution success measure test correctness, coverage quantifies how much of the target code is exercised, and token consumption and throughput indicate the efficiency of different models and configurations.

3.2 Test Generation Pipeline

The pipeline automates the full process of generating and evaluating unit tests for external Java libraries. It identifies testable classes in a Maven artifact, constructs prompts, queries a language model, compiles and executes the generated tests, and records coverage and performance metrics. I implemented all stages in Python and allowed easy configurations through a single file, enabling reproducible experiments across models, prompting strategies, and representations.

Class Extraction and Representation The pipeline first downloads the compiled JAR for each library from Maven Central, caching it locally for reuse across runs. It then enumerates all `.class` files in the JAR, excluding `META-INF` entries, and inspects each remaining class with `javap`. It classifies interfaces, abstract classes, enumerations, etc. The number of classes processed per library is capped to limit computational cost and keep the evaluation from being dominated by a small number of large libraries.

Each testable class can be represented in one of three ways: `javap` produces either a full disassembly or a condensed variant with only public constructors and method signatures, while Vineflower reconstructs Java source code from the compiled class. This allows the same class to be represented as disassembled or decompiled bytecode while keeping every other aspect of the experiment unchanged.

Prompt Construction and Test Generation Prompts are built from configurable template files, allowing prompting strategies to be swapped without changing the pipeline itself. Five templates were used: zero-shot, constrained zero-shot, few-shot, simplified zero-shot, and iterative reprompting, each instructing the model to generate a self-contained JUnit 4 test class under a fixed set of formatting and compilation constraints.

The pipeline submits the constructed prompt to the selected model, with model and temperature varied between runs and a fixed random seed (15) used across all experiments to support reproducibility, then cleans the output by stripping markdown formatting and non-ASCII characters, trimming to the first valid import or class declaration, and ensuring an import for the target class is present.

Test Compilation and Execution The pipeline places each generated test into a Maven [16] project declaring dependencies on the target library and JUnit 4 [17], with Surefire and JaCoCo [18] instrumentation enabled. It compiles tests with `mvn test-compile` and, on success, executes them with `mvn test`, applying timeouts to both steps and logging compilation failures and execution failures.

Coverage and Results Recording For successfully executed tests, the pipeline parses JaCoCo's XML report for line and branch coverage per class and library, and combines successful tests into aggregate library-level suites to measure overall coverage. It records compilation success, execution success, coverage, token usage, generation time, and error logs at both the class and library level.

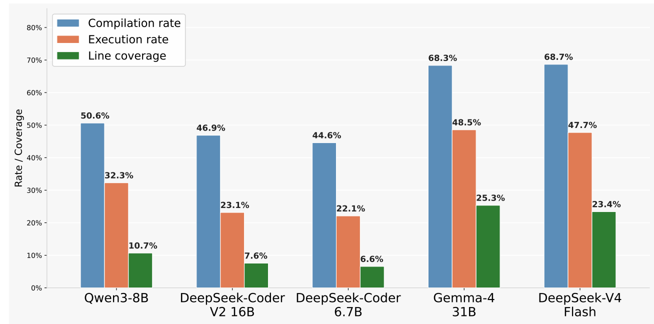

Figure 1: Compilation rate, execution rate, and line coverage by model.

4 Results

This section presents the results of the empirical evaluation. Results are organized by research question and, unless otherwise stated, aggregated across all successfully processed classes from the 50 selected libraries. To keep evaluation cost manageable, a maximum of 100 classes were sampled per library.

4.1 RQ1: How do different language models compare in their ability to generate unit tests from bytecode?

Before investigating the effect of prompting strategies, program representations, and temperature settings, it is important to understand how the selected models compare under a consistent baseline configuration. A model that struggles to produce valid tests in the first place would make it difficult to draw meaningful conclusions from later experiments. All five models were therefore evaluated on the same 2,445 classes using a consistent zero-shot prompt and temperature 0.

Figure 1 shows the compilation rate, execution rate, and line coverage achieved on the evaluated classes. After the per-library cap and the filtering described in Section 3, 2,445 classes remained for evaluation. Compilation and execution rate measure the percentage of these classes for which the generated test successfully compiled and ran to completion without errors or assertion failures. Coverage was measured at the library level: all successfully executing test classes for a library were aggregated into a single test suite, and line coverage was measured with JaCoCo.

Two clear tiers emerge. Gemma-4 31B and DeepSeek-V4 Flash achieved compilation rates of 68.3% and 68.7% and execution rates of 48.5% and 47.7% respectively, substantially outperforming the remaining models, with Gemma-4 31B achieving higher line coverage (25.3% vs 23.4%). Qwen3-8B formed a clear middle tier, while both DeepSeek-Coder variants achieved considerably weaker results across all metrics.

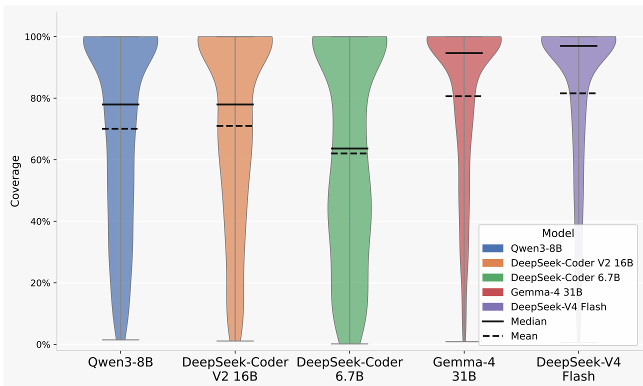

Figure 2: Distribution of per-class line coverage across models, restricted to test classes that compiled and executed successfully. The horizontal bar indicates the median.

The coverage distribution in Figure 2 reveals an important nuance: when tests did compile and execute successfully, Gemma-4 31B and DeepSeek-V4 Flash achieved consistently high per-class coverage, with medians above 90% and narrow distributions near the top of the range. However, the mean for both models lags noticeably behind the median (Gemma-4 31B: 92.3% median vs. 76.3% mean; DeepSeek-V4 Flash: 95.4% median vs. 77.1% mean), indicating that a subset of tests that compile and execute successfully still achieve little to no coverage. This suggests that for the top models, two challenges remain: generating tests that compile and execute successfully in the first place, and avoiding this smaller set of low-coverage outliers once they do. In contrast, weaker models show broader distributions with lower medians overall – particularly DeepSeek-Coder 6.7B at around 63%.

Qwen3-8B outperformed the larger DeepSeek-Coder-V2 16B across all metrics, suggesting architecture and training methodology matter more than parameter count for this task.

Token usage and generation throughput are reported in Table 3. Token usage showed no clear relationship with test quality. Qwen3-8B generated the most output tokens yet achieved substantially lower coverage than the top-performing models.

Table 3: Token usage and generation throughput by model.

Model	Prompt Tokens	Response Tokens	Total Tokens	Tokens/s
DeepSeek-V4 Flash	7.53M	3.07M	10.60M	65.9
Gemma-4 31B	7.66M	2.67M	10.33M	74.9
Qwen3-8B	6.94M	4.82M	11.76M	41.0
DeepSeek-Coder-V2 16B	7.71M	1.03M	8.74M	83.8
DeepSeek-Coder 6.7B	8.06M	0.93M	8.99M	44.6

In answer to RQ1, model selection has a substantial impact on bytecode-based test generation effectiveness. The two strongest models, Gemma-4 31B and DeepSeek-V4 Flash, are used as the basis for subsequent experiments, as establishing a strong baseline is essential for isolating the effects of representation, prompting strategy, and temperature investigated in the following sections.

4.2 RQ2: Does decompiled bytecode provide advantages over disassembled bytecode as an input representation?

Knowing which input representation works best has direct practical implications, since reliance on decompiled source-like code would limit the approach to libraries where decompilation succeeds. Three representations were compared: detailed disassembled bytecode (`javap -c -p -s`), a simplified public API view (`javap -public -c`), and decompiled source code produced by Vineflower, using DeepSeek-V4 Flash at temperature 0. The 415 classes for which Vineflower failed to produce a decompilation were excluded from the source code condition, so the comparison reflects only classes decompilable by Vineflower. As shown in Figure 3, bytecode and source code consistently achieved very simmilar compilation and execution rates. Coverage presents a more nuanced picture: decompiled source code achieved a higher 25.3% line coverage against 23.4% for disassembled bytecode, while full bytecode nearly matched it at 25.4%, suggesting richer bytecode detail partially compensates for the lack of source-like structure. Under the simple prompt configuration, source code again outperformed bytecode in coverage (20.8% vs 18.0%), though both performed worse than their zero-shot counterparts. Taken together, these results present a mixed answer to RQ2: disassembled bytecode is more reliable at producing valid tests, while the coverage differences for the two representations are small enough to fall within natural variance – making bytecode a viable, and more dependable, alternative.

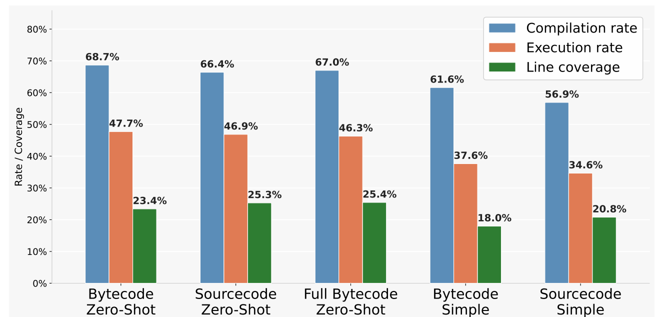

Figure 3: Compilation rate, execution rate, and line coverage by input representation and prompt style.

4.3 RQ3: How do different prompting strategies influence test compilation, execution, and coverage?

The prompt is one of the cheapest variables to tune, requiring no changes to the model or input representation. Four strategies were evaluated: a standard zero-shot prompt, a simplified zero-shot prompt with fewer constraints, a constrained zero-shot prompt that explicitly lists available public methods and forbids calling anything outside that list, and a few-shot prompt. The example provided to the few shot prompt consisted of a simple class's bytecode representation and a corresponding test suite. As shown in Figure 4, few-shot achieved the strongest overall results with a 54.3% execution rate and 26.0% line coverage, while constrained zero-shot achieved the second highest compilation rate at 70.7%, suggesting that explicitly restricting the model to known methods reduces hallucinations. Simple zero-shot performed worst on both metrics despite receiving the same bytecode input as standard zero-shot, indicating that prompt structure matters even when the underlying information is identical. One pattern worth

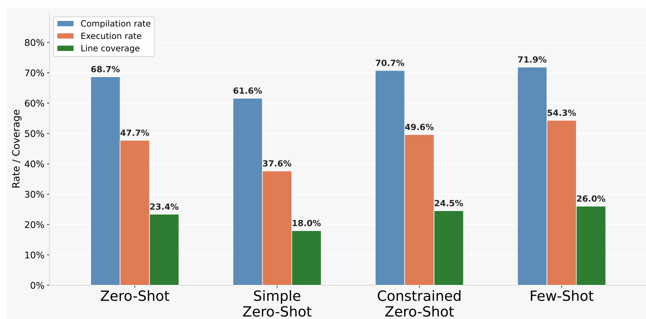

Figure 4: Compilation rate, execution rate, and line coverage by prompt strategy.

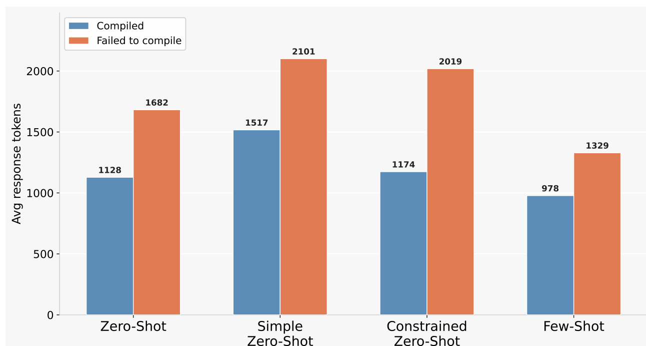

Figure 5: Average response tokens for compiled vs failed tests.

noticing, as shown in Figure 5, failed tests consistently required more response tokens than compiled ones across all strategies. Same thing can be observed on the input side: classes for which generation failed had larger prompt sizes on average across all four strategies, indicating that larger target classes are inherently harder to generate valid tests for. Finally, manual inspection of per-library coverage rates showed that libraries difficult to test under zero-shot remained difficult across all prompt strategies, suggesting prompt design cannot fully compensate for fundamentally complex libraries. In answer to RQ3, few-shot prompting produced the strongest overall results, while constrained zero-shot achieved a comparatively high compilation rate. Simple zero-shot performed worst overall, indicating the advantages of providing concrete constraints to the model.

4.4 RQ4: How does temperature setting affect test compilation, execution, and coverage?

Temperature controls the randomness of model outputs and is commonly tuned to balance diversity and coherence. Six values were evaluated ranging from 0.0 to 1.0. Neither compilation nor execution rates vary substantially across the range, with compilation staying between 67.7% and 70.1% and execution between 46.9% and 49.0%. Line coverage peaks at temperature 0.4 (25.5%) and declines at higher values, dropping to 21.3% at 1.0, suggesting that increased randomness begins to hurt coherence without meaningfully improving diversity. A related pattern appears in branch coverage, which follows a similar trend across the same six temperatures: 13.9% at 0.0, 13.6% at 0.2, rising to a peak of 15.8% at 0.4, then declining to 14.3% at 0.6, 13.7% at 0.8, and 14.2% at 1.0.

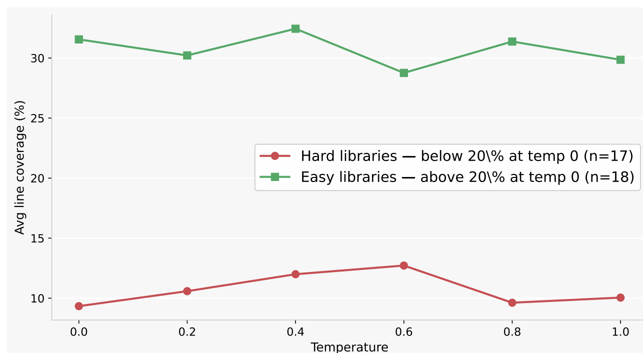

Figure 6: Average line coverage across temperatures for libraries below 20% coverage at temperature 0 and those above 20%.

The average across all libraries hides what is actually happening at the library level. As shown in Figure 6, libraries that achieved below 20% coverage at temperature 0.0 show a consistent improvement at moderate temperatures, peaking at 12.0% at temperature 0.6 compared to 9.3% at 0.0. Libraries already above 20% show no such benefit and decline slightly at higher temperatures, suggesting that the optimal temperature is library-dependent rather than globally fixed.

In answer to RQ4, temperature has limited impact on test generation effectiveness when averaged across all libraries. At the library level, however, temperature matters more than the aggregate numbers suggest. This points toward temperature as a setting worth tuning selectively, for hard cases, rather than a single value to fix globally.

4.5 RQ5: Can iterative prompting improve test coverage?

The preceding experiments generate each test once and accept the result regardless of outcome. This section evaluates iterative prompting: classes that fail to compile or execute are re-prompted with the compiler or runtime error, giving the model up to two further attempts to fix the issue.

Effect of Iterative Repair

Two scenarios were compared under identical conditions (DeepSeek-V4 Flash, zero-shot, temperature 0, disassembled bytecode): one without repair prompting and one with up to two repairs. Of the 2,445 evaluated classes, 51.7% required at least one repair attempt, and 1,008 of these 1,264 classes (79.7%) ultimately compiled. As shown in Figure 7, this produced the largest single improvement in the study: compilation rose 20.8 points (68.7% to 89.5%), execution rose 35.9 points (47.7% to 83.6%), and line coverage nearly doubled (23.4% to 39.2%). This suggests that much of the failure rate seen in RQ1–RQ4 reflects recoverable mistakes, like missing imports, wrong signatures, invalid assertions, rather than a fundamental inability to understand the target class.

Comparison to EvoSuite

The best-performing configuration (few-shot, Gemma-4 31B, temperature 0.4) was evaluated against EvoSuite on a subset of 20 libraries drawn from the original 50, with the class cap and filtering removed. One, `listenablefuture`, is an empty placeholder artifact with no classes, leaving 19. The full list of libraries used in this comparison is available in

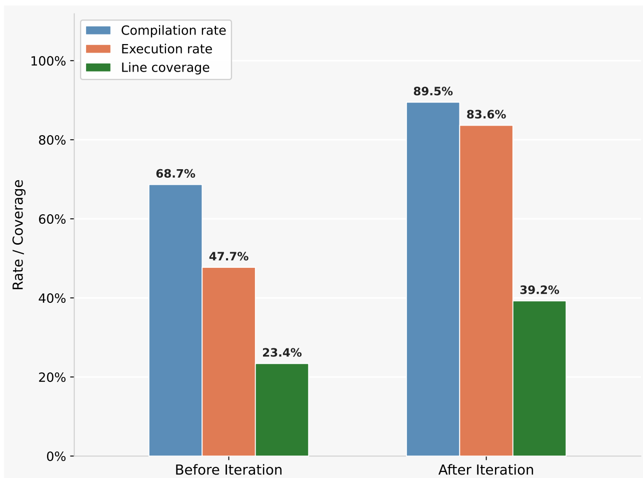

Figure 7: Compilation rate, execution rate, and line coverage before and after iterative repair prompting, aggregated across 50 libraries.

the accompanying Zenodo repository. The LLM produced at least one running test for all 19, while EvoSuite produced none at all for 10 of them.

Coverage was compared on these 9 shared libraries, the only set where both tools have a result. EvoSuite achieved a higher mean line coverage (60.9% vs 52.7%, median 51.5% vs 47.0%). On the 10 libraries where EvoSuite produced no usable output at all, the LLM still achieved a mean coverage of 43.8%. The 8.9-point gap between the matched and unmatched subsets suggests the libraries EvoSuite could not handle were also somewhat harder for the LLM, though it still produced a usable test suite for every one of them.

Overall, EvoSuite achieves higher coverage where it succeeds, but fails outright on over half the libraries tested. The LLM-based approach trades some peak coverage for succeeding on every library attempted. This can be an advantage when any usable test suite is preferable to none.

5 Responsible Research

Responsibility This study evaluates LLM-based unit test generation for open-source Java libraries. The evaluated models were deployed using a combination of local execution and external API access. No personal data, proprietary datasets, or confidential software artifacts were used during the experiments. The evaluated programs consisted exclusively of publicly available open-source Java libraries.

A potential risk of LLM-based test generation is overreliance on generated outputs. Developers may assume that generated tests are correct or complete without performing additional validation. This work therefore evaluates generated tests using objective metrics such as compilation success, execution success, and code coverage, and does not claim semantic correctness beyond what these metrics capture.

Another responsibility concern relates to the execution of generated code. The generated test suites were compiled and executed directly on the host machine without sandboxing or static analysis. Since LLM outputs cannot be fully predicted, this introduces risks including destructive file system operations, unintended network access, or excessive resource consumption. In the context of this study, these risks were considered acceptable because all experiments were conducted in a controlled local environment using non-sensitive data. However, a production deployment of such a pipeline should execute generated tests inside an isolated environment, such as a container with restricted file system permissions.

Reproducibility To support reproducibility, all experiments were performed using the same evaluation pipeline. Furthermore, all prompts, generated tests, and evaluation results were stored on disk. This makes it possible to inspect the exact inputs provided to the language models and the corresponding outputs produced during each experiment. Finally, all configuration settings, including the random seed, temperature, prompting strategy, and model identifier, were recorded alongside the experimental results. Although externally hosted models may change over time and exact model revisions are not always publicly available, documenting model identifiers improves transparency and facilitates future replication efforts.

6 Discussion

This section interprets the results in light of model capability, input representation, and their implications for bytecode-based test generation.

6.1 Model Selection

The results demonstrate that model selection is one of the most influential factors affecting the effectiveness of bytecode-based unit test generation. However, the differences between models cannot be explained by parameter count alone. Qwen3-8B consistently outperformed DeepSeek-Coder-V2 16B despite having fewer parameters, indicating that advances in model architecture, instruction tuning, and training methodology have a greater influence on test generation performance than model scale alone[19; 20].

When the strongest models produce executable tests, coverage of the target class is typically high. This suggests that the main bottleneck is test validity rather than test quality, and that improving compilation and execution rates may yield larger gains than further optimising coverage. This can be done by iterative prompting for example.

The token analysis adds a further nuance. Qwen3-8B generated nearly five million response tokens, almost double that of the top models, yet achieved substantially lower coverage. Generating more output does not translate to better tests in this setting.

6.2 Input Representation

The results suggest that providing richer bytecode detail matters. The simplified public API view (`javap -public -c`) only provides information about public members, while the full disassembly (`javap -c -p -s`) additionally includes private members, field descriptors, and bytecode instructions. The full representation rescued 37.5% of classes that the simplified view failed on, suggesting that the model uses the additional context to reason about how to instantiate and call

the target class. This is consistent with findings from source-based test generation showing that providing richer contextual information about the target class can improve the quality of generated tests [10].

When comparing full bytecode against decompiled source code, the differences largely disappear. One advantage of decompiled source code is that it typically requires fewer input tokens than full bytecode disassembly, reducing inference costs and allowing larger classes to fit within a model's context window. Decompilers such as Vineflower can fail on obfuscated code, compiler-specific constructs, or classes generated by frameworks at runtime, situations that are common in real third-party libraries [21]. Disassembly with `javap` is more robust in these cases since it operates directly on the class file format without attempting to reconstruct source-level abstractions. In practice, this makes full bytecode disassembly the more practical default when working with compiled artifacts where source code is unavailable.

6.3 Prompting Strategy

Few-shot prompting achieves the highest execution rate and coverage, consistent with recent work showing that LLMs can generate high-quality unit tests via few-shot prompting, with human-written examples producing the best coverage and correctness [22]. Constrained zero-shot prompt achieves one of highest compilation rate, suggesting that explicitly restricting the model to known public methods directly addresses one of the most common failure modes - the generation of tests that call non-existent methods or constructors, a well-documented form of LLM hallucination in code generation that can apparently be partially mitigated through prompt-level constraints without a worked example [23].

Despite receiving the same bytecode input, the simple zero-shot prompt performed substantially worse, indicating that prompt structure influences model behaviour even when the underlying information is unchanged. Less guidance on how to handle ambiguous cases, can lead to more frequent failures.

The finding that failed tests consistently used more response tokens than compiled ones across all strategies suggests that longer generated test suites introduce more opportunities for errors to appear: a single malformed method call or incorrect import is enough to prevent the entire class from compiling, so when one method's test is incorrect, the whole suite is discarded. Future work could explore generating tests method by method, allowing individual failures to be isolated without invalidating the rest of the suite.

6.4 Temperature

While aggregate metrics suggest temperature has limited impact, individual libraries vary drastically across temperatures, with some libraries nearly tripling their coverage at optimal settings. consistent with recent work suggesting that the effect of temperature is highly task and input-dependent rather than globally monotonic [24]. This suggests a possible adaptive strategy: applying higher temperatures specifically to classes where initial generation attempts fail, allowing the model to explore more diverse test structures, which could be implemented as a simple temperature retry mechanism without requiring compiler feedback.

A related pattern appears in branch coverage, which peaks at temperature 0.4 (15.8% vs 13.9% at 0.0). Branch coverage requires tests to exercise conditional logic, which demands more varied input combinations than simple line coverage, so a moderate increase in temperature may help the model explore a slightly wider range of test inputs without introducing enough randomness to break compilation. Above 0.4 this benefit disappears, consistent with the general observation that higher temperatures increase diversity at the cost of coherence in code generation tasks [13].

6.5 Iterative Repair and EvoSuite Comparison

The iterative repair results show that the bottleneck identified in RQ1 is largely recoverable, not fundamental. By adding at most two re-prompts it is possible to almost double the achieved coverage. A failed compilation does not necessarily indicate a misunderstanding of the target class, as 79.7% of initially failing classes were repaired after being shown the error message. This suggests the most effective way of improving bytecode-based test generation may be a second look at the model's own output, rather than a better model, prompt, or representation.

The comparison with EvoSuite points to a different kind of tradeoff. The two tools tend to fail in different ways: the LLM tends to degrade gracefully, often producing a partially useful test even for a hard class, whereas EvoSuite fails outright on over half the libraries tested. For dependency maintenance, a tool that succeeds unpredictably is arguably less useful than one that gives lower but reliable coverage. That said, on the 9 libraries where both tools succeeded, EvoSuite's exhaustive search found meaningfully more coverage, which suggests the two approaches may complement each other rather than directly compete: EvoSuite does well once a class can be analyzed, while the LLM still produces something useful even when it can't.

6.6 Threats to Validity and Limitations

The primary threat to internal validity arises from the sequential design of the experiments. The study first identifies the best-performing model before evaluating program representations, prompting strategies, temperature settings, and iterative prompting; while this reduces computational cost, it does not capture all possible interactions between these factors. The experimental pipeline also relies on external tools and services, including `javap`, Vineflower, JaCoCo, Maven, and externally hosted LLM APIs, whose limitations or behavioural changes may influence the observed results. To mitigate this risk, failures occurring during generation, compilation, execution, and coverage measurement were logged and included in the analysis.

The main threat to external validity concerns generalizability. The experiments were conducted exclusively on Java libraries obtained from Maven repositories and evaluated only five language models, with selection constrained by practical considerations such as computational requirements, API availability, and cost; the findings should therefore not be interpreted as representative of all software ecosystems or all

current and future language models. Furthermore, only disassembled bytecode and decompiled source code were considered as program representations – alternative representations may yield different results.

Construct validity is limited by the metrics used to evaluate test quality. Compilation success, execution success, and test coverage are widely used measures of test-generation performance, but high coverage does not necessarily imply strong fault-detection capability or semantically correct tests, so the reported results are best interpreted as indicators of test effectiveness rather than comprehensive measures of software quality. Generated test suites were also evaluated as complete units rather than at the individual test-method level, so a single compilation or runtime error could cause an otherwise partially valid suite to be classified as failed, potentially underestimating the usefulness of some generated tests.

7 Conclusion and Future Work

Third-party dependencies are often shipped without source code or usable test suites, leaving developers with little automated support for verifying that an update has not broken existing behaviour. This thesis investigated whether LLMs can generate unit tests directly from compiled bytecode to close that gap, using a pipeline that generated, compiled, executed, and evaluated JUnit tests across 50 Java libraries.

Across the five research questions, a few patterns stand out. Model choice matters substantially, but the gap between models is driven by architecture and instruction tuning rather than parameter count alone. Decompilation turns out not to be a prerequisite for effective generation: disassembled bytecode performs comparably to decompiled source code once failed decompilations are accounted for. Few-shot prompting maximized both coverage and execution rates, whereas the inclusion of explicit constraints led to higher compilation success rates. Temperature has little effect in aggregate, but it meaningfully helps specific hard libraries, suggesting any benefit is local rather than global. The clearest win comes from iterative repair prompting, which nearly doubles coverage by fixing recoverable mistakes rather than addressing fundamental misunderstandings. The resulting approach succeeds on every library attempted, compared to under half for EvoSuite, though EvoSuite reaches higher coverage where it does succeed.

Roughly half of all generation attempts fail to compile or execute, and a single failing test method is enough to discard an otherwise useful suite, since test suites are currently evaluated as one unit. This motivates the clearest direction for future work: isolating and retaining individual passing tests, rather than discarding the whole suite over one failure, could directly raise the effective success rates reported here. Second, since temperature gains in RQ4 were concentrated in already-weak libraries, applying a higher temperature selectively as a retry mechanism for failed classes is a natural next step. Finally, combining LLM-based generation with the systematic input exploration of search-based tools such as EvoSuite may help address the compilation and execution failures that persist across every configuration tested in this study.

A Use of Artificial Intelligence Tools

This project investigates the use of Large Language Models (LLMs) for automated unit test generation. As such, LLMs were used to generate the unit tests evaluated throughout this study.

ChatGPT was used to assist with the writing and editing of this thesis. Additionally, LLMs were consulted during the development of the experimental pipeline to help investigate platform-specific issues encountered when running the software on different operating systems, particularly differences between macOS and Windows environments, and to help generate documentation for parts of the codebase prior to publishing it alongside this thesis. AI tools were additionally used to assist in generating example plotting and data-visualization scripts used to produce the figures in this thesis; the underlying data, the choice of what to plot and how to interpret it, and all reported numerical results were independently verified by the author. All generated content and suggestions, in writing, code, and visualization, were reviewed and validated before use.

The author takes full responsibility for the design of the experiments, implementation of the pipeline, interpretation of the results, and the conclusions presented in this report.

References

[1] H. Carter, C. Delia, T. Ellison, C. Eberhardt, S. Hendrick, and P. Holleran, "The 2022 state of open source in financial services," The Linux Foundation, Tech. Rep., 2022.

[2] R. He, H. He, Y. Zhang, and M. Zhou, "Automating dependency updates in practice: An exploratory study on github dependabot," *IEEE Transactions on Software Engineering*, vol. 49, no. 8, pp. 4004–4022, 2023.

[3] 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. Association for Computing Machinery, 2011, pp. 416–419.

[4] A. Abdullin, P. Derakhshanfar, and A. Panichella, "Test wars: A comparative study of sbst, symbolic execution, and llm-based approaches to unit test generation," in *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2025, pp. 221–232.

[5] T.-T. Nguyen, T. T. Vu, H. D. Vo, and S. Nguyen, "An empirical study on capability of large language models in understanding code semantics," *arXiv preprint arXiv:2407.03611*, 2024.

[6] X. Shang, Z. Fu, S. Cheng, G. Chen, G. Li, L. Hu, W. Zhang, and N. Yu, "An empirical study on the effectiveness of Large Language Models for binary code understanding," *arXiv preprint arXiv:2504.21803*, 2025.

[7] 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," *arXiv preprint arXiv:2305.04207*, 2023.

[8] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *arXiv preprint arXiv:2305.12107*, 2024.

[9] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," *arXiv preprint arXiv:2310.03533*, 2023.

[10] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "ChatUniTest: A framework for LLM-based test generation," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 358–362.

[11] 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 2024 International Conference on Software Engineering Workshops*, ser. ICSE-W '24. New York, NY, USA: Association for Computing Machinery, 2024.

[12] H. Tan, Q. Luo, J. Li, and Y. Zhang, "LLM4Decompile: Decompiling binary code with large language models," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2024, pp. 3585–3600.

[13] C. Arora, A. I. Sayeed, S. Licorish, F. Wang, and C. Treude, "Optimizing large language model hyperparameters for code generation," *arXiv preprint arXiv:2408.10577*, 2024.

[14] Oracle. The javap Command. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/specs/man/javap.html

[15] Vineflower Team. Vineflower. [Online]. Available: https://github.com/Vineflower/vineflower

[16] Apache Software Foundation. Apache Maven. [Online]. Available: https://maven.apache.org/

[17] K. Beck and E. Gamma. JUnit 4. [Online]. Available: https://junit.org/junit4/

[18] EclEmma team. EclEmma - JaCoCo Java Code Coverage Library. [Online]. Available: https://www.jacoco.org/jacoco/

[19] Qwen Team, "Qwen3 Technical Report," *arXiv preprint arXiv:2505.09388*, 2025. [Online]. Available: https://arxiv.org/abs/2505.09388

[20] D. Qi, Y. Liang, Y. Zhao *et al.*, "Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence," *arXiv preprint arXiv:2406.11931*, 2024. [Online]. Available: https://arxiv.org/abs/2406.11931

[21] N. Harrand, C. Soto-Valero, M. Monperrus, and B. Baudry, "The strengths and behavioral quirks of java bytecode decompilers," in *Proceedings of the 19th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2019, pp. 92–102.

[22] A. Chudic and G. Çalıklı, "Automated test suite enhancement using large language models with few-shot prompting," *arXiv preprint arXiv:2602.12256*, 2026.

[23] A. Eghbali and M. Pradel, "De-hallucinator: Mitigating llm hallucinations in code generation tasks via iterative grounding," *arXiv preprint arXiv:2401.01701*, 2024.

[24] L. Li, L. Sleem *et al.*, "Exploring the impact of temperature on large language models: Hot or cold?" *Procedia Computer Science*, 2025.