Evaluating the Impact of Software Context on the Quality of LLM-Generated Test Oracles

Hugo Klijn[1]

Supervisor(s): Annibale Panichella[1]**, Mitchell Olsthoorn**[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: Hugo Klijn
Final project course: CSE3000 Research Project
Thesis committee: Annibale Panichella, Mitchell Olsthoorn, Alex Voulimeneas

Abstract

Testing software is essential for verifying that software is correct and behaves as intended. Large Language Models (LLMs) have shown promise in generating effective test oracles, which are defined as the mechanism used to determine the correctness of the behaviour for a given input to a System Under Test (SUT). Prior work has shown that the type of context provided to an LLM influences the quality of generated oracles. However, existing work often evaluates these oracles by comparing them to human-written assertions, which may not fully reflect real-world oracle quality. This paper investigates how different configurations of context types influence the quality of LLM-generated test oracles. We replicate prior work by evaluating eight context configurations using more realistic quantitative quality measures, including compilation rate, pass rate, mutation score, and test strength. Furthermore, we extend this evaluation by investigating whether compressed context can retain enough relevant information to generate useful oracles. The results suggest that including the focal class improves the quality of LLM-generated assertions the most among the evaluated context types. The effect of Javadoc is mixed: it improves results when available code context is limited. However, its effect is limited or even negative when richer code context is already available. Compression methods effectively reduce the number of tokens, but do not retain the full quality of the generated test oracles. The uncompressed configuration performs best overall. However, when context size is important, the test prefix paired with a summary provides a reasonable trade-off between oracle quality and token usage.

1 Introduction

Generating strong test oracles remains one of the most challenging aspects in automated software testing [2]. Testing is essential for verifying that software is correct and behaves as intended, and automation can significantly improve the efficiency of this process [12, 15]. A test oracle is defined as the mechanism used to determine the correctness of the behaviour for a given input to a System Under Test (SUT) [2]. Recent advancements in Large Language Models (LLMs) have made it more feasible to generate stronger test oracles, reducing the manual effort required in software testing [11].

Prior work has shown that LLM-generated assertions can be correct, and that different types of context have influence on the oracle quality [3, 11]. However, much work evaluates the performance of generated tests by using human-generated assertions as a benchmark by measuring character-level similarity. This assumes that human-generated oracles are correct, which is not always realistic. Thus, there remains a research gap in understanding which types of context improve assertion quality the most using mutation testing, and how such context can be compressed efficiently.

To address these limitations, this paper first replicates prior findings on the influence of context under a different evaluation setting. We extend this work through ablation and compression experiments to investigate what types of context contribute most to the quality of generated oracles by using more realistic evaluation metrics such as compilation rate, pass rate, mutation score, and test strength. This paper considers four types of context: the test prefix, the invoked methods, the focal class, and Javadoc documentation.

This research aims to investigate the influence context has on the quality of LLM-generated test assertions by using more realistic metrics tied more closely to real-world quality evaluation. In addition, due to limitations in how LLMs process long input sequences, not all provided context is equally utilized. This highlights the importance of context engineering, where the goal is not only to provide more context, but to provide the most relevant information with high information density.

The results from our dataset suggest that adding relevant code context improves the quality of LLM-generated assertions compared to using only the test prefix. The inclusion of the focal class contributed most among the evaluated context types. The effect of Javadoc was mixed, results improved most whenever the available code context was limited. However, whenever richer code context is available, its positive effect was limited and even reduced assertion quality when included with the focal class. Finally, compression methods reduced the number of tokens, but did not retain the full quality of test oracles. The uncompressed configuration performed best overall. However, when context size is important, the test prefix paired with a summary provides a reasonable trade-off.

The main contributions of this paper are:

- A replication and extension study of LLM-based oracle generation
- A comparison on how different types of context influence the quality of generated oracles
- An evaluation of how context compression influences the quality of generated oracles
- An adapted reproducible experimental pipeline and replication package to support future research

2 Background & Related Work

This section introduces several relevant topics that recur throughout the paper. We will discuss the oracle problem (2.1), terminology around software testing (2.2), mutation testing (2.3), Large Language Models (2.4), context compression (2.5) and LLM-based assertion generation (2.6).

2.1 The oracle problem

An oracle is the mechanism used to determine the correctness of the behaviour of a System Under Test (SUT) for a given input [2]. In unit testing frameworks such as Java's testing framework JUnit, oracles are typically implemented as assertions. These are used to evaluate a piece of source code by comparing the observed behaviour to an expected outcome. The oracle problem can be defined as the challenge of, given an input for a system, distinguishing corresponding desired, correct behaviour from potentially incorrect behaviour [2]. Since modern programs may have an infinite number of possible observable behaviours, it is generally infeasible to account for all possible scenarios. Thus creating strong test oracles remains one of the most challenging aspects in software testing [2].

Figure 1: Example decomposition of a Java test from the project `retel-io/ari-proxy` into focal class, test class, test prefix, and test oracle.

```
// Focal class
public class HealthReport {
    private List<String> errors;

    (...)

    // Invoked Method
    public HealthReport merge(HealthReport other) {
        if (other != null) {
            return new HealthReport(errors.appendAll(other.errors
     ()));
        }
        return this;
    }
}

// Test Class
class HealthReportTest {

    // Test Case
    @Test
    void mergeReturnSameInstanceIfOtherIsNull(){
        HealthReport report = HealthReport.ok();
        HealthReport mergedReport = report.merge(null);
        assertThat(mergedReport, is(report));
    }
}

// Test Prefix
@Test
void mergeReturnSameInstanceIfOtherIsNull(){
    HealthReport report = HealthReport.ok();
    HealthReport mergedReport = report.merge(null);
}

// Test Oracle (assertion)
assertThat(mergedReport, is(report));
```

2.2 Test terminology

A test case typically consists of input data, method calls and one or more oracles. In addition, we can define the *test prefix* for a given oracle to be all code inside the test case that precedes the oracle, including any prior oracles [11]. During a test run a test case typically invokes one or more methods, we define these methods as the *invoked methods*. The primary method being tested by a test case is referred to as the *focal method*, and is part of the *invoked methods*. The *focal method* usually exists in a class of other methods, we define this as the *focal class*. An example of the decomposition of a Java test class and its focal class can be seen in Figure 1. After execution, a test case either passes, meaning all assertions are satisfied, or fails, meaning one or more assertions were not satisfied.

2.3 Mutation testing

Mutation testing is a fault-based testing technique that can evaluate the quality of a test by introducing small modifications in the source code and afterwards produces a *mutation adequacy score* [7]. A widely used framework for mutation testing in Java is PIT, which we will be using in this paper. The core idea of this technique is that small modifications, known as *mutants*, are introduced in the source code of the SUT. Examples of such mutants are changing comparison operators, arithmetic operators, or logical conditions. The test suite is then executed against this mutated version of the SUT. If a test fails on the mutated code, the mutant was caught and is said to be *killed*. Otherwise, if a test did not fail, the mutant was not detected, meaning the mutant *survived*. The mutation score is computed as defined by the PIT framework [13]:

$$\text{Mutation score} = \frac{\text{killed mutants}}{\text{total mutants}} \tag{1}$$

We also use test strength as a metric in our evaluation. Which is computed as defined by the PIT framework [13]:

$$\text{Test strength} = \frac{\text{killed mutants}}{\text{killed mutants} + \text{survived mutants}} \tag{2}$$

Additionally, we report the compilation rate and the pass rate. These are computed as the ratio of all tests to compiling and passing tests on unmodified source-code.

Mutation testing is frequently used in research on automated test generation and oracle generation since it provides a quantitative measure of test quality [16]. A high mutation score and test strength indicates that more mutants were killed, thus indicating a stronger test suite since the test cases are more effective at finding real faults.

2.4 Large Language Models

Large Language Models (LLMs) are artificial intelligence (AI) based on deep learning. They typically contain tens to hundreds of billions of parameters, that are pre-trained on massive datasets of text [10]. LLMs are constrained by their context window, which limits the length of input prompts they can process [18]. Given an input prompt, these models generate human-like text by predicting likely token sequences based on patterns in the training data. Recent advancements in LLMs have made them much more feasible for code generation [11]. As a result, they have enabled automated software engineering tasks such as code generation and test generation.

2.5 Context Compression

Context compression aims to reduce the amount of information provided to an LLM, while retaining enough information for the LLM to solve the desired task [5]. Source code context can become very large when including large methods and classes to a prompt. Since LLMs have limited context windows, it makes sense to compress this context in order to fit more context into a smaller context window. In test oracle generation, the use of compression introduces a trade-off. On the one hand, reducing the amount of information reduces the context size, but removing important information may make it harder for the model to generate the desired output. Thus compression should aim to increase information density by preserving the information most relevant to the task.

2.6 LLM-based assertion generation

Prior work has investigated LLM-based assertion generation. Molinelli et al. show that LLMs can generate correct test oracles, and

Table 1: Repositories used in this research and corresponding number of datapoints

Repository	Data points
whatsapp-business-java-api	4
nbvcxz	168
cbor-java	152
word-wrap	71
solarpositioning	35
ari-proxy	14
semver4j	2
urnlib	84
Total	530

that additional context can improve oracle quality [11]. In addition, Bodicoat et al. show that adding the focal class improves assertion quality the most [3]. However, both studies evaluate the performance of their generated tests by using human-generated assertions as a benchmark by measuring character level similarity. This assumes that human-generated oracles are correct, which is not always realistic. Human-written assertions may overlook edge cases, while LLMs may be better at identifying these overlooked edge cases [12]. Therefore, there remains a research gap in understanding which types of context improve assertion quality the most using mutation testing, and how such context can be compressed efficiently.

3 Methodology

The goal of this research is to investigate how context influences the quality of LLM-generated test oracles. To this end, the paper first replicates prior findings on the influence of context by Molinelli et al. [11]. We then extend this work through ablation and compression experiments to investigate what types of context contribute most to assertion quality.

3.1 Research Questions

Based on the overarching research question, *How does context influence the quality of LLM-generated test oracles?*, we define the following research questions:

RQ1 *What types of code context, individually or in combination, contribute most to the quality of LLM-generated test oracles?*

RQ2 *What effect does the addition of documentation to code context have on the quality of LLM-generated test oracles?*

RQ3 *To what extent can compression methods on the best-performing context configuration retain the quality of test oracles while reducing context size?*

To answer these research questions, we designed the following pipeline. First, we built a dataset based on popular real-world repositories. Using these repositories, we construct our prompts containing different types and amounts of software context. These prompts are then fed into an LLM to generate assertions. Finally, the generated assertions will be evaluated based on a set of metrics of which the results are saved and available for comparison.

3.2 Dataset Construction

For our dataset we used *gitbug-java*, a publicly available and reproducible benchmark of Java bugs featuring 199 bug-fixes sourced from 55 relevant open-source repositories from 2023 [17]. The benchmark includes both a buggy version, and a manually fixed version for each bug. A recent benchmark is suited for this study considering that LLMs may already have been exposed to older well-known bugs and their fixed versions during their training. This is especially relevant since most benchmarks that have contributed to software engineering research like Defects4J are outdated, with bugs that are currently 10 to 15 years old [8].

To construct our dataset we follow the algorithms proposed by Molinelli et al. [11]. First a script downloads all 55 open-source repositories and saves them in our project.

Then a script checks whether the repository satisfies the following conditions: the project is a Maven project, has a single `pom.xml` file, and uses JUnit as its testing framework. After these checks, the script mines all commit-related information and saves this in a JSON file. Resulting in 33 eligible projects.

Now, a final algorithm generates the dataset. This algorithm first performs several checks and discards an additional project. The algorithm then finds all assertions from the result of the mining sub-process, and analyses the Java project. This final dataset is represented as a JSON object containing an entry for each test class and focal class pair in the repository. Each entry contains relevant metadata about the test class, the focal class and all data points. The data points correspond to test cases separated in a signature, body and an assertion. While creating this dataset, some test methods contain multiple assertions. These are removed by the algorithm such that only the assertions prior to the masked assertion are present. This resulted in 32 candidate repositories within our dataset for further research.

After running mutation testing on these 32 candidate repositories, 24 repositories failed due to build issues or because the initial test run failed. Meaning only 8 repositories had a successful run for all context configurations. Thus the final dataset consists of 8 different repositories with a total of 530 data points as shown in Table 1.

3.3 Context Configurations

In order to investigate the influence of context, we must define the types of context used in this study. We distinguish four different types of context: the test prefix (TestPrefix), the invoked methods (InvokedMethods), the focal class (FocalClass), and Javadoc documentation (Javadoc).

This separation is based on work by Primbs et al. and Molinelli et al. [11, 14], who distinguish several types of context in their research, including the test prefix, focal method, invoked methods, the focal class and the test class.

The test prefix is used as a baseline since it contains the smallest amount of context for which we aim to generate oracles. We include invoked methods, but will not include the focal method as a separate context type, since the focal method is always included in our definition of invoked methods. The focal class is included because it can be inferred from the focal method and provides additional class-level context, such as fields, constructors, and methods. Although

the focal class has overlap with the invoked methods, this overlap is incomplete, since invoked methods may also belong to classes other than the focal class. The test class is excluded because it does not directly follow from the invoked methods or the focal class. This keeps the configurations focused on production-code context, without introducing additional test-code context beyond the test prefix. Finally, in order to answer research question two, we include Javadoc documentation for each of the context types, including the test prefix, since some repositories contain documented test cases.

To evaluate which types of context contribute most to oracle quality, we define eight configurations. These configurations allow paired comparisons in which one context type is added or removed while the rest of the setup remains unchanged. All context configurations are defined as follows:

- **TestPrefix** The prompt contains only the test prefix
- **TestPrefix-Javadoc** The prompt contains the test prefix and test prefix Javadoc
- **TestPrefix-InvokedMethods** The prompt contains the test prefix and the full implementation of all invoked methods without Javadoc
- **TestPrefix-FocalClass** The prompt contains the test prefix and the full implementation of the focal class including fields, constructors and defined methods without Javadoc
- **TestPrefix-InvokedMethods-FocalClass** The prompt contains the test prefix, the full implementation of all invoked methods and the full implementation of the focal class without Javadoc
- **TestPrefix-InvokedMethods-Javadoc** The prompt contains the test prefix and the full implementation of all invoked methods with Javadoc
- **TestPrefix-FocalClass-Javadoc** The prompt contains the test prefix and the full implementation of the focal class including fields, constructors and defined methods with Javadoc
- **TestPrefix-InvokedMethods-FocalClass-Javadoc** The prompt contains the test prefix, the full implementation of all invoked methods and the full implementation of the focal class with Javadoc

3.4 Prompt Construction

In addition to the previously defined context configurations, we define a general instruction that is inserted at the start of each call to the LLM. The combination of this general instruction paired with the source code of a context configuration will be the final prompt that will be given to the LLM. The full general instruction is shown in Figure 2. This instruction contains guidelines the LLM should follow during assertion generation. It explains the task, being the creation of a single assertion and the way it should format its output. In addition, the instruction explains that the LLM should put its result inside a custom tag to make sure it can be identified correctly in later stages. Additionally, it states that the answer must be less than 200 characters long, to ensure the LLM only provides an assertion without any other explanation or reasoning.

Figure 2: General instruction for generating a Java test oracle. From the codebase by Molinelli et al. [11]

```
===== TASK DESCRIPTION =====
Given a Java test prefix and contextual information such as the
      javadoc, the signature and the source code of the methods
      invoked within the test prefix and the field and the
      methods declared in the focal class and in the test class,
      generate the test oracle to add to the current test prefix
      substituting the mask placeholder <mask_id> (the oracle
      must conform the api of given JUnit version).
Put your answer between: [oracle][/oracle] and reply only with
      the assertion (do not add any context or comments, reply
      only with the assertion, like for example `[oracle]
      assertEquals(a,b)[/oracle]`. Ensure you do not forget both
      the opening tag `[oracle]` and the closing tag `[/oracle]`).

If the prefix should throw an exception, respond with [oracle]
      EXCEPTION[/oracle].
The content of the answer must be less than 200 characters.
Do not show me the reasoning behind your answer, but only the
      answer in the mentioned format.

===== TASK TO EXECUTE =====
Here is the query for this task:

Query:
<QUERY_INPUT>

Answer:
```

3.5 LLM Inference

The LLM used in this paper is gpt-oss:20b-cloud. This model uses 20 billion parameters, was developed based on feedback from the open-source community and was designed to support reasoning-heavy tasks. The model was trained on a text-only dataset consisting of trillions of tokens, with a focus on STEM, coding, and general knowledge, and has a knowledge cutoff date of June 2024 [1]. This model is therefore a valid choice for our research since we need to run inference on text-only content with a focus on generating code in the form of test assertions. With this LLM we can generate test assertions by providing it with a prompt for each of the different configurations. To ensure reproducibility we set the random seed to 42. All other unspecified decoding parameters were left unchanged from the default settings. The model was run through Ollama Cloud for all prompt configurations.

3.6 Compression Configurations

After the initial experiment, we investigate the influence of compression methods. The goal of these compression methods is to preserve the most relevant information, while using less tokens than the original input would have. We will use the compression methods on the best-performing uncompressed context configuration based on our evaluation metrics from the initial experiment.

Work by Xu et al. shows that summaries can be used as compact representations of long context, reducing computational cost and helping a model focus on task-relevant information [19]. To this end, We prompt the same gpt-oss:20b-cloud model with a summarization

Figure 3: Instruction for summarizing context [11].

```
===== TASK DESCRIPTION =====
Given a Java test prefix and optional contextual information,
      generate a concise textual summary of what the test prefix
      does.
The contextual information may include method signatures, focal
      class fields and methods, and test class fields and methods.
       Use this context only to understand the behavior of the
      test prefix. Do not summarize the contextual information
      itself unless it is directly needed to explain the test
      prefix.
The summary should be useful as input for a later step that
      generates the missing test assertion. Therefore, preserve
      information about the exercised behavior, relevant inputs,
      object state, method calls, return values, side effects,
      and expected outcome. Do not generate the assertion itself.

Do not generate Java code or markdown code, only provide English
      words.
The content of the answer must be less than 200 tokens.
Do not show me the reasoning behind your answer, but only the
      answer in the mentioned format.

===== TASK TO EXECUTE =====
Input:
<QUERY_INPUT>

Summary:
```

task as seen in Figure 3. We investigate the influence of the summary as a standalone configuration, and paired with the test prefix.

Prior work by Dinella et al. shows that an oracle generation model can use compact unit context, such as method signatures or docstrings, together with a test prefix to infer test oracles without requiring the full implementation of the method under test [4]. Thus we will investigate the influence of only the method signatures paired with the test prefix as well. We therefore define the following compression methods:

- **LLM Summary** The prompt contains only a summary created by an LLM based on the best-performing context configuration.
- **Test Prefix + LLM Summary** The prompt contains the test prefix and a summary created by the LLM based on the best-performing context configuration.
- **Test Prefix + Skeleton** The prompt contains the test prefix paired with the method signatures and fields of the best-performing context configuration.

3.7 Evaluation Metrics

To evaluate the LLM-generated assertions and to answer our research questions, we will use a mutation testing framework for Java called PIT. This framework performs mutation testing and outputs four evaluation metrics:

- **Compilation rate** The ratio of compiling assertions compared to the total number of assertions.
- **Pass rate** The ratio of passing tests compared to the number of compiling tests.
- **Mutation score** The ratio of killed mutants compared to the total number of mutants.
- **Test strength** The ratio of killed mutants compared to the total number of covered mutants.

To evaluate each LLM-generated assertion, we follow the algorithm proposed by Molinelli et al. [11]. This algorithm identifies the test prefix to which the assertion belongs. After identifying this prefix, the algorithm replaces a placeholder tag with the newly generated assertion. When all assertions are put in, the algorithm performs initial run to first check if the assertions compile and then if they pass. Then mutants are created and the tests are run again. The algorithm then reports the number of killed, survived and covered mutants for each configuration. The higher the ratios of the evaluation metrics are, the higher the quality of the assertions is. Since mutants can be semantically equivalent to the source code, we will treat these measures as an indication rather than a perfect measure of oracle quality.

3.8 Answering the research questions

In order to answer our research questions we will use the following subset of results to evaluate them:

RQ1 *What types of code context, individually or in combination, contribute most to the quality of LLM-generated test oracles?*
This question is answered by looking at configurations that combine the test prefix with the other code context types. The baseline in this case is the configuration containing only the test prefix. We will then compare this baseline against the configurations that add either the invoked methods (T + I), the focal class (T + F) or both (T + I + F). This way we can estimate the relative change each type of context makes in comparison to only the base test prefix.

RQ2 *What effect does the addition of documentation to code context have on the quality of LLM-generated test oracles?*
This question is answered through paired comparisons between configurations with and without Javadoc documentation. These comparisons are T versus T+J, T+I versus T+I+J, T+F versus T+F+J, and T+I+F versus T+I+F+J.

RQ3 *To what extent can compression methods on the best-performing context configuration retain the quality of test oracles while reducing context size?*
This question is answered through comparing the compression configurations against the best-performing uncompressed configuration as a baseline. In this comparison we will evaluate the performance metrics discussed before, as well as the token count of the total input to infer what compression method, if any, performs best.

4 Results & Discussion

In the following section we will answer the research questions based on the results we obtained by the experiments.

4.1 RQ1: Performance of different context types

4.1.1 Results. Table 2 presents the results from the first experiment, in which each row is represented as a context configuration. The first row is the baseline containing only the test prefix. The other rows contain this baseline including one or more of the other

Table 2: Oracle quality by context configuration with the test prefix (TestPrefix) as T, the invoked methods (InvokedMethods) as I, the focal class (FocalClass) as F. The metrics used are compilation rate as Comp, pass rate as Pass, mutation score as Mut score, test strength as Test str

Context	Comp	Pass	Mut score (Δ)	Test str (Δ)
T	68.4%	69.1%	40.4% (+0.0)	68.8% (+0.0)
T + I	69.1%	76.7%	43.7% (+3.3)	73.1% (+4.4)
T + F	76.7%	79.2%	48.2% (+7.7)	77.4% (+8.7)
T + I + F	74.2%	83.8%	45.9% (+5.5)	75.2% (+6.4)

Note. Values in parentheses indicate percentage-point changes relative to the baseline configuration T.

context types. The columns report the compilation rate, pass rate, mutation score and the test strength. The values in parentheses show the percentage-point change compared to the baseline.

From Table 2 we observe that the baseline containing only the test prefix **T** obtains the lowest score in all metrics. Looking at the quantitative measures of test quality, we observe that the test prefix by itself reaches a mutation score of 40.4% and a test strength of 68.8%. Adding the invoked methods **T + I** improves these scores to 43.7% and 73.1%, an increase of 3.3 and 4.4 percentage-points respectively. Adding the focal class **T + F** results in a larger improvement, with scores of 48.2% and 77.4%, an increase of 7.7 and 8.7 percentage-points respectively compared to the baseline. The configuration that combines both the invoked methods and focal class **T + I + F** also improves over the baseline, reaching a mutation score of 45.9% and a test strength of 75.2%. However, this configuration does not outperform **T + F** based on these two metrics. It does reach the highest pass rate of all configurations, being 83.8%.

4.1.2 Discussion. The results suggest that adding more context generally helps the LLM to generate higher-quality assertions. The type of context appears to matter, with the focal class providing more useful information than the invoked methods. One possible explanation for this observation could be that the focal class includes more relevant information for the LLM to help generate assertions. To further explore this explanation, we manually inspected a sample of prompts from both configurations. This inspection suggested that in several prompts, the focal class contained more methods than the number of invoked methods. Thus likely containing more relevant information. This could therefore explain why **T + F** outperformed **T + I**. It is also notable that **T + I + F** did not outperform the **T + F** configuration despite having more context. A possible explanation could relate to the partial overlap the focal class and the invoked methods have, since the invoked methods are not necessarily distinct from the methods in the focal class. After manual inspection of a sample of prompts from this configuration, it became clear that there often was an overlap of methods. Due to this overlap, the amount of the context increased while the information stayed similar. This may relate to the difficulty LLMs have when using long context. Which is supported by Prior work from Liu et al., which explains that LLM performance significantly degrades in the middle of their input context [9].

Table 3: Oracle quality by context configuration with and without Javadoc with the test prefix (TestPrefix) as T, the invoked methods (InvokedMethods) as I, the focal class (FocalClass) as F, and Javadoc documentation (Javadoc) as J. The metrics used are compilation rate as Comp, pass rate as Pass, mutation score as Mut score, test strength as Test str

Context	Comp	Pass	Mut score (Δ)	Test str (Δ)
Baseline: T				
T	68.4%	69.1%	40.4% (+0.0)	68.8% (+0.0)
T + J	67.6%	75.4%	43.8% (+3.3)	73.8% (+5.1)
Baseline: T + I				
T + I	69.1%	76.7%	43.7% (+0.0)	73.1% (+0.0)
T + I + J	63.6%	77.9%	44.3% (+0.6)	73.7% (+0.6)
Baseline: T + F				
T + F	76.7%	79.2%	48.2% (+0.0)	77.4% (+0.0)
T + F + J	74.2%	82.9%	46.7% (−1.5)	73.5% (−4.0)
Baseline: T + I + F				
T + I + F	74.2%	83.8%	45.9% (+0.0)	75.2% (+0.0)
T + I + F + J	78.7%	81.2%	47.1% (+1.1)	75.8% (+0.7)

Note. Values in parentheses indicate percentage-point changes relative to the baseline configuration shown at the start of each block.

Conclusion RQ1

Adding more context generally improves the quality of LLM-generated assertions, compared to only the test prefix. The inclusion of the focal class generally contributes most to the quality among the evaluated context types.

4.2 RQ2: Influence of Javadoc documentation

4.2.1 Results. Table 3 presents the results from the first experiment. The table is separated into four parts, in which each part has its own baseline which is compared to the same configuration extended with Javadoc. The columns report the compilation rate, pass rate, mutation score and the test strength. The values in parentheses show the percentage-point change compared to the baseline.

From Table 3 we observe that for the configuration with the baseline containing only the test prefix **T**, most evaluation metrics have increased when the test prefix was extended with Javadoc. The mutation score becomes 43.8%, a 3.3 percentage-point increase compared to the baseline of 40.4%. The test strength becomes 73.8%, a 5.1 percentage-point increase compared to the baseline of 68.8%.

The configuration with a baseline of the test prefix and the invoked methods **T + I** saw an increase of all metrics except for the compilation rate when extending it with Javadoc. The mutation score and test strength both increased by 0.6 percentage-points.

The test prefix and the focal class **T + F** configuration saw a decrease for all metrics except for the pass rate when including Javadoc. The mutation score decreased by 1.5 percentage-points, and the test strength decreased by 4.0 percentage-points.

Finally, the context configuration including the test prefix and both the invoked methods and focal class **T + I + F** saw an increase in

all metrics except for the pass rate. The compilation rate increased from 74.2% to 78.7%, while the mutation score increased by 1.1 percentage-points. The test strength increased by 0.7 percentage-points.

4.2.2 Discussion. The results suggest that the inclusion of Javadoc has a different effect depending on the type of context. In particular, configuration **T + J** performs better than the baseline. An explanation for this can be that the LLM now has access to more useful information about the test prefix and the underlying implementation. This suggests that while the inclusion of Javadoc is not common practice in Java development, it can improve the quality of the generated assertions when code context is limited.

The slight increase of 0.6 percentage-points when including Javadoc to the baseline consisting of the test prefix and the invoked methods **T + I** can be explained similarly to the previous point. The results suggest that the extra added information may have caused the LLM to have more insight in the underlying implementation behind the invoked methods. Making the LLM more effective at generating high quality assertions.

The most notable negative effect was observed from the **T + F + J** configuration. Although the pass rate increases, all other metrics decrease. A possible explanation for this observation is that the focal class already provides enough relevant information for the LLM. With the inclusion of Javadoc introducing possibly redundant or less precise information, making the model rely less on the relevant source code and thus producing lower quality assertions.

The **T + I + F + J** configuration shows a balanced result. Adding Javadoc slightly improves both the mutation score and the test strength. While these effects are relatively small, it does suggest that including Javadoc in context including a large amount of relevant information adds limited value.

Conclusion RQ2

The inclusion of Javadoc does not always improve the quality of LLM-generated assertions. The biggest improvements are made whenever the code context is limited, like in configuration **T**. But whenever there is a rich code context available, the effects are limited and might even reduce assertion quality.

4.3 RQ3: Performance of compression methods

4.3.1 Results. Table 4 presents the results from the second experiment. The first row is the uncompressed baseline containing the test prefix and the focal class **T + F**. The remaining rows contain the compression configurations. The columns report the compilation rate, pass rate, mutation score, test strength, inference time in seconds, average number of tokens used in the prompts, and the fraction of tokens compared to the baseline. The values in parentheses show the percentage-point change compared to the baseline.

From Table 4 we observe that all compression methods performed worse than the baseline. The plain **Summary** configuration shows a decrease of 18.7 percentage-points of the mutation score, and a decrease of 9.9 percentage-points of the test strength compared to the baseline. In addition, both the compilation rate and pass rate decreased significantly. However, the number of tokens used decreased by 1000.4, being 23.7% of the baseline. The **T + Summary** configuration shows a decrease of 6.5 percentage-points of the mutation score, and a decrease of 2.8 percentage-points of the test strength compared to the baseline. In addition, both the compilation rate and pass rate decreased and the number of tokens used decreased by 807.6 being 38.4% of the baseline. Finally, the **T + Skeleton** configuration shows a decrease of 4.6 percentage-points of the mutation score, and a decrease of 4.9 percentage-points of the test strength compared to the baseline. In addition, both the compilation rate and pass rate decreased slightly, and the number of tokens used decreased by 586.6, being 55.3% of the baseline.

4.3.2 Discussion. None of the compression methods outperformed the uncompressed **T + F** baseline. This is expected since the amount of informative context is reduced. However, the number of tokens used was also lower than the baseline, introducing a tradeoff between the number of tokens and the assertion quality.

The large decrease in all metrics for the plain **Summary** configuration suggests that the summary removed too much relevant context. Whenever we compare the results to the **T + Summary** configuration, we observe that this configuration performs much better, indicating that absence of the test prefix had an impact on the model's ability to create a good test oracle, thus explaining the low scores for the **Summary** configuration.

The **T + Skeleton** configuration and the **T + Summary** configuration perform relatively similarly, but show different strengths. **T + Skeleton** achieves a higher compilation rate and mutation score. This may be because the skeleton preserves more of the original source-code structure, while still reducing the amount of context compared to the uncompressed configuration. This is consistent with RQ 1, where configurations with more informative context generally produced higher quality assertions.

In contrast, **T + Summary** achieves a higher pass rate and a higher test strength, while using fewer tokens than **T + Skeleton**. This suggests that the summary may preserve useful behavioural information, despite containing less source-code structure. If mutation score should be optimized, the **T + Skeleton** configuration performs generally better. However, if token reduction is preferred, the **T + Summary** configuration offers a stronger trade-off. This comparison should also consider the extra summarization step required for the **T + Summary** configuration. In situations where this summarization step is too expensive, the **T + Skeleton** configuration may still be preferred.

Conclusion RQ3

Compression methods reduce the number of tokens, but do not fully preserve the quality of test oracles. When context size is not a constraint, the uncompressed **T + F** configuration performs the best. If context size is relevant, the **T + Summary** configuration offers a strong trade-off by using only 38.4% of the tokens, while preserving most of the oracle quality. **T + Skeleton** may be preferable when mutation score is prioritized, but requires more tokens than the **T + Summary**.

Table 4: Oracle quality and efficiency by context compression configuration with test prefix (TestPrefix) as T and the focal class (FocalClass) as F. The metrics used are compilation rate Comp, pass rate Pass, mutation score Mut score, test strength Test str

Context	Comp	Pass	Mut score (Δ)	Test str (Δ)	Tokens (Δ)	% Tokens	Time
T + F	76.7%	79.2%	48.2% (+0.0)	77.4% (+0.0)	1311.7 (+0.0)	100.0%	4.31s
Summary	32.4%	73.9%	29.5% (−18.7)	67.5% (−9.9)	311.2 (−1000.4)	23.7%	2.58s
T + Summary	66.8%	76.5%	41.7% (−6.5)	74.7% (−2.8)	504.0 (−807.6)	38.4%	3.35s
T + Skeleton	73.1%	65.2%	43.6% (−4.6)	72.6% (−4.9)	725.1 (−586.6)	55.3%	3.30s

Note. Values in parentheses indicate percentage-point changes relative to the baseline configuration T + F. Time is reported in seconds.

5 Threats to Validity

In this section we will discuss the threats to the validity of this research. First the threat of data leakage is explained (5.1). Afterwards, potential problems with the number of data points used in our research is explained (5.2).

5.1 Data Leakage

There is a risk of data leakage since there is an overlap in the training cutoff date of the LLM used in this paper and the date that the commits originate from in the *gitbug-java* dataset. The cutoff date for our LLM, gpt-oss:20b-cloud, is the first of June 2024 [1]. While the commits in the dataset are from 2023 [17]. This means that there is an overlap of about one and a half years in which the LLM could have learned the code which could make the use of this LLM unreliable. The LLM used was relatively small and the overlap of these dates was relatively short making it unlikely that data leakage was present.

5.2 Generalizability

The number of data points is something to consider when making claims about the results. As discussed in the methodology section, the dataset consisted of a total of 55 repositories with reproducible bugs. From these 55 repositories, only eight were used. As a result, the number of available data points also decreased, which might introduce a potential selection bias. This means that our sample of repositories is less likely to be a significant representation of all repositories, and thus we have to be careful when making claims based on our findings from the experiment.

6 Responsible Research

6.1 Reproducibility

For the research to be reproducible, a full experimental pipeline is available on GitHub[1]. The repository includes a README.md with the steps required to reproduce the experiment. Where possible, random seeds and model settings were fixed or documented.

6.2 Dataset

The dataset used in this study is based on the publicly available *gitbug-java* project, which has an MIT licence and is cited properly as requested by its authors [17]. The repositories used in this dataset might contain personal information of the authors. However, this study only uses technical code-related content, making the potential use of this information unlikely.

[1]https://github.com/Hugo-Klijn/impact-of-context

6.3 Ethical use of LLMs

This study relies on the use of LLMs as part of the experimental pipeline. Output generated by LLMs may be incorrect or misleading and are therefore not treated as ground truth but rather as artifacts to be evaluated. The `gpt-oss:20b-cloud` model was trained on filtered data based on the filters from GPT-4o [1].

6.4 AI statement

Generative AI tools were used to support writing and editing of this paper, as well as assistance in coding tasks. All AI-generated content was reviewed, verified and edited by the author on correctness.

7 Conclusions and Future Work

In this paper, we investigated what influence context has on the quality of LLM-generated test assertions, by using metrics tied more closely to real-world quality evaluation. We also studied whether compressed context can reduce context size while preserving assertion quality.

In our dataset, the results suggest that adding relevant code context improves the quality of LLM-generated assertions compared to using only the test prefix. The inclusion of the focal class contributed most among the evaluated context types. The effect of Javadoc was mixed: it improved results most whenever the available code context was limited. However, when richer code context was available, its effect was limited and even reduced assertion quality when included with the focal class. Finally, compression methods reduced the number of tokens, but did not retain the full quality of test oracles. The uncompressed configuration performed best overall. However, when context size is important, the test prefix paired with a summary provides a reasonable trade-off. But if mutation score is prioritized, the test prefix paired with the skeleton may be preferred.

Future work could evaluate these findings on more repositories. In addition, the generated assertions could be evaluated against the buggy versions of the gitbug-java dataset to extend the potential fault catching abilities of the generated assertions even further. Finally, future work could evaluate the effect of various other, more advanced context-compression techniques like Retrieval Augmented Generation (RAG) or by manipulating text embeddings as discussed by Jha et al. [6].

8 Acknowledgements

A special thanks to Professor Annibale Panichella and Professor Mitchell Olsthoorn from the TU Delft for the feedback and guidance during this project.

References

[1] Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

[2] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The Oracle problem in software Testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 11 2014.

[3] Adam Bodicoat, Gunel Jahangirova, and Valerio Terragni. Understanding LLM-Driven Test Oracle Generation. *ArXiv.org*, 1 2026.

[4] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K Lahiri. Toga: A neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2130–2141, 2022.

[5] Tao Ge, Jing Hu, Lei Wang, Xun Wang, Si-Qing Chen, and Furu Wei. In-context autoencoder for context compression in a large language model. *arXiv preprint arXiv:2307.06945*, 2023.

[6] Siddharth Jha, Lutfi Eren Erdogan, Sehoon Kim, Kurt Keutzer, and Amir Gholami. Characterizing prompt compression methods for long context inference. *arXiv preprint arXiv:2407.08892*, 2024.

[7] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[8] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014.

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

[10] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large language models: A survey. *arXiv preprint arXiv:2402.06196*, 2024.

[11] Davide Molinelli, Luca Di Grazia, Alberto Martin-Lopez, Michael D. Ernst, and Mauro Pezzè. Do LLMs Generate Useful Test Oracles? An Empirical Study with an Unbiased Dataset. *Gesellschaft für Informatik (GI)*, 1 2026.

[12] Paulo Augusto Nardi and Eduardo F Damasceno. A survey on test oracles. 2015.

[13] PIT Mutation Testing. Command line quick start, 2026. Accessed: 2026-06-05.

[14] Severin Primbs, Benedikt Fein, and Gordon Fraser. Assert5: Test assertion generation using a fine-tuned code language model. In *2025 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 12–23. IEEE, 2025.

[15] Debra J Richardson, Stephanie Leif Aha, and T Owen O'malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th international conference on Software engineering*, pages 105–118, 1992.

[16] Ana B Sánchez, Pedro Delgado-Pérez, Inmaculada Medina-Bulo, and Sergio Segura. Mutation testing in the wild: findings from github. *Empirical Software Engineering*, 27(6):132, 2022.

[17] André Silva, Nuno Saavedra, and Martin Monperrus. Gitbug-java: A reproducible benchmark of recent java bugs. In *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024.

[18] Xindi Wang, Mahsa Salmani, Parsa Omidi, Xiangyu Ren, Mehdi Rezagholizadeh, and Armaghan Eshaghi. Beyond the limits: A survey of techniques to extend the context length in large language models. *arXiv preprint arXiv:2402.02244*, 2024.

[19] Fangyuan Xu, Weijia Shi, and Eunsol Choi. Recomp: Improving retrieval-augmented lms with compression and selective augmentation (2023). *arXiv preprint arXiv:2310.04408*, 2024.