LLM Test Generation for Java Libraries in Low Context Settings

The Impact of Javadoc on LLM Test Generation Without Source Code

Ipshit Raychaudhuri[1]

Supervisor(s): Sebastian Proksch[1]**, Cathrine Paulsen**[1]

[1]**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: Ipshit Raychaudhuri
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

Many software systems depend on external packages that evolve independently. Updating these packages is important for reliability and security, but compatibility afterwards with dependent systems is not guaranteed. Testing can help verify compatibility, and LLMs show promise in test generation, but current approaches assume access to source code and additional context, something unavailable for packages released only as compiled bytecode, potentially with documentation. The quality of LLM generated tests in such settings is relatively unexplored, as is the context amount needed to generate sufficient tests. Javadoc provides contextual information not present in bytecode, possibly offsetting source code absence. To investigate if it does, I assessed generated test suite quality for bytecode, bytecode and Javadoc, and full source code. Evaluation metrics included the percentage of compiling test classes and passing tests, along with line, branch, and mutation coverages. Results showed that adding Javadoc led to increases over only bytecode in almost all metrics, reducing the gap to source code noticeably. While this combination consumes more tokens than bytecode and source code, the difference is small enough to be outweighed by the quality gains. Thus, I conclude that a bytecode and Javadoc configuration is an effective substitute and a promising option for verifying dependency compatibility in source code's absence.

1 Introduction

Open-source components play a central role in software development, with a recent estimate finding that 96% of commercial code bases contain open-source software [1]. As a result, most software systems are reliant on external dependencies that are updated independently of the systems that use them. Keeping these dependencies up-to-date is critical, as new versions often contain key bug fixes and security patches. However, the burden falls on the developer to ensure that these packages remain compatible after updates. Testing is a common mechanism to assess compatibility, making use of either client or library side tests. Though, in practice, both approaches have drawbacks: client tests may not fully exercise the behavior of a dependency [2], and in many cases, library tests and their source code are unavailable [3].

Large-language models offer a possible remedy to this problem, with recent research displaying that LLMs show promise in automatic test generation [4], [5], [6]. However, current approaches rely on the fact that the complete source code and a large amount of contextual information is readily available. Thus, it is unclear what the quality of LLM generated tests are in limited context settings, which is particularly relevant as many real-world software packages are released on repositories as compiled bytecode, where documentation may be available but source code is absent, either because it is not provided or not findable. This brings up a key point of interest, when source code is absent, how much context is sufficient to generate satisfactory test suites. In particular, Javadoc provides natural-language descriptions of class and method behaviors which cannot be obtained from just bytecode. This information could compensate for the absence of source code and improve the quality of generated test suites. This is precisely what I aim to investigate, more specifically, the main research question I explore in this study is: *To what extent does Javadoc mitigate the impact of the lack of source code context in LLM-based test generation for Java libraries?* By looking into this, I assess the viability of LLM generated test suites, created within limited context scenarios, in verifying the compatibility of external libraries without source code availability.

To systematically investigate the main research question, I consider three different levels of input context for LLM test generation, only bytecode, bytecode and Javadoc, and full source code which contains the Javadoc as comments. To investigate the quality of test suites across these three levels, I define the following sub-questions:

- **RQ1:** What impact does the level of input context have on the compilation rate and execution success rate of LLM generated tests?
- **RQ2:** What impact does the level of input context have on the degree to which LLM generated tests exercise library behavior, measured in terms of line and branch coverage?
- **RQ3:** What impact does the level of input context have on the fault detection capabilities of LLM generated tests, measured in terms of mutation coverage?

Each sub-question above considers all three levels of input context, and builds upon the findings from the one prior to it. I first examine the validity and usability of a generated test suite (RQ1) using the compilation rate, which is the percentage of compiling test classes generated, and the execution success rate, the percentage of tests that successfully run. Then I assess how much do the tests actually probe into the workings of the library (RQ2) using the line and branch coverage. Finally, I evaluate whether the tests actually meaningfully detect faults in the original code (RQ3) using the mutation score, which is the percentage of artificially added faults detected by the test suite. All tests are generated using zero-shot prompting, a prompting strategy where the LLM is told to perform a task without any examples to guide it.

The key findings for each sub-question are as follows. For RQ1, I found that all levels of context produced tests that largely compiled and executed successfully showing that these test suites were largely valid and usable, and were open to deeper investigations. Although additional context did improve the compilation rate, with Javadoc helping to narrow the divide between bytecode and source code. In contrast, execution success rate was less affected by the context level and even decreased slightly with higher contexts. For RQ2, I found that additional context enables the LLM to explore the libraries under test more extensively, with increases in both line and branch coverage. The results for this also increased my confidence in using bytecode and Javadoc as a substitute

for source code when it is not available, as again, the additional context from the Javadoc helped to reduce the gap to source code. Finally, for RQ3, I saw that additional context did not simply make the model explore libraries more, it also improved the fault detection capabilities of the generated test suites. Again, Javadoc lessened the distance to source code here. Since Javadoc consistently bridged the gap to source code in almost every metric measured, I concluded that bytecode with Javadoc is a more than adequate substitute for source code, when it is not available, in LLM test generation. Though it does consume more tokens than bytecode and source code to generate tests, I believe the benefits observed are significant enough to outweigh this increased cost. This in turn presents a strong option for those looking to perform dependency compatibility verification in low context settings.

The novel contributions of my work are twofold. Firstly, I provide an empirical study on the effectiveness of Javadoc in reducing the impact of a lack of source code context in LLM test generation, measured across a multitude of metrics. Secondly, I display the sensitivity of these metrics to the amount of contextual information present, showing how they are affected in terms of their average values, range, and interquartile spread. Together, I believe that these contributions offer new insights and a better understanding of the relatively unexplored realm of low-context LLM test generation.

The paper is structured as follows. Chapter 2 delves into the related work that has been conducted in this field, specifically for test adequacy metrics and LLM generated tests, summarizing their findings, along with how my work differs. In Chapter 3, I describe the methodology used to create and evaluate the generated test suites across the three context levels. Then, in Chapter 4, I lay out the results obtained from the experiments I conducted. Subsequently, in Chapter 5, I discuss these results and their implications, followed by the threats to validity and possible pathways for future research. Chapter 6 discusses the reproducibility of my work as well as its ethical aspects. Finally, Chapter 7 lays out my conclusions.

For those interested, I provide here a Zenodo link containing the scripts used, the generated test suites, and the numerical results for each measured metric: https://doi.org/10.5281/zenodo.20774933.

2 Related Work

In this section, I summarize the findings from related work on both test adequacy metrics and LLM-based test generation, and highlight how my work differs.

2.1 Test Adequacy Metrics

Line and branch coverage are two of the most common criteria used for measuring test adequacy [7]. Line coverage measures the percentage of executable lines run by a test suite, and branch coverage measures the percentage of branches executed. Branch coverage is considered to be stronger than line (statement) coverage as it subsumes it [8]. Though, this is not always the case, as discussed by Zhou et al. Even if a branch is covered, all lines within it may not be necessarily executed, as lines can exit abnormally [9]. The findings of Gopinath et al. also suggest that the situation is not as straightforward as that. They state, under real-world conditions, line coverage is "generally the most effective predictor of suite quality", outperforming branch coverage (as well as block and path coverage) [10]. This suggests that one criterion alone cannot be used to fully evaluate the effectiveness of a test suite, thus a combination of multiple criteria may be more effective, an idea supported by research [11].

In addition to the two previously mentioned, mutation coverage is another metric used to assess test suites. Mutation coverage refers to the percentage of mutants killed by a test suite. A mutant is a class that has been modified to contain a fault, which changes its behavior from the original class, with "killing" one meaning that a test case fails due to the introduction of the fault. Some advantages of mutation coverage are that it provides higher fault-detection capabilities than line coverage [12], and reveals more weaknesses in test suites than branch coverage [13]. However, mutation testing is computationally expensive to run, and some generated mutants are equivalent in behavior to the original program, making them undetectable by test cases and thus distorting the mutation coverage score [14].

Each metric considered here has its own advantages and disadvantages, with no clear winner between them. As such, for the case of my paper I have decided to make use of all three to provide the clearest idea for the adequacy of generated test suites. Differentiating from prior studies, I also compute the compilation rate (percentage of classes with a compiling test class), and the execution success rate (percentage of passing tests). These are two measures which are taken for granted in human-created tests but cannot be assumed to fully hold in LLM-generated tests.

2.2 LLM-Based Test Generation

TestPilot [6] is a system that leverages LLMs to generate tests for JavaScript. The goal of TestPilot was to evaluate the quality of tests generated by models without any additional training. Using gpt3.5-turbo [15], it achieves significantly higher statement and branch coverage than Nessie, a data and feedback driven test generation tool [16]. The TestPilot paper also investigates the impact of removing specific contextual information (error messages, documentation comments, function bodies, examples) from the prompts used. It concludes that the removal of any one component always led to degraded test quality compared to the full prompt in at least one metric (percentage of passing tests, coverage, or non-trivial coverage).

ChatUnitTest [4] and TestSpark [5] are examples of LLM-based test generation tools specifically for Java.

ChatUnitTest aims to resolve two primary issues: the challenge of providing sufficient contextual information to LLMs while not exceeding context length constraints, and the need to introduce effective validation techniques to reduce compilation and runtime errors in LLM generated tests. It addresses the first through its "Adaptive Focal Context Generation" system and the second through rule and LLM-based repair techniques. With these additions, it achieves relatively high line coverage results using gpt-3.5-turbo-0613 compared

to Testspark as well as Evosuite, a search-based Java test generation tool [17].

TestSpark, on the other hand, focuses on providing a more user-friendly experience compared to other technologies. This was done by making it available as an IntelliJ IDEA plugin, allowing users to generate tests entirely within their IDE. With TestSpark, users can modify and analyze the coverage of generated tests, as well as incorporate these tests into their existing test suites.

All three of the above-mentioned systems follow a similar high-level pipeline of prompt construction, test generation, and iterative test repair. However, they all assume that a high amount of contextual information is available for the prompt construction phase, including source code, documentation, signatures, usage examples, etc. Thus, the quality of generated tests in low context scenarios is unclear, as well as the extent to which partial added context in the form of Javadoc mitigates the impact of source code absence. This gap in knowledge is precisely what I aim to address with this paper by examining low context LLM test generation and evaluating the impact of adding Javadoc context.

3 Methodology

In this section I describe the methodology used for my experiments. I detail the pipeline used, with motivations for the technologies chosen. This is done to aid reproducibility for future researchers.

My experiments aimed to systematically compare three different levels of context (bytecode, bytecode & Javadoc, and source code) in LLM test generation for Java libraries. I do this to evaluate the degree to which the addition of Javadoc context improves LLM test generation when source code is not present. The experiments were ran within a controlled environment, where the only changed factor was the input configuration. I did this to ensure that any observed differences was solely due to a change in the level of context.

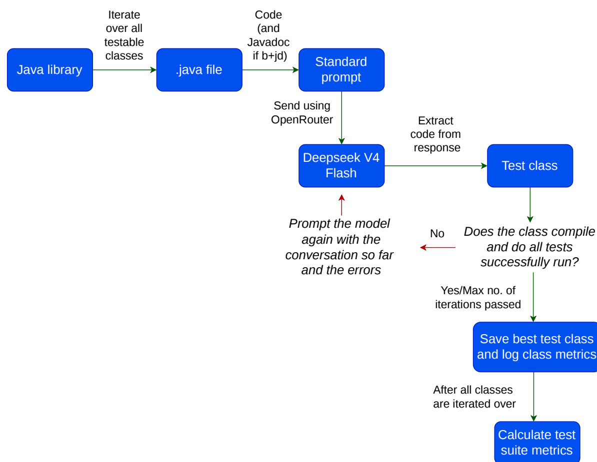

Figure 1: A flow chart representing the pipeline I used to generate tests for a Java library.

I generated the test suites using the pipeline visualized in Figure 1 and described in the bullet points below. It takes a Java library at a certain context level as input. For the bytecode cases, I used a decompiler to convert the compiled .class files to corresponding .java files. I used CFR 0.152 [18] for decompilation as it produces a low amount of deceptive decompilations, and has high semantic and syntactic correctness [19]. I tested another popular Java decompiler, Fernflower [20], but it produced several errors during initial testing, so I opted against using it. The mechanisms of my pipeline are as follows:

1. Given a Java library, it iterates over all classes that are testable. I consider interfaces and abstract classes which contain no executable methods, or any other enclosed logic as "non-testable".

2. For each testable class, I extract the code from the .java file, and place this in a standard prompt which requests the generation of a test class. If the pipeline is set to bytecode and Javadoc mode, I also add the contents of the corresponding Javadoc file to the prompt. To do this, I render the corresponding Javadoc HTML file for the class using Lynx 2.9.0 [21], extract the plaintext content, and include it in the prompt.

3. I then send the filled-in prompt to Deepseek V4 Flash [22]. I selected this model due to its strong performance in test generation tasks during preliminary testing, its much faster response times compared to other "larger" current LLMs, and its recent release as of the conduction of my study, making it well representative of current LLM test generation performance. I used OpenRouter's [23] API to interface with the model as it was considerably faster than running it on locally available hardware. It also made it trivial to adjust the model parameters. To improve reproducibility, I fixed the seed to a constant value (42) and set the temperature to 0, since lower temperature values increase determinism [24] and 0 was the minimum possible value. I set the maximum number of output tokens to 40,000, enough to ensure that almost all outputs were returned without being cut off while still avoiding unnecessarily long outputs and keeping response times reasonable.

4. After the model's response is received, I extract the code from it, and create a test class. I then compile and execute this class with JUnit 5.14.4 [25] and Mockito 5.23.0 [26], though Mockito may not always be required.

5. If the test class successfully compiles, and all tests pass, I add the class to the test suite. If this is not the case, the system enters a repair loop.

6. Within this loop, I re-prompt the model with the complete conversation history, along with the compilation/runtime error messages, instructing it to repair the previously generated test class. The loop continues until a compiling class with all tests passing is generated, or a predefined maximum number of iterations has been exceeded. This iterative approach to repair is common in similar research into LLM test generation [4], [5], [6]. I set the maximum number of repair iterations to 2, as initial results from a fellow student conducting related

research noted that iterations further than this either perform the same or worse in terms of coverage. As such, I considered the additional computational overhead of further iterations unjustified and thus capped it at 2.

7. If the maximum number of iterations are exceeded and a fully fixed test class has not been created, I add the best test class generated to the test suite. Here, best refers to the class with the highest percentage of passing tests (in case of ties, the more recent one is picked). If no iteration produces a compiling result, I save the most recent one with the code commented out so that it can still be analyzed. After the loop terminates, I log class-specific metrics including whether the final test class compiles, the number of tests ran, as well as the number of tests that fail/throw an error/are skipped during execution. I also note the total tokens consumed by the LLM for the current class being processed. I calculate this by summing the values in the "total_tokens" field of the OpenRouter JSON response from each iteration.
8. Once all testable classes are iterated over, I calculate the metrics for the final test suite.

I use the following metrics to evaluate a generated test suite:

- **Compilation rate:** The percentage of testable classes in the library for which a compiling test class was generated.
- **Execution success rate:** The percentage of tests that successfully run in the generated test suite for a library.
- **Line coverage:** The percentage of executable lines of the source code exercised by the generated test suite.
- **Branch coverage:** The percentage of branches of the source code exercised by the generated test suite.
- **Mutation coverage:** The percentage of mutants killed by the generated test suite, where a mutant is a modified Java class containing a fault that changes its behavior from the original, non-mutated class. A mutant is considered killed if a test case fails due to the fault it added.

The compilation and execution success rates are calculated by my pipeline with respective accumulator variables. These measure the amount of non-compiling classes for the compilation rate, and the total number of test cases along with the total number of failures/errors/skips for the execution success rate. I use JaCoCo 0.8.14 [27] to compute the line and branch coverage, and PIT 1.25.1 [28] for the mutation coverage. I only calculate the coverage metrics for the tests that compile and successfully run. If a test class fails to compile, all test cases within it are considered to be non-compiling and thus are excluded from the coverage calculations.

To select the Java libraries for the experiments, an initial set of the top 1000 most popular libraries from the Maven Central Repository (MCR) was created by querying libraries.io. As each library can contain multiple versions, a representative one was selected by picking the most popular version from the latest major release that was available at the current time, using deps.dev for version-level popularity metrics. Any library that could not be found on the MCR, or had no popularity measure in deps.dv, was removed leading to a collection of 930 libraries. I filtered this set to only those that contained published Javadoc documentation, reducing it to 853 libraries. I then performed further filtering to retain only those for which every testable class had a corresponding Javadoc file and contained at least 1 but fewer than 120 testable classes, resulting in a group of 144 libraries.

A selection of 20 libraries was picked from this final set, with a focus on choosing a variety of different library sizes in order to increase the generalizibility of the results. Evaluating each library involved generating, executing, and analyzing tests for each testable class under three different input configurations. This entire process results in substantial extra token usage with each additional library tested. For this reason, as well as due to the time constraints of this project, I set a cap of 20 libraries for the purposes of my experiments. The limit of 120 testable classes was selected for similar reasons, as for libraries larger than this, merely generating a test suite took several hours.

4 Results

In this chapter, I present the results of the experiments conducted on my selected Java libraries, organized in order of the sub-questions, followed by details of the tokens consumed.

4.1 RQ1 Compilation & Execution Success Rates

Before a test suite can be analyzed in a meaningful way, in terms of the extent to which it exercises a library as well as its fault detection capabilities, it must first compile and successfully execute. A test suite that does not compile provides little information, and one that has errors gives information that is distorted. Thus with RQ1, I examine how the validity and the actual practical usability of LLM-generated tests improves with further context.

I measure this in terms of the compilation rate (% of testable classes with a compiling test class) and the execution success rate (% of passing tests). To measure the compilation rate, at the end of my pipeline I divide the amount of compiling test classes by the total number of generated test classes. For the execution success rate, I divide the total number of successfully passing test cases by the total number of test cases created.

The violin plots in figure 2 show that the median compilation rate, while relatively high across all three configurations, increases with the level of context provided. The same is true for the means, with bytecode achieving a value of 85.16%, bytecode plus Javadoc 88.80%, and source code 89.52%. The inter-quartile spread is quite similar for the three context levels, though the range does decrease with increasing context, ignoring extreme points (points lying more than 1.5 times the inter-quartile spread below the first quartile or above the third quartile).

On the other hand, as shown in Figure 3, the median execution success rate seems to very marginally decrease with increasing context, though the percentages observed are far more similar across the three context levels. The means also observe a marginal decrease, bytecode achieves a value of 92.34%, bytecode plus Javadoc 90.89%, and source code

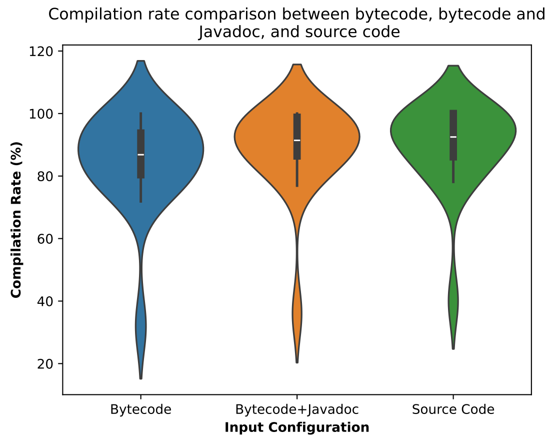

Figure 2: Violin plots comparing the compilation rates (% of compiling test classes) for the tested libraries across the three context levels.

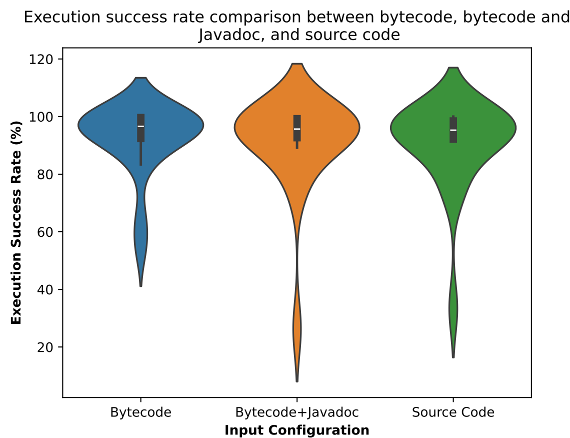

Figure 3: Violin plots comparing the execution success rates (% of passing tests) for the tested libraries across the three context levels.

90.32%. Again, the values here are closer than they were for the compilation rates. The inter-quartile spreads are again fairly similar, and the range decreases with increasing context once more, disregarding outliers. However, the values for both the spread and range are noticeably smaller than the ones noted for compilation rates.

The distributions for both metrics across all three context levels have a fairly normal shape, however all of them have a slight lower bump due to one library that performs particularly poorly. Since it displays low performance across all three context levels, the issue is library specific. Though, I am unsure of what specific aspects of the library causes this, as from a cursory inspection it does not appear to be significantly different from the other libraries in the set.

The results here already begin to show differences between the three context levels. For the compilation rate, adding Javadoc leads to an improvement in the mean by 2.84%, though the source code mean remains better by a smaller margin of 0.72%. The medians see similar increases as well.

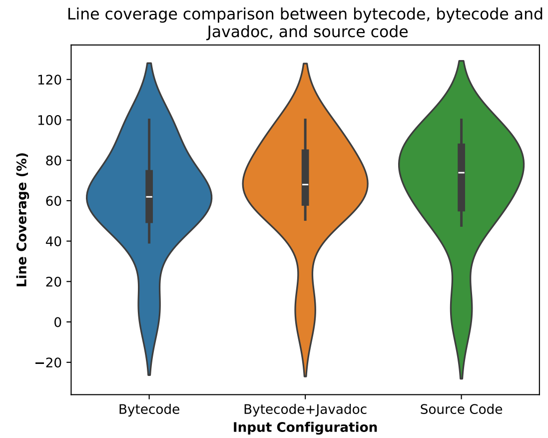

Figure 4: Violin plots comparing the line coverages for the tested libraries across the three context levels.

In contrast, execution success rate appears to slightly worsen with more context as shown by the decreasing medians. The mean also decreases by 1.45% from only bytecode to bytecode and Javadoc, and drops again by 0.57% from Javadoc to source code, though the average values here are noticeably closer than the ones for the compilation rates. Overall, these results display that all three configurations produce tests that largely compile and successfully execute, setting the foundation for further analysis in the next sub-questions.

4.2 RQ2 Line & Branch Coverage

After establishing that the generated tests across all three context levels are largely valid and usable, with RQ2 I assess how thoroughly the tests exercise the behavior of the libraries.

I measure this in terms of the line and branch coverage, observing whether higher context leads the LLM to generate tests that reach more parts of the code. I use JaCoCo to calculate these coverages, calling it at the end of my pipeline, after a full test suite has been generated for a library.

Similarly to the results seen with the compilation rates, the median line coverage also increases as more context is provided, as displayed in Figure 4. The mean values increase as well, with bytecode, bytecode plus Javadoc, and source code scoring 61.89%, 66.09%, and 68.24% respectively. Interestingly, the inter-quartile spread seems to slightly increase with added context, while the range does not appear to follow a context-related pattern, disregarding the extreme points.

This increasing pattern is also observed in the branch coverage, with more context leading to higher medians as shown in Figure 5. The means again increase, with bytecode having a value of 53.46%, bytecode plus Javadoc 56.77%, and source code 58.73%. The range decreases significantly with the addition of Javadoc, not counting extreme values again, with the Javadoc value being very similar to the source code one, however, the inter-quartile spread does not appear to follow a trend dependent on the context.

The distributions for both the line and branch coverage across all three context levels again have a fairly normal shape, as was the case for the compilation and execution suc-

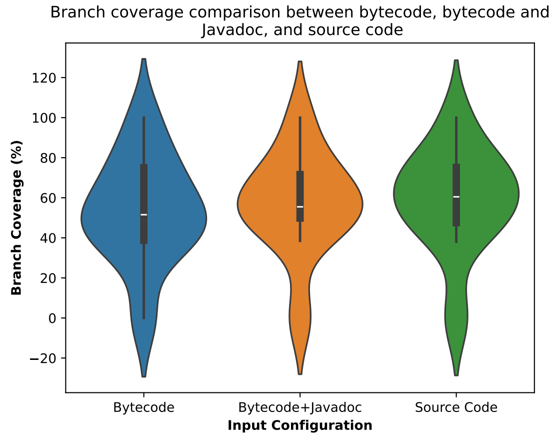

Figure 5: Violin plots comparing the branch coverages for the tested libraries across the three context levels.

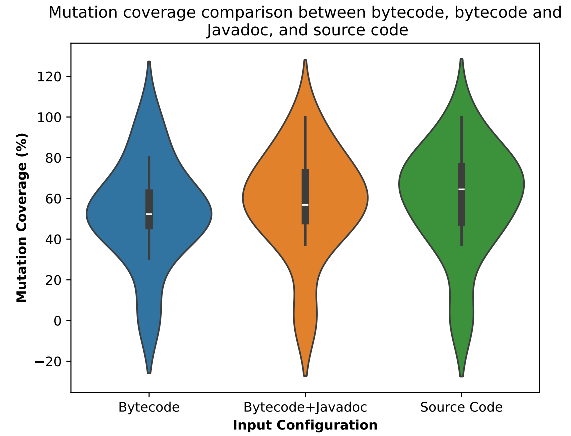

Figure 6: Violin plots comparing the mutation coverages for the tested libraries across the three context levels.

cess rates. The lower bump due to the poorly performing library still exists for both, however, it is much closer to the main body but this is predominantly due to the coverage results themselves having lower values overall, so I do not believe this requires further investigation.

The improvements in the average values for both line and branch coverage highlights the fact that additional context guides the LLM towards deeper explorations of the libraries under test. Javadoc improves in terms of the mean over just bytecode (4.2% for lines and 3.31% for branches), however a slight observable distance still exists for both compared to source code (2.15% and 1.96%). Although, as for the compilation rate, these distances are smaller than the jump from bytecode alone to bytecode and Javadoc. Knowing this, the question now becomes whether more context also translates to further fault-detection capabilities, which I investigate in RQ3.

4.3 RQ3 Mutation Coverage

Line and branch coverage indicate how much of the code under test is executed by a test suite, however, they do not provide a direct idea of the suite's capacity to detect faults.

To measure this fault detection capability of a test suite, I make use of the mutation coverage, measured using PIT. I call PIT at the end of the pipeline, after a complete test suite for a library has been created.

With the mutation coverage, increased context once again leads to higher median values (as displayed in Figure 6), following the patterns of the previous two coverage metrics. However, this time it appears that source code has a substantial improvement in the median value, though this difference is more subdued in the means. Bytecode has a mean of 52.85%, bytecode plus Javadoc 57.53%, and source code 59.01%. Both the inter-quartile spread and the range appear to increase with higher context levels, ignoring outliers for the range.

The shape of the mutation coverage distributions is very similar to those previously observed. Again, I notice a normal shape for the distribution. The bump due to the low performance library is closer to the main body again, but once more I believe that is mainly due to the mutation coverages having lower values on average.

Taking together the results of this sub-question along with RQ2, I observe that higher context leads to test suites that not only exercise library behavior more, but also have higher fault detection capabilities. Once again, Javadoc improves upon performance, with an increase of 4.68% in the mean. Source code achieves a higher mean result than that by 1.48%, with this distance again being smaller than the jump from only bytecode. Though, Javadoc in this case does not significantly help to bridge the divide to the notably higher median value of source code.

4.4 Token Consumption

The previous subsections deal with the overall quality of the test suites. Here, I present the results related to the costs involved in terms of the token usage.

Table 1: Mean and median token consumption per class for the libraries tested across the three context levels

	Bytecode	Bytecode+Javadoc	Source Code
Mean Token Consumption Per Class	17,218.77	21,139.07	18,945.44
Median Token Consumption Per Class	12,891.00	15,976.00	14,680.00

After I process each class in my pipeline, I note the total tokens consumed by the LLM for that class, calculated by summing the values from the "total_tokens" field from the OpenRouter JSON responses across all iterations. Table 1 displays the mean and median token consumption per class. Bytecode consumes the least, with source code noting a slight increase. Interestingly, bytecode and Javadoc consumes the most, exceeding source code by about 2,200 tokens in terms of the mean and about 1,300 tokens in the median.

5 Discussion

In this section, I discuss and further interpret the results from the previous chapter, followed by an exploration of their im-

plications. Then, I mention the threats to validity and potential directions for future research.

5.1 RQ1 Compilation & Execution Success Rates

The results for the compilation rates already importantly show that bytecode configurations are capable of producing test suites comparable to source code. Nevertheless, the presence of more context did lead to improvements, with increased average values as well as reduced ranges (the spread stayed mostly stable). The increases with further context suggest to me that the model becomes more confident in what is callable and how they are callable, resulting in it committing errors less often. The addition of Javadoc here helps to bridge most of the gap to source code, leaving only a marginal difference. Additionally, the distribution of compilation rates having a fairly normal shape gives me confidence that the observed increases are a general broad trend caused by more context, not because a few certain libraries had dramatic improvements in performance. This result already begins to display the potential a bytecode and Javadoc configuration has and that, at least in terms of compilation rate, it performs very closely to source code.

In contrast to the compilation rates, I observed that the execution success rate had a slight inverse relation to the level of context present, in terms of the mean and median values. Though, similarly to the compilation rates, the range still decreased and the spread remained stable. The results seen here refute a blanket claim that more context always leads to better results. Execution success rate is less impacted by the level of context, as the differences in the means are lower than those observed for the compilation rates. It is potentially even independent of it, as the differences between the levels is less noticeable. Although, I should note that this strong claim of independence cannot be confirmed with the current data collected and requires further investigation.

5.2 RQ2 Line & Branch Coverage

The line and branch coverage increasing with further context shows that more contextual information does not just reduce compilation errors, it guides the model towards lines and paths that it would not exercise otherwise. Compared to the compilation rates, the differences in the average values between context levels was slightly more pronounced for these metrics. Again, the distribution has a normal pattern, supporting my belief that further context actually leads to a general increasing trend for these coverage metrics, rather than the means being inflated due to a small set of libraries. The addition of Javadoc in particular leads to noticeable jumps, further proving the potential of the bytecode and Javadoc configuration in the absence of source code. While a gap to source code still exists, the results for RQ2 prove that a bytecode and Javadoc configuration for LLM test generation also shows great promise in the extent to which it exercises the library under test. This result is particularly notable in the case of dependency compatibility verification since it shows that test suites generated with this configuration can provide meaningful code coverage of the dependency under test, comparable to that of source code.

Another point I believe is worth discussing is the surprising marginal increase in the inter-quartile spread of the line coverages with more context. One possible explanation for this increase is that the additional context adds more information for the LLM to process, leading to further branches to explore and more potential test generation strategies which results in greater variance.

5.3 RQ3 Mutation Coverage

The results seen for the mutation coverages shows that additional context does not just lead the LLM to blindly explore further lines and branches of the code. They suggest that there is an actual increase in the understanding of the code under test, leading to more meaningful test suites being created with higher fault detection capabilities. However, more context does lead to higher variance, in terms of both the range and the inter-quartile spread. Again, I apply the explanation I provided for line coverage's spread increase here, namely that higher context potentially leads to further branches to explore and more strategies, causing more variance in the results. The normal-like distribution for the mutation coverages once again supports that the increases are a general trend caused by the context.

Once again, Javadoc helps reduce the gap to source code. Although the distance to the median source code value is relatively large, the Javadoc value still displays an improvement over just bytecode. Furthermore, the smaller difference in the mean values show that the distance between the two is not as drastic as the medians may make it seem. This suggests that a bytecode and Javadoc configuration also has quite promising results in terms of fault detection. A result particularly substantial in showing that it can be used for dependency compatibility verification since fault detection capacity is quite significant in detecting whether an update has broken a dependency's expected behavior. Thus, it is encouraging to see that a bytecode and Javadoc configuration can provide performance close to that of source code.

5.4 Implications

Now that I have presented and discussed the results for each sub-question, I can now talk about the implications of my findings.

The central point that I try to explore in this study is to what extent can a bytecode and Javadoc configuration substitute for source code in LLM test generation. The results show that the addition of Javadoc leads to marked increases in almost every metric I measure, including the compilation rate, line coverage, branch coverage, and mutation coverage, over just bytecode. Javadoc also helps to noticeably bridge the distance that was present to source code in these metrics.

One aspect to note is that bytecode and Javadoc had the highest token consumption of the configurations tested. I attribute this primarily to the much larger prompt sizes of this combination, as the rendered Javadoc HTML files have noticeably more textual content compared to the corresponding Javadoc comments in the source code, notably due to the inclusion of the surrounding context of the library within the file. This leads to increased input tokens as well as potentially more reasoning tokens. Since it is only more expensive

per class by a few thousand tokens, whereas LLM costs are typically measured in millions of tokens, I believe that the difference is small enough for the gains in quality to justify this overhead in practice.

Taking all of this together, I feel comfortable recommending a bytecode and Javadoc configuration as a viable option for LLM test generation when source code is not available.

These findings are particularly relevant in the case of verifying dependency compatibility after updates in a scenario without source code, since the bytecode and Javadoc combination can produce test suites that have high compilation rates, effectively exercise the dependency under test, and provide strong fault detection capabilities. It is also relevant for researchers who are investigating LLM test generation. Current approaches (ChatUnitTest, TestSpark, TestPilot) assume source code access and extensive contextual information. My results show that a lower context configuration of bytecode and Javadoc still generates tests of quality close to full source code, suggesting that the scope of these methods could be expanded to include more libraries which have less context available.

5.5 Threats to Validity

The primary threat to validity is the size of the set used for my experiments. I limited this to 20 due to the effort involved with fully evaluating each library, and due to the time constraints of this project. This is relevant as the differences observed in the average values is relatively smaller compared to the corresponding inter-quartile spreads. With only 20 libraries tested, I cannot rule out that some portion of these differences are due to sampling variation rather than because of context level changes. The cap of 120 testable classes is another aspect to address, as it is uncertain to what extent my findings hold for libraries of much larger sizes. Another point is that due to the requirement of each testable class having a Javadoc file, the scope of my experiments was limited to only libraries which are, relatively speaking, well-maintained. It is therefore unclear on how effective LLM generated test suites created using bytecode and Javadoc would be, for libraries containing only partial documentation. Additionally, when a generated test class is non-compiling, the whole class is removed from coverage calculations when it may only be a few specific test cases that do not compile. This makes it so the coverage results are a bit lower than their "true" values. Finally, it is possible that the LLM used (Deepseek V4 Flash) contained the libraries tested with in its training set, as they are publicly available on the Maven Central Repository, with some having GitHub repositories. This could have introduced a slight positive bias to my results, as the model is working with code it has already previously seen.

5.6 Future Work

Multiple pathways to future work follow. The most obvious is removing the cap of 120 testable classes and 20 libraries to observe how results are affected. Another is observing test generation quality with varying levels of Javadoc presence, where it could be investigated whether LLM generated summaries could be used instead. In fact, a full study comparing developer written Javadoc to such generated summaries in terms of test generation could be carried out. To address the limitation of excluding the entirety of non-compiling test classes from code coverage, future work could identify compilation failures at the test case level. This would allow valid test cases within classes that are partially failing to still contribute towards coverage metrics. An analysis could also be carried out on the types of errors observed between the different levels of context, in order to examine how the thinking of the LLM changes. Due to the possibility that the model was trained on the Java libraries investigated in this study, perhaps experiments could be carried out with completely newly written Java code to see how performance changes. Finally, the effect of using prompting strategies other than zero-shot, such as few-shot, chain-of-thought, constraint-setting, etc., could be analyzed.

6 Responsible Research

In this part, I first discuss the reproducibility of the experiments that I carried out, followed by a reflection on the ethical aspects of my study.

6.1 Reproducibility

I made an active effort to improve the reproducibility of my results. Since my experiments involved the usage of LLMs, which are notoriously non-deterministic, this was the most important aspect I had to address. To minimize variability, I fixed the seed to a constant value of 42 and set the temperature parameter, which controls the creativeness of the model, to the lowest available value of 0.

In Section 3, I explicitly state the values chosen for the other tweakable parameters, including the maximum output tokens and the number of repair iterations. I also specify the precise versions of the software packages used, as well as the particular LLM utilized in the experiments. Furthermore, I detail the methodology used to select the Java libraries examined in this study. Finally, I provide a Zenodo repository link (https://doi.org/10.5281/zenodo.20774933) which contains the list of selected libraries, the scripts used, and the generated test suites along with their measured metric values.

Despite all of this, I cannot guarantee results will be identical when my experiments are reproduced due to two primary reasons. Firstly, the LLM is accessed through OpenRouter, a third party service which over time may direct user requests to computational systems that differ from the ones that my requests were processed by. Secondly, although I set the seed to a constant value and the temperature to 0, the LLM is still not fully deterministic and variations are expected for the same inputs. One way to reduce the impact of this would be to run the experiments multiple times on the model and then take the mean of the output values. Unfortunately, this was not feasible for me to carry out due to both the time constraints of the study, and the significant additional costs the repetition would incur.

6.2 Ethical Aspects

An ethical aspect that must be considered with research that involves LLMs is the environment impact. While my particular experiments were limited to a set of 20 libraries, whereby

such impacts were minimal, larger scale studies in the future could have noticeable consequences for the environment. As such, potential future researchers must take care in balancing extensive experimentation against environmental costs.

Another aspect to consider is the impact of the generated tests. While the results I found in this study are promising, developers must take care to not directly place LLM generated test code into production without thorough manual examinations. Without such checks, there is the potential of incorrect tests being added to test suites, leading to a false sense of security in terms of compatibility. Additionally, the LLM could generate tests that create and remove files on a computer system, so again, care must be taken and any generated test suites must be first thoroughly checked before execution.

7 Conclusions

In this paper, I set out to evaluate the extent to which Javadoc could mitigate the impact of source code absence, in the case of LLM test generation. This was done to assess the feasibility of using LLM generated tests to verify the compatibility of third-party software dependencies after updates, in a scenario where source code is not provided (closed source) or not findable, but documentation is available. To analyze this, I defined three sub-questions, each comparing three different levels of context: bytecode, bytecode and Javadoc, and full source code.

The first sub-question examined how the validity and practical usability of LLM generated tests, in terms of the percentage of compiling test classes produced and the percentage of passing tests, was affected by the context level. The second assessed how the amount of contextual information impacted the extent to which a library's behavior was exercised by the generated test suites, as quantified by line and branch coverage. Finally, the third delved into how the level of context influenced the fault detection capabilities of the generated tests, in terms of the mutation coverage.

The results obtained show that increasing context led to jumps in the observed average values for nearly every metric, with the exception of the percentage of passing tests which displayed a slight decrease. Moreover, I found that the addition of Javadoc resulted in noteworthy rises over simply bytecode in every metric that increased with further context. These gains noticeably narrowed the gap that existed between the bytecode and source code configurations. Though this combination is the most expensive in terms of token consumption, more than both bytecode and source code, I believe that the difference is small enough that it is outweighed by the advantages observed.

Thus, I conclude that a bytecode and Javadoc configuration is a more than sufficient substitute for source code in its absence. The marked improvements over bytecode demonstrates how Javadoc provides valuable contextual information for LLM-based test generation. While it does not fully match the performance of source code, it meaningfully reduces the distance to it, and provides sufficient context for generating effective tests, making it a promising approach to verifying the compatibility of external dependencies.

A Artificial Intelligence Usage

I used Deepseek V4 Flash to generate the tests for this project. Additionally, while I was writing this paper, I used ChatGPT to check for grammar mistakes, and to refine certain phrases to improve clarity. I reviewed all outputs before use and none were directly copied word-for-word as I made an active effort to paraphrase. Furthermore, no new content was created using ChatGPT, I only used it for refinement purposes.

References

[1] Synopsys, *Open source security and risk analysis report*, 2024.

[2] J. Hejderup and G. Gousios, "Can we trust tests to automate dependency updates? a case study of java projects," *Journal of Systems and Software*, vol. 183, p. 111 097, 2022.

[3] C. Paulsen and S. Proksch, "Marco: Compatible version ranges in maven," in *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2025, pp. 910–914.

[4] 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*, 2024, pp. 572–576.

[5] A. Sapozhnikov, M. Olsthoorn, A. Panichella, V. Kovalenko, and P. Derakhshanfar, "Testspark: Intellij idea's ultimate test generation companion," in *Proceedings of the 2024 IEEE/ACM 46th international conference on software engineering: companion proceedings*, 2024, pp. 30–34.

[6] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 85–105, 2023.

[7] R. Kuhn, R. N. Kacker, Y. Lei, and D. Simos, "Input space coverage matters," *Computer*, vol. 53, no. 1, pp. 37–44, 2020.

[8] H. Zhu, P. A. Hall, and J. H. May, "Software unit test coverage and adequacy," *Acm computing surveys (csur)*, vol. 29, no. 4, pp. 366–427, 1997.

[9] Z. Zhou, Y. Zhou, C. Fang, Z. Chen, and Y. Tang, "Selectively combining multiple coverage goals in search-based unit test generation," in *Proceedings of the 37th IEEE/ACM international conference on automated software engineering*, 2022, pp. 1–12.

[10] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 72–82.

[11] G. Gay, "Generating effective test suites by combining coverage criteria," in *International Symposium on Search Based Software Engineering*, Springer, 2017, pp. 65–82.

[12] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 654–665.

[13] A. Parsai and S. Demeyer, "Comparing mutation coverage against branch coverage in an industrial setting," *International Journal on Software Tools for Technology Transfer*, vol. 22, no. 4, pp. 365–388, 2020.

[14] R. Niedermayr, E. Juergens, and S. Wagner, "Will my tests tell me if i break this code?" In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, 2016, pp. 23–29.

[15] OpenAI. "GPT-3.5 Turbo Model OpenAI API," Accessed: Jun. 13, 2026. [Online]. Available: https://developers.openai.com/api/docs/models/gpt-3.5-turbo.

[16] E. Arteca, S. Harner, M. Pradel, and F. Tip, "Nessie: Automatically testing javascript apis with asynchronous callbacks," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1494–1505.

[17] 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*, 2011, pp. 416–419.

[18] L. Benfield. "CFR - yet another java decompiler," Accessed: Jun. 12, 2026. [Online]. Available: https://www.benf.org/other/cfr/.

[19] N. Harrand, C. Soto-Valero, M. Monperrus, and B. Baudry, "The strengths and behavioral quirks of java bytecode decompilers," in *2019 19th International working conference on source code analysis and manipulation (SCAM)*, IEEE, 2019, pp. 92–102.

[20] JetBrains. "Fernflower github page," Accessed: Jun. 12, 2026. [Online]. Available: https://github.com/JetBrains/fernflower.

[21] T. E. Dickey. "LYNX - The Text Web-Browser," Accessed: Jun. 21, 2026. [Online]. Available: https://lynx.invisible-island.net/.

[22] DeepSeek-AI, *Deepseek-v4: Towards highly efficient million-token context intelligence*, 2026.

[23] OpenRouter Team. "OpenRouter," Accessed: Jun. 12, 2026. [Online]. Available: https://openrouter.ai/.

[24] Y. Song, G. Wang, S. Li, and B. Y. Lin, "The good, the bad, and the greedy: Evaluation of llms should not ignore non-determinism," in *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 2025, pp. 4195–4206.

[25] JUnit team. "JUnit," Accessed: Jun. 12, 2026. [Online]. Available: https://junit.org/.

[26] Mockito contributors. "Mockito framework site," Accessed: Jun. 12, 2026. [Online]. Available: https://site.mockito.org/.

[27] EclEmma team. "EclEmma - JaCoCo Java Code Coverage Library," Accessed: Jun. 12, 2026. [Online]. Available: https://www.jacoco.org/jacoco/.

[28] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: A practical mutation testing tool for java," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 449–452.