Enhancing Unit Tests using ChatGPT-3.5

Stefan Creasta[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 23, 2024

Name of the student: Stefan Creasta
Final project course: CSE3000 Research Project
Thesis committee: Annibale Panichella, Mitchell Olsthoorn, Casper Poulsen

Abstract

Manually crafting test suites is time-consuming and susceptible to bugs. The automation of this process has the potential to make this task more appealing. While current tools like EvoSuite manage to obtain high coverages, their generated tests are not always readable. Recent literature indicates that Large Language Models (LLMs) could address readability and comprehension issues. Our objective in this study is to explore the capabilities of ChatGPT-3.5-Turbo in enhancing existing Java unit tests. We have designed an algorithm that sends multiple prompts to the LLM and overwrites the test cases with the ones received from GPT-3.5. Thus, we have assessed its performance by measuring the initial mutation score of the test suite with the new coverage. The benchmark consists of 16 non-trivial Java classes, on which we performed 80 runs of our algorithm. The results indicate that after one run, GPT-3.5 increases mutation coverage by 23% on average for isolated classes. However, for classes with dependencies, it is less reliable, often producing code with run-time or compile-time errors. Through this paper, we hope to emphasize the importance of ongoing research in this domain to optimize LLMs for providing better test cases.

1 Introduction

Testing is a crucial aspect of Software Engineering, as it assures the correctness of the program and that it behaves as intended. However, manually developing test cases that verify this can be a laborious task, which is time-intensive and prone to human error [1]. Therefore, the automation of this process is sought-after, as this would result in less time spent by a Software Engineer to reason about test cases or perform debugging.

Previous attempts have been made to automatically craft test cases, generally by employing *fuzz testers*: tools that generate random and unexpected inputs to test a program for vulnerabilities. Due to the non-determinism involved and the ability to produce a large number of inputs, these tools have emerged as a state-of-the-art technique in exposing software bugs [2].

Despite the efficacy of fuzz testing, recent advancements in Artificial Intelligence, particularly in the development of Large Language Models (LLMs), suggest that these models might offer unique advantages for automatically generating tests. LLMs have shown great potential in understanding and generating human-like text based on vast amounts of data [3]. This capability can be used to create more contextually relevant test cases, potentially surpassing the input generation capabilities of traditional fuzzers.

In this study, we have explored the potential of OpenAI's ChatGPT-3.5-Turbo[1] to enhance existing Java unit test cases. We have devised an algorithm that sends multiple prompts to the language model and overwrites the original test cases with the new ones, received from the LLM. We aim to maximize the mutation score, which measures the effectiveness of a test suite by determining the percentage of artificially introduced faults (mutations) it can detect. It often provides a more stringent evaluation than traditional coverage metrics like line or branch coverage [4], and thus being the metric that we focused on. The classes that we tested this approach on were extracted from the SF110 repository[2] and the Apache Commons library[3]. We observed that for isolated classes, GPT-3.5 can improve upon the original test suite, killing 7 additional mutants on average, for every 9 prompts sent. The LLM performs worse for classes with dependencies, detecting only 2 additional mutants, while also providing crashing code more frequently.

This paper is structured as follows: in Section 2, we will discuss the general topic in more detail, alongside the approaches of previous studies. Section 3 will showcase our general methodology for automatically improving test suites. The experimental protocol, benchmark, and other crucial aspects will be presented in Section 4. While Section 5 will discuss the obtained results, Section 6 will address possible ethical, legal, and scientific considerations. Threats to validity will be discussed in Section 7. Lastly, Section 8 will conclude this study and emphasize future directions.

2 Background and Related Works

Our main objective of improving test cases with GPT-3.5 is closely related to the challenge of distinguishing correct behavior from potentially incorrect behavior given an input, known as the *"test oracle problem"* [5]. Solving the oracle problem ensures optimal test suites. Although it is an inherently difficult problem, we can attempt to address it by using mutation score, as it correlates better with test effectiveness than line or branch coverage [6]. High line or branch coverage can be achieved with trivial assertions, which do not necessarily contribute to the robustness of the test suite and are not the focus of this study.

Search-Based and Fuzz Testing (SBFT) tools have been a cornerstone in automatically generating test suites for the past decades. The most notable one, EvoSuite[4], which uses an evolutionary algorithm, has outperformed other tools, achieving a median line coverage of 97% on classes extracted from multiple libraries [7]. While the results are significant, as EvoSuite's main focus is on maximizing coverage [8], SBFT tools in general struggle to understand semantic information, which can lead to unreadable tests [9].

An alternative to automatically creating test suites is to employ LLMs, which have the potential to enhance software test generation. For example, LLMs have proven effective in improving the readability of Python test cases [10] and coverage for JavaScript test suites [11]. Fan et al. provide a comprehensive survey encompassing various software engineering applications of LLMs [12], while Wang et al. focus specifically on testing [13]. Both studies emphasize the

[1] https://openai.com/chatgpt/

[2] https://www.evosuite.org/experimental-data/sf110/

[3] https://commons.apache.org/

[4] https://www.evosuite.org/

prevalence of language models in test generation and improvement in the literature. However, Tang et al. demonstrate that EvoSuite achieves 18.8% more code coverage than ChatGPT [14], which underlines the need for continued research to optimize language models for testing.

Processes that involve LLMs are stochastic due to the unpredictable nature of these models [15], making the replication of results a challenging task. Nonetheless, our experiments reflect findings in the literature, as we have observed that almost two-thirds, if not more, of the LLM-generated code crashes, depending on the number of prompts sent. This is similar to the study conducted by Yuan et al., where only 24.8% of tests created by GPT-3.5 were executable [16], highlighting the importance of effective prompt engineering. The results regarding crafting test suites from scratch vary widely: Siddiq et al. reported as high as 80% coverage on classes from the HumanEval dataset [17], while achieving merely 2% on the SF110 repository [8]. However, our focus is not to generate test suites from scratch but to improve existing ones. As highlighted by some researchers working at Meta, the TestGen-LLM tool's purpose is to aid software engineers by suggesting potential improvements, not replace them [18].

Moreover, we acknowledge the work of Dakhel et al., which employs an iterative approach similar to ours, aimed at maximizing the mutation score [19]. Their method demonstrates promising results, achieving 100% mutation coverage for up to 70% of the classes tested. While their focus is on evaluating the effectiveness of Codex and Llama-2-chat, our study assesses the capabilities of ChatGPT-3.5 in enhancing test suites. Although we achieved high mutation scores for certain classes, we did not attain 100% mutation coverage in any run. Several factors may account for this difference, including the use of a different LLM. Additionally, Dakhel et al. tested Python classes, whereas we focused on Java classes. The number of iterations may also contribute to the disparity in performance, as their method involves a maximum of 10 iterations compared to our 9.

3 Approach

In Subsection 3.1, we will showcase why we chose the dynamic approach, while in Subsection 3.2 the general methodology will be provided. Subsection 3.3 will detail how we structured our prompts. Implementation details will be described in Subsection 3.4.

3.1 Dynamic Approach

To enhance tests and evaluate improvements, various methods can be utilized, including SBFT tools, LLMs, or a combination of both [20]. When leveraging an LLM, there are two primary strategies: static and dynamic. Given the longer response times associated with LLMs, the static approach offers speed, while the dynamic one ensures maximum test coverage due to its iterative nature. Since this study focuses on maximizing the number of mutants detected, the dynamic approach is our ultimate goal. By starting with a single initial prompt in the dynamic approach, we can also evaluate its performance, thereby assessing the static strategy.

A key consideration is the number of prompts sent to the LLM to achieve a high mutation score: the more iterations of the dynamic approach needed to increase coverage, the greater the computational resources and processing time required. As previously mentioned, LLMs can be unreliable, as the generated code does not always function properly. Compilation and run-time errors are particularly problematic as they hinder our ability to assess the model's performance accurately. To extract a valid mutation score, we employed ChatGPT-3.5 to fix these issues. Due to its ability to understand language, it can also grasp the errors present in the code. Consequently, whenever we encountered crashing code, we extracted the errors from the console and fed them back into the LLM. This iterative feedback loop, facilitated by the dynamic strategy, is another reason for choosing this method.

3.2 General Overview

By using multiple iterations, we were able to take full advantage of the LLM to either fix the faulty code generated by GPT-3.5 or further increase the mutation coverage. Each prompt was unique: the initial prompt included the source code and the initial test suite, while subsequent prompts presented the outcomes of the LLM's responses and requested improvements.

Initially, we manually extracted the source code and its corresponding test suite. Assuming that the initial test cases work correctly, we focused solely on determining the mutation score. The first prompt, containing the source code, test suite, and mutation score, was sent to GPT-3.5. The LLM's responses always included updated code due to our prompt design.

The iterative process began by overwriting the current test suite with the code from the LLM's response and recording the results (crashes or mutation scores if the code worked correctly). We then decided whether to continue or stop the algorithm based on the number of prompts sent, halting the process if this number exceeded the threshold of 9.

If the algorithm continued, we compiled and ran the test suite, automatically extracting its mutation coverage or any console errors. These results were used to craft the next prompt, which was sent to GPT-3.5, continuing the iterative improvement process. Figure 1 details the top-level architecture of our approach.

3.3 Prompt Engineering

Crafting an effective prompt is crucial when working with an LLM, as the quality of the prompts directly affects how well the task is communicated to the language model. We have observed that by including more details in the prompt, we can greatly improve the performance of our algorithm.

Initially, we simply asked ChatGPT-3.5 to provide additional test cases to increase the mutation score. The results were not favorable, as the test cases would often fail for various reasons, such as crashing assertions or missing the correct package. In the case of classes with dependencies, the tests failed to compile due to incorrect lambda expressions or issues with inferring the types of generic arguments. These errors will be addressed in more detail in Section 5. The structure of the provided tests would also vary from response to

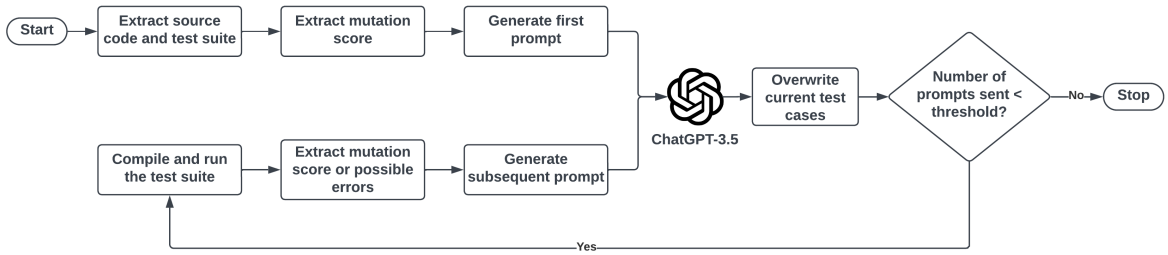

Figure 1: The top-level architecture of the algorithm demonstrates the crafting of the initial prompt, the iterative loop utilized in the dynamic approach, and the generation process for subsequent prompts.

response, and sometimes, the response would not even contain code. Parsing these responses proved to be challenging, if not impossible, in case there would be no test cases provided. By being more explicit and asking the LLM to provide the entire code, we managed to mitigate these issues. An example of an initial prompt is the following:

```
I have Java classes and JUnit tests for them.
I am interested in their mutation score and how
to improve it. Currently, I have 1 Java class.
This is the XClass.java class:
"package bytevector;
public class XClass {
    int value;
    public XClass(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
}"
These are the test cases:
"package bytevector;

import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class XClassTest {
    @Test
    public void test_get() {
        XClass x = new XClass(0);
        assertEquals(0, x.getValue());
    }
}"
I use Pitest to compute the mutation score,
which currently is: 0.0. Can you provide me
with the entire code for the tests, such that
the mutation score is improved? Please include
the package on the first line
```

To address the problem of crashing assertions, we enumerated the errors encountered while running the tests in a subsequent prompt and asked GPT-3.5 to fix them. While this did not guarantee the absence of incorrect assertions, it increased the likelihood of receiving proper test cases in subsequent responses. An example of a prompt which contains an error is the following:

```
I have included your suggestions in the test
suite. I get the following errors: XClassTest.
test_get:10 expected:<1> but was:<0>. Can you
provide me with the entire code for the tests,
such that the errors are fixed? Please include
the package on the first line
```

Interestingly, we observed that to increase the mutation score, it was necessary to send at least two prompts. Often, the first prompt resulted in new test cases with incorrect assertions. The second prompt's role was to enumerate the problems and allow the LLM to fix them. However, if the number of errors was high, the LLM would generally fail to fix them, even after multiple identical prompts were sent. Because of this, we decided to limit the number of errors in the prompt to three, which resulted in fewer crashing test suites.

Another issue we encountered was the stagnation of the mutation score despite prompts asking GPT-3.5 to increase the number of mutants killed. To address this, we had to be more specific once again. We refined our prompts to be more explicit about the task by also adding terms like "changing logical operators" or "altering arithmetic operators" to properly describe the type of mutants we wanted to detect. An example of a subsequent prompt that asks the LLM to further improve the mutation coverage is the following:

```
I have included your suggestions in the test
suite. Using PiTest, the current mutation score
is 0.0. I need to further improve this score by
adding more robust and comprehensive test cases
Can you suggest additional test cases to include
in the test suite to improve the mutation score?
Please ensure the new test cases are designed to
cover edge cases, exceptional conditions, and any
missed scenarios. Here are the specific types of
changes I would like to focus on: changing logi-
cal operators, altering arithmetic operators,
boolean mutations, relational mutations, numeri-
cal mutations, string mutations, return value mu-
tations, increment/decrement mutations, statement
mutations. The goal is to catch as many mutations
as possible. Please provide only the entire code
(with the old and new test cases), and make sure
to include the package on the first line
```

3.4 Implementation Details

The project is divided into two parts: the Java source code accompanied by the test suite, and the Python scripts which automatically run and improve the tests. The dependency between PiTest[5] and the program is managed by Maven[6].

In one script, the mutation score is determined using PiTest and Python's subprocesses, which enable us to extract the mutation score for the present classes. Another Python script overwrites the current test cases with improved ones and sends prompts to ChatGPT-3.5 via OpenAI's endpoint. The third script is responsible for creating the prompts and the last one can run the Java tests normally and catch any errors.

The version of Python we used is 3.9.18, and for Maven we used 3.9.6, which facilitated easy dependency management for the project. We used Java version 21.0.1, although any version that properly supports the classes and test cases would suffice. Additionally, we used the default settings for PiTest.

4 Study Design

In Subsection 4.1, we will highlight the three proposed research questions that we have addressed. The benchmark of classes will be explored in Subsection 4.2, and the experimental protocol will be described in Subsection 4.3.

4.1 Research Questions

Our main objective is to assess the performance of GPT-3.5 in improving Java unit tests. Generally, to determine the effectiveness of tests, line or branch coverage metrics are employed. However, as discussed earlier, having a high line or branch coverage does not necessarily indicate a thoroughly tested source code. Therefore, mutation coverage was the metric we utilized to assess the efficacy of test suites.

To verify the capabilities of GPT-3.5, we have devised three research questions:

- **RQ1:** *To what extent can the static approach improve the mutation coverage for Java unit tests?*
- **RQ2:** *To what extent can the dynamic approach improve the mutation score for Java unit tests?*
- **RQ3:** *How many prompts are required for the dynamic approach, in order to maximize the mutation coverage?*

To address the first question, we will investigate whether the number of mutants killed increases after the initial iteration of the algorithm. For the second question, we will verify whether the mutation score improves with each iteration. Finally, for the third question, we will determine which iteration shows the largest improvement.

4.2 Benchmark

To obtain a comprehensive analysis of GPT-3.5's capabilities, we assessed it on a diverse set of classes. Table 1 presents statistics for each class: each row corresponds to a different class considered in this study. The last three rows represent three groups of classes with dependencies that we have explored; for these groups, the statistics (number of lines, initial mutation score, and number of mutants) were aggregated. The second column shows the number of lines in each class, while the third column presents the baseline mutation score before any improvements. The fourth column depicts the total number of mutants. The fifth column indicates whether the tests were manually written; if the checkmark (✓) is missing, the tests were generated by EvoSuite. The final column indicates whether the classes were taken from SF110; if the checkmark (✓) is missing, the classes were extracted from the Apache Commons library. As stated earlier, we devised two scenarios - classes with dependencies and classes without dependencies - due to the differing results obtained for each.

We selected five standalone classes - classes without dependencies - that initially had low mutation scores. These are ByteVector, Utils, BooleanComparator, CommandLine, and Queue. Except for BooleanComparator, all were extracted from the SF110 repository. These classes involve boolean, numerical, and string manipulations, as well as loops, which allow for a variety of mutants. Additionally, we examined whether manually written tests versus EvoSuite-generated tests impact the LLM's effectiveness. We observed that when improving upon existing tests for all these classes, there was no noticeable difference in performance, regardless of whether the tests were manually crafted or not.

We also studied classes with dependencies, exploring three groups of classes, all from the Apache Commons library. The first group included ComparatorChain and ComparatorUtils. In this group, there was no inheritance; ComparatorChain simply utilized some functions from ComparatorUtils, and vice-versa. The second group comprised AbstractCollectionDecorator (abstract), TransformedCollection, Transformer (interface), and NOPTransformer. This group allowed us to assess GPT-3.5's ability to understand inheritance dependencies and generate test cases accordingly. The third group consisted of FixedSizeList, BoundedCollection (interface), Unmodifiable (interface), UnmodifiableIterator, and AbstractSerializableListDecorator (abstract). These classes had dependencies and involved a significant amount of numerical manipulations.

All of these classes differ in scope and size. Through this, we hoped to achieve an extensive understanding of how well ChatGPT-3.5 can improve test suites.

4.3 Experimental Protocol

Before employing the dynamic approach for a particular class, we had to preprocess the classes with dependencies. Firstly, we organized the initial tests into a single class by copying all of the test cases into one file. This reduced the size of the prompts and allowed the LLM to focus on a single test suite.

Additionally, we limited the number of dependent classes to five. This was necessary for the third group of classes with dependencies, which originally contained seven classes. The results for all seven classes were unsatisfactory, with 84 out of 90 responses from the LLM producing crashing code. To address this, we removed two classes, AbstractListIteratorDecorator and AbstractCollectionDecorator, by deleting

[5]https://pitest.org/

[6]https://maven.apache.org/

Table 1: The table showcases for every class or group of classes the lines of code, initial mutation coverage (before any improvements), the total number of mutants, and whether the initial test suite was manually written or generated by EvoSuite. It also indicates whether the classes were extracted from SF110 or the Apache Commons library.

Class	Number of lines	Initial mutation score	Total number of mutants	Manual written tests	SF110
ByteVector	294	15	138		✓
Utils	175	21	24	✓	✓
BooleanComparator	190	35	24	✓	
CommandLine	198	41	32		✓
Queue	232	38	34		✓
Group 1	593	27	29	✓	
Group 2	462	29	38	✓	
Group 3	404	10	52	✓	

any *override* annotation. Whenever a function from these removed classes was called, we replaced those calls with the actual implementation of the functions.

These preprocessing steps improved the reliability of GPT-3.5 for the classes with dependencies. The standalone classes required no preprocessing.

We applied the dynamic approach to all of these classes, conducting 10 separate runs of our algorithm for each class. For each run, we sent 9 prompts and recorded the outcome for each response. Therefore, each run yielded 10 results: the initial mutation score and the outcomes of the 9 subsequent prompts. These outcomes indicated whether the tests crashed or the mutation score changed. The results will be explored in Section 5.

Generally, it takes about 1 minute to run PiTest for a class and its corresponding test suite, craft the prompt based on the mutation score, send it to GPT-3.5, receive a response, and extract the code from it. Therefore, one entire run could take 9 or 10 minutes.

While our primary focus was to maximize the mutation score, this was not possible whenever we received crashing test cases. As a result, fixing these cases with ChatGPT-3.5 became a priority. We observed that most errors in the extracted classes could be fixed with only a few prompts, never requiring more than 9 prompts to resolve a bug. This is the reason for using 9 iterations for each run in our experiments.

5 Results and Discussion

In Subsection 5.1 we will explore the performance of GPT-3.5 for the isolated classes, whereas for Subsection 5.2 the results for classes with dependencies will be considered. A discussion about the results and their significance can be found in Subsection 5.3.

5.1 Standalone classes

We first considered the five classes without dependencies. As highlighted in Table 1, the classes vary in scope and size: a certain mutation score for the ByteVector class will not be equivalent to the same score for the BooleanComparator class, due to the much larger number of mutants in ByteVector. Therefore, to compare the runs between these classes and how effective GPT-3.5 was in detecting mutants, we only examined the number of mutants killed, rather than the mutation score. Furthermore, as each test suite has a different number of initial mutants detected, we plotted only the additional number of mutants killed by the LLM. Thus, we considered the baseline number of mutants killed to be 0 in our figures, to properly compare the performance of ChatGPT-3.5. We aggregated the outcomes for every iteration of our algorithm and every run and class. The mean and median values of the number of mutants killed by the LLM can be viewed in Figure 2. Whenever the test suite would crash, we would not include a mutation score of 0 in the statistic with the number of mutants killed: we would only represent the cases where the program functioned properly. Figure 3 showcases the number of times the test suite present in the LLM's response crashed. Moreover, Table 2 illustrates comparisons between the means of mutation score distributions. We employed Welch's t-test due to differences in variances and calculated the effect size using Cohen's d to quantify the magnitude of the variations between distributions, aiding in the interpretation of the significance of our results.

We observe in Figure 2 that both the mean and median values exhibit a consistent upward trend, with the number of detected mutants peaking in the final iteration of the algorithm. Additionally, by the ninth iteration, we have noted a general decrease in the number of crashes compared to earlier stages.

For Welch's t-test, we selected a significance threshold (α) of 0.05 and compared the p-value with α to determine statistical significance. We compared the distribution obtained for the i^{th} iteration with the baseline distribution, which contains only zeros in our case. For every comparison, we observed statistical significance between distributions, as the p-values were lower than α, leading us to reject the null hypothesis that the means of the distributions are equivalent. Notably, across comparisons from the second to the eighth iteration, we observe an upward trend followed by a subsequent decline, as indicated by the effect size (Cohen's d).

Based on these plots, we can address the original research questions. Sending only one prompt likely does not lead to an increase in the mutation score, as emphasized by the median being zero. However, the likelihood of receiving functional code remains high. When multiple prompts are sent,

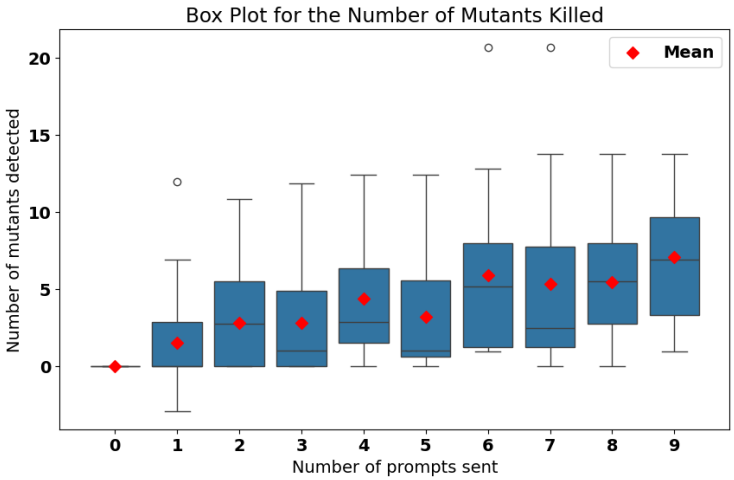

Figure 2: Box plots for the number of mutants killed by ChatGPT-3.5 for the standalone classes. The X-axis represents the number of prompts sent to GPT-3.5, while the Y-axis showcases the number of mutants killed. One box spans from the first quartile (Q1) to the third (Q3), its length denoting the interquartile range (IQR). The whiskers extend from the edges of the box to the smallest and largest values within 1.5 times the IQR from Q1 and Q3, respectively. The red diamond presents the average number of mutants detected. The line inside a box showcases the median (Q2), while the outliers are depicted as dots

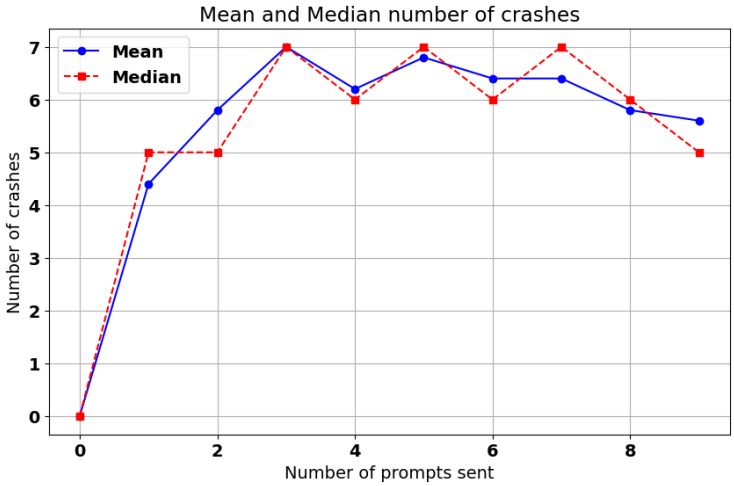

Figure 3: Mean and median values of number of crashes for the standalone classes' tests. The X-axis represents the number of prompts sent to GPT-3.5, while the Y-axis showcases the number of crashes

Table 2: Comparisons of mutation score distributions between iterations using Welch's t-test. The second row showcases the value of Cohen's d, while the third row presents the p-value. Each column represents the comparison between distributions: the i^{th} column shows the comparison between the initial distribution (iteration 0) and the distribution obtained from the i^{th} iteration

Comparison	0-1	0-2	0-3	0-4	0-5	0-6	0-7	0-8	0-9
Cohen's d	2.8552	4.2378	2.8945	4.9287	3.2755	4.7274	4.0135	6.5928	8.7453
P-value	0.0082	0.0004	0.0118	0.0001	0.0051	0.0002	0.0009	0.000002	0.000000019

although the reliability of the code decreases (as the chance of receiving crashing code increases), there is a noticeable increase in the number of mutants killed. Lastly, for maximizing the mutation score, sending 9 prompts emerges as the optimal approach.

5.2 Classes with Dependencies

We will now consider the three groups of classes with dependencies. A similar overview of GPT-3.5's performance has been made, where in Figure 4 the number of mutants detected can be visualized, while the number of crashes is depicted in

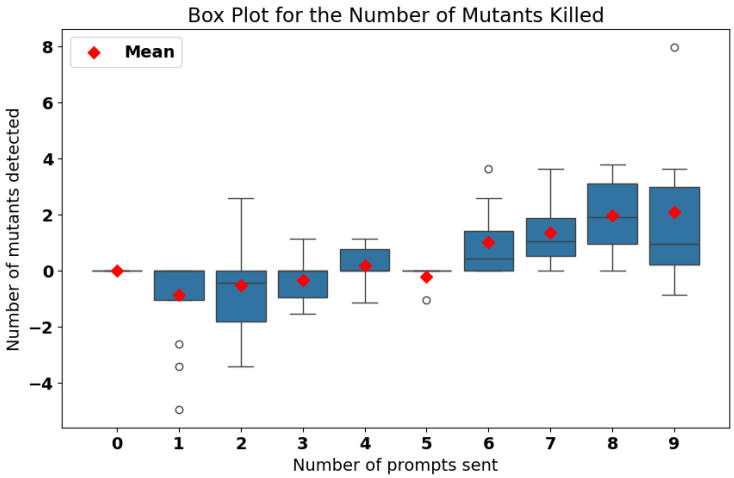

Figure 4: Box plots for the number of mutants killed by ChatGPT-3.5 for the classes with dependencies. The X-axis represents the number of prompts sent to GPT-3.5, while the Y-axis showcases the number of mutants killed. One box spans from the first quartile (Q1) to the third (Q3), its length denoting the interquartile range (IQR). The whiskers extend from the edges of the box to the smallest and largest values within 1.5 times the IQR from Q1 and Q3, respectively. The red diamond presents the average number of mutants detected. The line inside a box showcases the median (Q2), while the outliers are depicted as dots

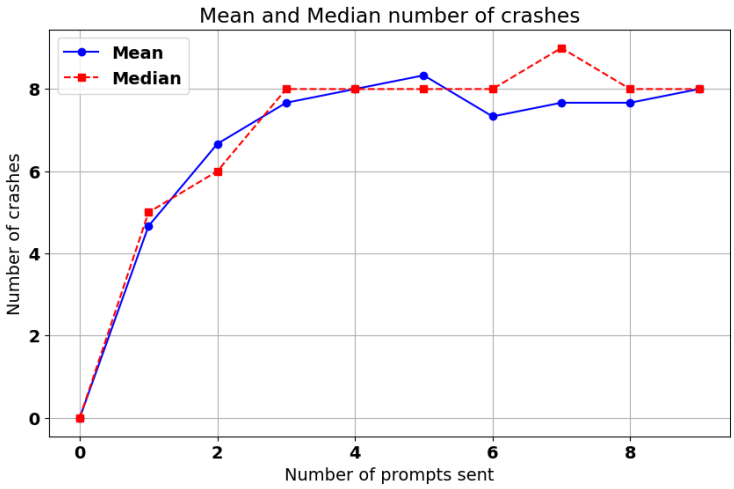

Figure 5: Mean and median values of number of crashes for the three groups of classes. The X-axis represents the number of prompts sent to GPT-3.5, while the Y-axis showcases the number of crashes

Table 3: Comparisons of mutation score distributions between iterations using Welch's t-test. The second row showcases the value of Cohen's d, while the third row presents the p-value. Each column represents the comparison between distributions: the i^{th} column shows the comparison between the initial distribution (iteration 0) and the distribution obtained from the i^{th} iteration

Comparison	0-1	0-2	0-3	0-4	0-5	0-6	0-7	0-8	0-9
Cohen's d	-2.3210	-0.9465	-0.9780	0.5074	-1.0000	2.0714	2.6790	3.6275	1.5903
P-value	0.0348	0.3686	0.3658	0.6334	0.3739	0.0771	0.0366	0.0110	0.1726

Figure 5. Table 3 showcases comparisons between the mutation score distributions. Once again, we have tried to verify whether the means of these distributions are significantly different, and performed Welch's t-test accordingly.

We have noticed that the LLM removed some test cases for the first iterations, as exemplified in Figure 4, where we observe that fewer mutants have been killed. Even though the highest median value was achieved by sending 8 prompts, the reliability of the code is very similar to other iterations (as the program crashes 8 out of 10 times).

Once again, for Welch's t-test, we selected a significance threshold (α) of 0.05. We observed that for every iteration between the second and sixth, as well as the ninth, the p-value is greater than α; for them, we fail to reject the null hypothesis that the distributions share the same mean. This is also indicated by the effect size being closer to 0.

The original three questions can be answered according to these plots. The static approach would not yield a better mutation score, as fewer mutants would be detected. We can improve the mutation coverage with the dynamic strategy by sending at least 6 prompts. Lastly, we observe that while the means obtained for the eighth and ninth iterations are very similar, the median is higher by sending only 8 prompts. Thus, to maximize the mutation score, we generally need to perform 8 iterations of our algorithm.

5.3 Discussion

In the case of the isolated classes, we observe a common pattern between the second and eighth iterations: the LLM initially provides wrong test cases, in the following prompt we enumerate the errors present, and in the subsequent response the test cases are fixed. This is the reason why we see an upward trend, followed by a downward one in Table 2. Initially, it may seem that the LLM removes some test cases, as emphasized by the downward trends. However, generally, this is not the case: with each response that has a working code, we see an increase in mutation scores. For the odd-numbered prompts - where the downward trends are usually noticed - we have fewer working test suites. Because more test suites may result in a higher mutation score overall, we notice the upward trends with the even-numbered prompts; this is also exemplified by Figures 2 and 3. An exception is a particular run of the ByteVector class, where we see a very high number of mutants killed at the first, sixth, and seventh iterations, as showcased by the outliers. While this occurred only once, it could be the case that the LLM can perform better on larger classes, as a greater number of mutants can be killed. Nonetheless, as more and more prompts are sent, we see a convergence at the ninth prompt, where both the mutation score is maximized, and the number of crashes is not as high as for the previous prompts. This is also highlighted in Table 2, by the highest effect size. Given that, on average, 7 mutants are eliminated when 9 prompts are sent, this results in a 23% increase in mutation coverage, primarily because most classes contain between 24 and 34 mutants.

Another notable aspect is that crashing test suites typically encounter errors at run-time rather than compile-time, often derived from incorrect assertion values. These assertions are generally trivial to correct, and in most instances, doing so leads to an improved mutation score. Hence, even when encountering a crashing test suite through automated generation, manual intervention at the final stage can increase the mutation score even further.

The results are worse for the other classes as GPT-3.5 is not able to properly understand the dependencies, this being showcased by the higher number of times when the code fails to run. When dependencies are involved, we also have compile-time errors regarding wrong lambda expressions, or when the LLM fails to explicitly specify the type parameter when calling a generic method. Interestingly, there have also been a few crashes where GPT-3.5 performed parentheses mismatch. Moreover, the number of run-time errors was also abundant, due to wrong assertions, but also null pointer and class cast exceptions. In contrast to the case with classes without dependencies, in this case, the errors are not as easy to fix. Additionally, when the errors are manually repaired, the mutation score is usually not improved.

From our results, we believe that ChatGPT-3.5 can be employed and successfully increase the mutation score when there are no dependencies involved.

6 Responsible Research

In Subsection 6.1, we will discuss how we handled any possible ethical issues. Matters related to legal considerations will be approached in Subsection 6.2, and Subsection 6.3 will showcase how we maintained scientific integrity throughout this project.

6.1 Ethical Considerations

Transparency should be a cornerstone of every research endeavor. We have prioritized this aspect while assessing the performance of the tests provided by GPT-3.5. We analyzed a multitude of Java classes, primarily focused on numerical and string manipulations. Therefore, none of the classes contain sensitive information. As no personal data was used, compliance with the General Data Protection Regulation (GDPR) is assured.

6.2 Legal Considerations

The Java classes employed in our research were taken from the SF110 and Apache Commons databases. Both of these databases are publicly available and consist of open-source software, ensuring that their use complies with relevant legal standards and licensing agreements. We have adhered to all licensing requirements associated with these resources, providing proper attribution where necessary. Furthermore, the results and data generated from this research were made available by the terms of these licenses, promoting transparency and collaboration within the research community.

6.3 Scientific Integrity

We are committed to maintaining the highest standards of scientific integrity throughout our research. All methodologies and results have been documented comprehensively to ensure reproducibility. We reported our findings honestly, including any limitations and negative results, to provide a complete and accurate representation of our research outcomes. By sharing every step of our approach, we facilitate further research and validation by other researchers in the field.

7 Threats to Validity

Validity is crucial in research to ensure that study results and interpretations are accurate and credible. *Construct validity* relates to how well a study measures the theoretical constructs it intends to assess, ensuring that the evaluation techniques align accurately with the intended concepts. *Internal validity* focuses on establishing a causal relationship between

variables within the study, minimizing the influence of confounding factors. *External validity* concerns the generalizability of study findings to other settings, addressing whether the results can be applied beyond the specific conditions of the study. Lastly, *conclusion validity* involves the accuracy of the inferences drawn from the study results, ensuring that conclusions are logically supported by the data and statistical analyses, and not overly influenced by chance or biases.

To mitigate potential threats to construct validity, we used weak mutants, a well-established metric for fault detection. For internal validity, we addressed GPT-3.5's potential bias towards certain classes by extracting classes from two repositories with different coding styles. To enhance external validity, we included classes of varying sizes and scopes to improve generalizability. For conclusion validity, we minimized the impact of GPT-3.5's non-determinism by performing 10 runs for each class or group of classes.

8 Conclusion and Future Work

Testing is vital for ensuring software correctness, and the automation of it leads to less time spent on this process. We have assessed GPT-3.5's performance in automatically augmenting test suites for Java classes. We have observed a difference between standalone classes and classes with dependencies: while GPT-3.5 can improve test cases for isolated classes, it struggles to understand dependencies. We have noticed that a higher number of prompts sent usually translates to a higher number of mutants detected, as generally, the ninth prompt maximized the mutation score in the case of single classes. As expected, the dynamic approach outperformed the static one in terms of mutation coverage.

In the future, we plan to perform a similar analysis for a dynamically typed programming language, such as Python. A comparison between the results obtained for Java and Python classes would showcase how much of an impact static typing has on the LLM's ability to produce reliable code. Moreover, performing a similar study using a higher-performing model, such as GPT-4 or GPT-4o, would allow us to assess the overall impact of LLMs more accurately.

References

[1] C. Klammer and A. Kern, "Writing unit tests: It's now or never!," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 1–4, 2015.

[2] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, pp. 1199–1218, 2018.

[3] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian, "A comprehensive overview of large language models," 2024.

[4] 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, p. 365–388, May 2020.

[5] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[6] 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*, FSE 2014, (New York, NY, USA), p. 654–665, Association for Computing Machinery, 2014.

[7] G. Jahangirova and V. Terragni, "Sbft tool competition 2023 - java test case generation track," in *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, pp. 61–64, 2023.

[8] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," vol. 24, no. 2, 2014.

[9] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pp. 263–272, 2017.

[10] G. Gay, "Improving the readability of generated tests using GPT-4 and ChatGPT code interpreter," in *Search-Based Software Engineering. SSBSE 2023* (P. Arcaini, T. Yue, and E. M. Fredericks, eds.), vol. 14415 of *Lecture Notes in Computer Science*, Springer, Cham, 2024.

[11] 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, 2024.

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

[13] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," 2024.

[14] Y. Tang, Z. Liu, Z. Zhou, and X. Luo, "Chatgpt vs sbst: A comparative assessment of unit test suite generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 6, pp. 1340–1359, 2024.

[15] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "Llm is like a box of chocolates: the non-determinism of chatgpt in code generation," 2023.

[16] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng, "No more manual tests? evaluating and improving chatgpt for unit test generation," 2024.

[17] M. L. Siddiq, J. C. S. Santos, R. H. Tanvir, N. Ulfat, F. A. Rifat, and V. C. Lopes, "Using large language models to generate junit tests: An empirical study," 2024.

[18] N. Alshahwan, J. Chheda, A. Finegenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and

E. Wang, “Automated unit test improvement using large language models at meta,” 2024.

[19] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, “Effective test generation using pre-trained large language models and mutation testing,” 2023.

[20] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, “Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 919–931, 2023.