TUDelft
Faculty of Electrical Engineering, Mathematics and Computer Science

Anna Glodek
Supervisors: Sebastian Proksch, Cathrine Paulsen
EEMCS, Delft University of Technology, The Netherlands

LLM-Based Unit Test Generation Without Source Code

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

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, it also introduces maintenance challenges. Dependency management tools provide limited guarantees on behavioral compatibility, so developers are often responsible for manually validating that an update has not introduced breaking changes.

Testing is a common strategy for deciding whether a dependency update is safe, but client tests rarely cover all functionality, and library tests, or even source code, are often unavailable. Automated test generation offers a potential solution: search-based tools such as EvoSuite can generate tests directly from bytecode, and Large Language Models have recently shown strong results generating tests from source code. **However, almost no empirical evidence exists on how well LLMs perform when only compiled bytecode is accessible.**

This thesis investigates LLM-based unit test generation for Java libraries using only bytecode-derived program representations, guided by five 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?

Background

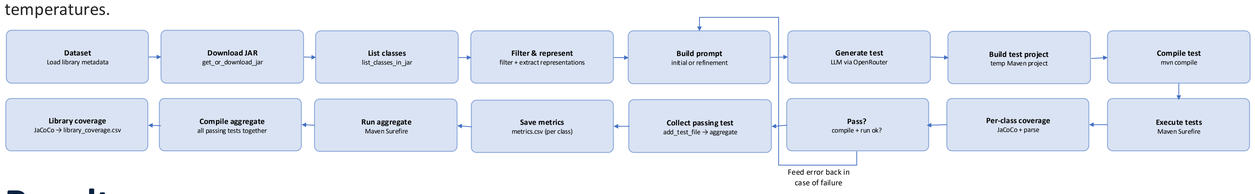
Automated unit test generation has traditionally been addressed using search-based techniques such as EvoSuite, which generates tests directly from Java bytecode [2]. More recently, Large Language Models (LLMs) have demonstrated strong performance in software engineering tasks, including automated unit test generation from source code [4,5]. Existing work, however, largely assumes source code availability. At the same time, recent studies suggest that LLMs can reason about lower-level program representations, including bytecode and binary code [3]. This raises the question of whether LLMs can effectively generate unit tests when only compiled artifacts are available.

Dataset

50 Java libraries were drawn from the top-1000 most popular artifacts on libraries.io. Across these libraries, the downloaded JAR files contained **10,084 classes** in total, of which **4,766 testable classes** remained after excluding interfaces (1,612), abstract classes (686), enumerations (117), and inner classes (2,903). On average there were 95.3 testable classes per library.

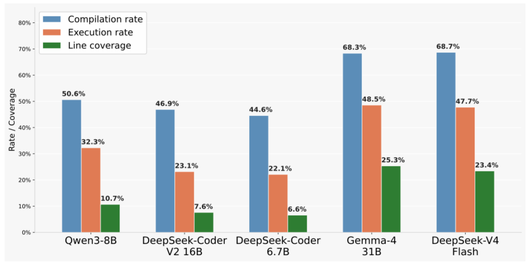
Methodology

An automated pipeline generates, compiles, executes, and evaluates JUnit 4 test suites for classes extracted from external Java libraries. For each library, the compiled JAR is downloaded from Maven Central; testable classes are identified and represented as disassembled bytecode (javap), a simplified public-API view, or decompiled source (Vineflower). A prompt is constructed and sent to the model and the resulting test is compiled with Maven, executed with Surefire, and measured with JaCoCo. All stages are configurable from a single file, enabling controlled, reproducible comparisons across models, representations, prompting strategies, and temperatures.

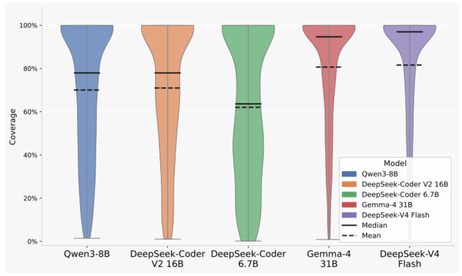


Results

RQ1 – Model Comparison

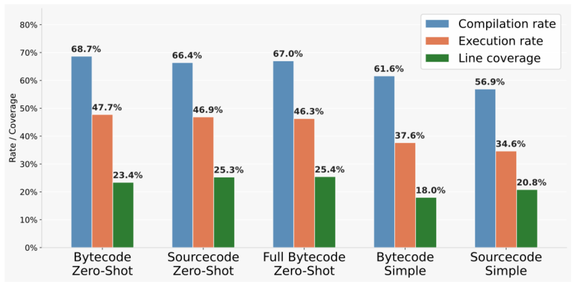


Compilation rate, execution rate, and line coverage by model.



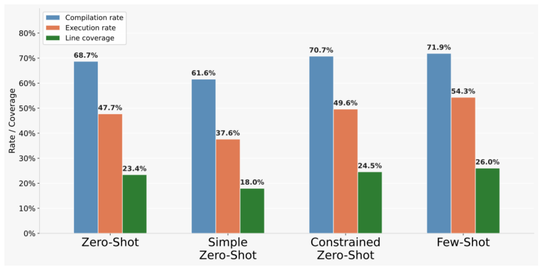
Distribution of per-class line coverage across models, restricted to test classes that compiled and executed successfully.

RQ2 – Bytecode vs. Decompiled Source



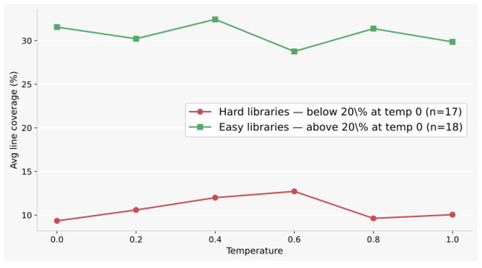
Compilation rate, execution rate, and line coverage by input representation and prompt style.

RQ3 – Prompting Strategy



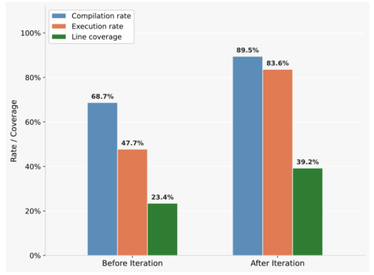
Compilation rate, execution rate, and line coverage by prompt strategy.

RQ4 – Temperature



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

RQ5 – Iterative Repair Effect



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

Discussion

Model selection has a substantial impact: the strongest models, Gemma-4 31B and DeepSeek-V4 Flash, reached compilation rates around 68% and execution rates around 48%, clearly ahead of the other three models. Differences are not explained by parameter count alone. Qwen3-8B consistently outperformed the larger DeepSeek-Coder-V2 16B, suggesting architecture and instruction tuning matter more than scale.

Disassembled bytecode proved more reliable than decompiled source code at producing valid tests, while coverage differences was small enough to fall within natural variance. Full bytecode detail rescued 37.5% of classes that a simplified public-API view failed on, indicating the model uses private members and instructions to reason about how to instantiate and call a class.

Few-shot prompting achieved the strongest overall coverage (26.0%), while constrained zero-shot, which explicitly lists permitted methods, achieved the second highest compilation rate (71.9%), directly addressing hallucinated method calls.

Temperature had only a limited effect when averaged across all libraries, peaking at 0.4, but mattered much more for individual hard-to-test libraries, where moderate temperatures nearly tripled coverage in some cases.

Iterative repair prompting produced the largest single improvement in the study: compilation rose from 68.7% to 89.5%, execution from 47.7% to 83.6%, and line coverage nearly doubled from 23.4% to 39.2%, showing that much of the original failure rate reflects recoverable mistakes rather than a fundamental inability to understand the target class.

Comparison to EvoSuite

The best configuration (few-shot, Gemma-4 31B, temperature 0.4) was compared against EvoSuite on 19 libraries. The LLM produced at least one running test for all 19, while EvoSuite produced none at all for 10 of them. On the 9 libraries where both succeeded, EvoSuite achieved higher mean coverage (60.9% vs 52.7%). The two approaches trade peak coverage for reliability and may complement each other.

Limitations

The sequential experimental design (best model chosen before later RQs) does not capture all interactions between factors. Findings are specific to 50 Maven-hosted Java libraries and 5 evaluated LLMs and may not generalize to other ecosystems or future models. Test suites were evaluated as whole units, so a single failing method could cause an otherwise valid suite to be classified as failed.

Conclusion

Bytecode-based LLM unit test generation is a promising path when source code is unavailable. Model choice and iterative repair matter most; decompilation is not a prerequisite for effective generation. The resulting approach produced usable tests for **all 19** evaluated libraries, compared to 9 of 19 for EvoSuite, trading some peak coverage for substantially greater reliability.

Future Work

- Retain individual passing tests instead of discarding a whole suite over one failing method.
- Use a higher temperature selectively as a retry mechanism for classes that initially fail.
- Combine LLM-based generation with EvoSuite's systematic, search-based input exploration.

References

[1] H. Carter et al., "The 2022 state of open source in financial services," The Linux Foundation, Tech. Rep., 2022.
[2] G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software," Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), 2011.
[3] H. Tan, Q. Luo, J. Li, and Y. Zhang, "LLM4Decompile: Decompiling Binary Code with Large Language Models," Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2024.
[4] Y. Schäfer, M. Nadi, and F. Palomba, "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation," 2024.
[5] X. Chen et al., "ChatUniTest: A ChatGPT-Based Automated Unit Test Generation and Repair Framework," 2024.

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