The Effectiveness of GPT-4o for Generating Test Assertions

Adomas Bagdonas
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: Adomas Bagdonas
Final project course: CSE3000 Research Project
Thesis committee: Annibale Panichella, Mitchell Olsthoorn, Casper Poulsen

ABSTRACT

Over the last few years, Large Language Models have become remarkably popular in research and in daily use with GPT-4o being the most advanced model from OpenAI as of the publishing of this paper. We assessed its performance in unit test generation using mutation testing. 20 Java classes were selected from the SF110 Corpus of classes, and for each 10 different test classes were generated. After we resolved build errors and removed failing assertions, the evaluation using Pitest produced around 71% of mutation coverage on average on the sample dataset. Manually fixing the failing assertions increased the overall mutation score to 75%. Nonetheless, one of the main drawbacks was the need to manually resolve problems that the GPT-4o responses produced, such as code hallucination and incorrect assumptions about the classes under test.

KEYWORDS

mutation testing, GPT-4o, JUnit, OpenAI, EvoSuite, artificial intelligence, generative AI

1 INTRODUCTION

Software testing is a crucial part of software development that requires significant time and effort [5, 9]. It is one of the main options for developers to find mistakes in the code before shipping it to production. There are many different ways to test code each having different purposes, such as unit, integration, and system testing, out of which unit testing is one of the cheapest and most prevalent.

For a test case to be meaningful it must have at least a single assertion. After a test method code generates some output, they fail if there is a mismatch between expected and actual output, thus, indicating that something is not working as expected. The assertions are supposed to be written in a way that they detect if a developer has made a mistake in the implementation. Furthermore, effective assertions are a key factor in the quality of a test suite [26].

Although testing can provide valuable information, it is a complicated task. Writing good tests is not straightforward. There are numerous methods to evaluate the quality of a test suite, with line and branch coverage being some of the most popular. Nonetheless, research has shown that mutation testing is more indicative of test quality than the previously mentioned metrics [25], albeit less widespread [19]. All of these factors may contribute to lower-quality test cases.

There have been attempts to automate testing, for example, search-based unit test generation can produce test cases with strong mutation but is quite expensive [14]. Another problem is that they struggle with readability [20] which is important for maintainability of test suites. On the other hand, Large Language Models (LLMs) have demonstrated promising results working with natural language in related domains [6]. Therefore, they might overcome some of the limitations of search-based testing.

Currently, OpenAI is marketing GPT-4o as the most powerful state-of-the-art model they can offer. It will be used with a static approach, which entails a query with a single prompt rather than a dynamic approach, which aims to dynamically improve the results. As for the measurement of the results, mutation score will be utilized.

GPT-4o was used to generate test suites for 20 classes selected from the SF110 Corpus of Classes[1] [12]. One of the key reasons being that LLMs perform incredibly well on data they were trained on. Thus, for a proper evaluation, the code must be extracted from sources that have not been used to train the model. Each of the selected classes has 10 generated test classes to account for the non-determinism of LLMs. The results were compared with the performance of EvoSuite, a search-based software testing tool.

Even though the experiment results had plenty of trivial build errors and even more failing assertions, it produced notable results. The assertions successfully killed 71% of the mutants in the sample dataset. Furthermore, the tests exposed problems in the behavior of some of the classes under test. Manually fixing the assertions increased the score to 75%. However, more often than not it performed worse than EvoSuite.

The paper is structured as follows. Section 2 elaborates on the background along with related work. In Section 3, the study approach is described. The design of the experiment is detailed in Section 4. The results are presented and analysed in Section 5. In Section 6, potential threats to the validity of this research are discussed. Section 7 discusses the importance of responsible research in this domain. Lastly, the conclusions are presented in Section 8 together with future work.

2 BACKGROUND & RELATED WORK

2.1 Search-Based Software Testing

Search-Based Software Testing (SBST) is a research area that focus on automating automate test creation. Although there have been other attempts, SBST stands out among these as one of the most successful as shown by Software Based and Fuzz-Testing competitions [15]. Despite this, it cannot act as a test oracle, i.e. distinguish the correct behavior of a program from incorrect [4]. This necessitates human verification of the generated assertions. However, this process is complicated by the low readability of the tests generated using SBST [20]. One such SBST tool is EvoSuite which generates Java unit tests using a genetic algorithm [11].

2.2 Mutation testing

Mutation testing is a strategy for evaluating a unit test suite. The main idea behind it is to modify code under test and check whether the unit tests will fail. If any of the assertions in a test now fail, that means the mutant was killed. Hence, the resulting metric – mutation

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

ACM ISBN

score – is calculated by dividing the number of mutants killed by the total number of mutants. Additionally, research suggests that this metric is more insightful than line or branch coverages that are used more commonly [25].

There exist numerous tools that can determine the mutation score, for example, Pitest (PIT) is one of the most popular such frameworks [22] that is designed to evaluate Java unit tests.

2.3 Large Language Models

Over the past few years, Large Language Models (LLMs) have exploded in popularity, with OpenAI being one of the leaders in this field. These models have a plethora of real-world applications such as code generation. Currently, they play a vital role both in research and in daily use [7, 21]. On the other hand, one of their most notable flaws is hallucination. It happens when a model generates content that may appear plausible, but is factually incorrect.

The use of LLMs for testing is already discussed in the literature. It has been applied to generate test assertions [16] with readability being one of the main advantages compared to other test suite generation tools [13]. LLMs have performed reasonably well in generating test suites that aim for high code coverage [23]. Furthermore, some research has explored whether this applies to mutation testing, but only with older models [10]. Meanwhile, OpenAI has recently released GPT-4o which they claim to be their most powerful model as of yet.

3 APPROACH

The goal is to discover how well GPT-4o generates test assertions. We hand-picked a dataset of 20 Java classes. To account for the non-stochastic nature of LLMs, we generated 10 test classes for each of the dataset entries and evaluated them using Pitest. To judge the performance, we are comparing these results with 6 runs of EvoSuite. Figure 1 contains a schematic of this approach.

As LLMs are prone to code hallucinations, a portion of the generated assertions ends up failing. Therefore, we opted for two rounds of comparison, one with removing the failing assertions and the other with these assertions fixed manually.

For the test case generation, we came up with two possible approaches: static and dynamic. The static approach entails requesting the model for the response, whereas the dynamic approach means additional responses that aim to further iterate upon the results. For example, if some of the tests do not compile or are failing, another request can be sent to the LLM asking to fix the problem. Additionally, subsequent requests can be used to improve the test metric, which in this case is the mutation score. That being said, this paper focuses only on the static approach.

Regarding the test evaluation, mutation score will be used as it is proven to be a better metric for measuring the robustness of a test suite than line or branch coverage [25].

The model chosen for this experiment is OpenAI's flagship GPT-4o. As mentioned before, OpenAI claims that it is their most advanced model as of the writing of this paper. This claim is backed up by Large Model Systems Organization, which currently ranks it as the top LLM [8].

4 STUDY DESIGN

The question this paper aims to answer is *how effective is GPT-4o at generating test assertions with regards to mutation score?* Regarding the technologies used to implement the solution, we made the following choices. The programming language is Java due to its popularity, backward compatibility, and prevalence of mutation testing frameworks. To perform the evaluation we incorporated Pitest into a Gradle project that supported both JUnit 4 and JUnit 5.

4.1 Evaluation dataset

To evaluate the performance, 20 classes from the SF110 Corpus of Classes [12] were selected. The Corpus is a collection of 110 mostly stale open-source Java projects hosted on SourceForge. It was chosen because these projects are hosted on SourceForge rather than GitHub, thus, decreasing the likelihood that the code was in the training set of GPT-4o. That being said the Corpus contains a plethora of projects which all contain numerous classes, thus, there was a need to pick a smaller sample to base the experiment on. The classes were chosen using the following criteria:

(1) The classes should not depend on more than one other class from the same project not including the default Java classes. This is largely due to the fact that isolated classes are simpler and more suitable to test with unit tests. Furthermore, this criterion significantly simplifies the process of cherry-picking the classes.
(2) The classes should not be trivial to test. For example, testing a class with a single method having the cyclomatic complexity of 1 is unlikely to produce interesting results. Hence, we chose 5 as the cutoff point.
(3) The code should contain logic other than basic getters and setters. For instance, classes that perform string or number manipulation or parsers were welcome additions.

The selected classes are listed in Table 1. For each, the count of lines of code, McCabe's cyclomatic complexity, and the highest cyclomatic complexity were calculated using CK code metrics tool [1]. Meanwhile, the number of mutants was reported by Pitest.

Two of the classes in the dataset have other small classes as their dependencies, namely, *OpMatcher*, which relies on *NamedStyle*, and *Contract*, which uses *UnderComp*. There was a conscious decision not to provide the code on which the classes depended to glimpse if there was a drop in test quality and the mutation score.

4.2 Experiment

Regarding the experiment, for each of the 20 classes, 10 testing classes were generated using the OpenAI Python library [18] to account for randomness. The classes were grouped in 10 runs with each having a test class for the source code. The requests were sent as separate chats to the completions API endpoint with the sytem prompt: *You are a helpful assistant that can create robust test suites for Java classes. You will output code only.* Then the code was extracted from the responses and saved in the Gradle project. After all the classes were generated, every build error was manually resolved. Afterwards, all failing assertions were removed. If removing an assertion caused the test to have no remaining assertions, it was removed as well. Lastly, the mutation score of each class was measured using Pitest 1.15.0.

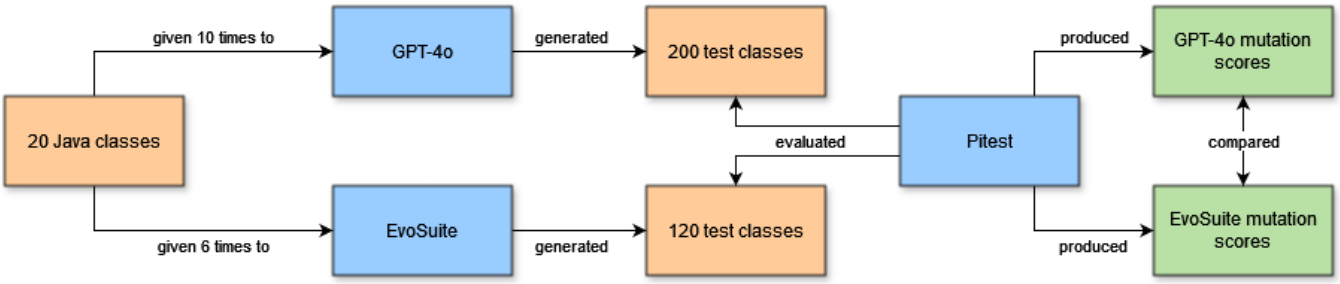

Figure 1: Illustration of the static approach

Table 1: Some code metrics of the classes under test. 'Dependency' column is 'Yes' when the class depends on another class within its project. CC stands for cyclomatic complexity.

Class under test	Lines of code	CC	Largest CC of a method	Method count	Dependency	Total mutants
saxpath.Axis	89	28	14	2	No	40
sfmis.Base64	103	22	9	9	No	95
javaviewcontrol.Base64Coder	81	35	17	7	No	94
jiprof.ByteVector	158	33	14	11	No	138
openjms.CommandLine	80	24	8	10	No	32
tulibee.Contract	90	26	23	4	Yes	36
javaviewcontrol.HtmlEncoder	40	14	14	1	No	18
imsmart.HTMLFilter	27	8	8	1	No	7
corina.NaturalSort	87	47	18	8	No	68
templateit.OpMatcher	90	27	8	5	Yes	29
biblestudy.Queue	113	28	7	13	No	34
corina.Sort	79	29	11	8	No	29
corina.StringComparator	20	7	6	2	No	16
fim1.StringEncoder64	139	38	10	9	No	177
corina.StringUtils	83	26	11	7	No	37
tulibee.Util	45	22	11	7	No	28
lagoon.Utils	69	25	11	5	No	24
schemaspy.Version	40	14	5	4	No	19
beanbin.WildcardSearch	42	11	10	2	No	24
battlecry.bcWord	76	27	10	9	No	39

For this part of the experiment, Java 17 was used for this experiment along with Gradle 8.0. Other dependencies along with their versions can be found in the replication package of this project [3].

4.3 Baseline

The results of this experiment are measured against mutation scores obtained by running EvoSuite. The different methods will be compared by calculating a Wilcoxon rank-sum test along with the Vargha-Delaney effect size for each of the classes separately [2, 24]. EvoSuite was run 6 times with a time budget of 120 seconds with the following setup: 2x AMD EPYC 7H12 64-Core Processor with 128 cores and 512 threads with hyperthreading. The CPU frequencies range between 1.5 GHz – 2.6 GHz. The main memory contains 256GB of RAM and the OS is Ubuntu 22.04.

5 RESULTS

5.1 Initial results

Even though the prompt asked for code only, 19 responses (less than 1%) contained other explanation text despite the explicit instructions to output code only. In the 200 classes generated by the experiment, there were a total of 38 build errors. The majority of errors were recurring: 13 were caused by missing imports from Java utility classes, and 10 were caused by the omission of the package name. Since the model was not explicitly informed about the APIs of the classes that were used as dependencies (*NamedStyle* and *UnderComp*), 7 of the build errors were caused by hallucinated calls to non-existent methods in the other classes used by the class under test. After the failing assertions were removed, 1580 tests were remaining, which on average is 7.9 tests per class. Collectively, they killed 71% of all the mutants. In terms of readability, the test suites seemed as though they were written by an actual person.

Table 2: Experiment results rounded to the nearest two decimal places. Classes with dependencies are highlighted in yellow.

Class under test	GPT-4o		EvoSuite		Wilcoxon rank-sum test		Vargha-Delaney A
	Mean score, %	σ	Mean score, %	σ	W-statistic	p-value	
saxpath.Axis	100	0	100	0	0	1	0.5
sfmis.Base64	75.6	13.59	76.33	4.27	-0.22	0.83	0.47
javaviewcontrol.Base64Coder	90.3	1.68	94.33	1.7	-2.82	0	0.07
jiprof.ByteVector	29.4	4.18	32.17	1.57	-1.41	0.16	0.28
openjms.CommandLine	83.1	3.73	87.17	2.48	-1.9	0.06	0.21
tulibee.Contract	67.4	8.62	99	1.41	-3.25	0	0
javaviewcontrol.HtmlEncoder	76.2	2.75	72.5	5.5	1.14	0.25	0.68
imsmart.HTMLFilter	100	0	100	0	0	1	0.5
corina.NaturalSort	71.8	8.63	70.33	6.34	-0.11	0.91	0.48
templateit.OpMatcher	84.2	3.66	72	0	3.25	0	1
biblestudy.Queue	86.3	5.33	80	3.32	2.22	0.03	0.84
corina.Sort	54.7	12.46	29	0	3.25	0	1
corina.StringComparator	92.7	6.63	100	0	-1.95	0.05	0.2
fim1.StringEncoder64	72.4	7.58	77.5	6.32	-1.19	0.23	0.32
corina.StringUtils	81.3	0.9	87.17	1.95	-3.2	0	0.01
tulibee.Util	89	10.18	100	0	-2.28	0.02	0.15
lagoon.Utils	74.5	9.63	91.67	6.13	-2.82	0	0.07
schemaspy.Version	80	3	84	0	-2.28	0.02	0.15
beanbin.WildcardSearch	88	7.24	96	0	-1.95	0.05	0.2
battlecry.bcWord	48.4	27.61	78.5	1.8	-1.95	0.05	0.2
Averages	77.27	6.89	81.38	2.14	-0.87	0.23	0.37

Table 3: Results after fixing the failing GPT-4o test cases manually to the nearest two decimal places. Classes with dependencies are highlighted in yellow.

Class under test	GPT-4o		EvoSuite		Wilcoxon rank-sum test		Vargha-Delaney A
	Mean score, %	σ	Mean score, %	σ	W-statistic	p-value	
saxpath.Axis	100	0	100	0	0	1	0.5
sfmis.Base64	83.1	7.67	76.33	4.27	1.03	0.3	0.66
javaviewcontrol.Base64Coder	90.5	1.63	94.33	1.7	-2.77	0.01	0.07
jiprof.ByteVector	30.4	3.38	32.17	1.57	-0.98	0.33	0.35
openjms.CommandLine	84.4	3.41	87.17	2.48	-1.41	0.16	0.28
tulibee.Contract	68.5	8.43	99	1.41	-3.25	0	0
javaviewcontrol.HtmlEncoder	76.2	2.75	72.5	5.5	1.14	0.25	0.68
imsmart.HTMLFilter	100	0	100	0	0	1	0.5
corina.NaturalSort	73.20	7.69	70.33	6.34	0.33	0.74	0.55
templateit.OpMatcher	98.2	1.47	72	0	3.25	0	1
biblestudy.Queue	87.4	3.5	80	3.32	2.82	0	0.93
corina.Sort	59.5	11.36	29	0	3.25	0	1
corina.StringComparator	95.1	6.11	100	0	-1.63	0.1	0.25
fim1.StringEncoder64	73.7	6.36	77.5	6.32	-0.87	0.39	0.37
corina.StringUtils	89	4.65	87.17	1.95	1.36	0.18	0.71
tulibee.Util	89	10.18	100	0	-2.28	0.02	0.15
lagoon.Utils	77.8	4.75	91.67	6.13	-2.82	0	0.07
schemaspy.Version	80	3	84	0	-2.28	0.02	0.15
beanbin.WildcardSearch	90.6	6.56	96	0	-1.63	0.1	0.25
battlecry.bcWord	79.5	15.09	78.5	1.8	0	1	0.5
Averages	81.3	5.4	81.38	2.14	-0.33	0.28	0.45

Table 4: Run durations

Run	Duration
1	4m 17s
2	4m 37s
3	4m 17s
4	4m 44s
5	5m 12s
6	5m 17s
7	5m 2s
8	4m 44s
9	4m 39s
10	4m 25s

The methods were named appropriately to the test cases, the variable names were meaningful, and sometimes there were comments explaining what was being done or other useful information. Additionally, there were even cases when the LLM made comments that proposed to adjust the assertion based on the actual expected value, recognising that it might fail.

Moving on to the analysis, Table 2 provides an overview of the results. A negative W-statistic implies that GPT-4o produced worse results than EvoSuite and a positive means the opposite is true. For 9 of the 20 classes Wilcoxon rank-sum test with p-value = 0.05 results in a statistically significant difference in favor of the EvoSuite. In 3 of the cases, the same test concludes that GPT-4o performed better. As for the Vargha-Delaney A measure, values below 0.5 show that EvoSuite showcased better results, 0.5 means both methods were equally good, and values above 0.5 indicate that GPT-4o was more effective.

Based on the findings obtained by analysing the data, GPT-4o with failing assertions removed performs slightly worse than EvoSuite. Although it did so with a smaller time budget: each run (which consists of generating a single test class for each entry in the dataset) took around 4 minutes and 44 seconds on average as depicted in Table 4. This is also considering an exponential backoff in case there was a rate limit error (too many tokens used per minute) [17]. However, it is worth noting that a timewise comparison might not be fair as GPT-4o requires large servers to run whereas EvoSuite can be run on personal computers. That said, the generated code was processed manually, which entailed finding and removing failing assertions and took way longer than generating all the replies. One of the possible reasons for that was our initial lack of familiarity with the classes under test.

5.2 Results after fixing failing assertions

The results improved after the tests with failing assertions and build errors were manually fixed. Except for the few assertions that were correct and detected unexpected behavior of the class under test – they were left out. Thus, a total of 225 unit tests were rewritten. As a consequence, the overall coverage increased to 75%.

More detailed results can be found in Table 3. Fixing the assertions improved the mutation scores, thus, changing the results of the statistical tests. This time GPT-4o outperforms EvoSuite with significant confidence in 3 cases. However, EvoSuite showcases better performance for 5 of the classes.

Even though for some classes there was a strong increase in mutation coverage, there were also examples of minimal or no improvement. The values can be found in Figure 2. We observed that for the classes that saw the greatest rise in mutation score, GPT-4o consistently failed to understand or correctly predict their usage based on the source file contents. Notable examples were *bcWord*, *NaturalSort*, *Utils* and alternate base64 encoding schemes in *Base64*:

- In the case of *bcWord*, the logic deals with language, phonemes, and rhyming schemes. It requires strings in a precise format to work properly. There were no examples of how the class was used, and the intended use is not intuitive to understand if only the source code is given. Thus, different runs resulted in mutation coverage as low as 21% when the model made a lot of wrong assumptions that resulted in failing assertions. Despite this, it scored 95% during the most successful run. Hence, the standard deviation for the model's results for this class was the highest and fixing the failing assertions substantially increased its mutation score.
- *OpMatcher* test in total had 24 assertions that failed because of unknown dependency API. The main problem was that GPT-4o based most of the assertions on the *assertEquals* assertions, but the class *NamedStyle* did not have an *equals* method implemented. Therefore, under the hood, the tests were just comparing different memory locations rather than the features of the objects.
- *Utils* had 58 failing assertions because of incorrectly predicted output. The class works with string manipulation. It often had custom rules for converting each character of a given string to some substring of the output string. The model seemingly had a lot of trouble with that as it could not adapt well to this exact rule set.
- *Base64*, a class that works with base64 encoding and decoding, offered an alternative encoding scheme that used different symbols. Moreover, nearly every internal field and function name had obscure names, which could have played a part in confusing GPT-4o. In 9 out of 10 runs, the LLM failed to assert the output of decoding or encoding strings. It would either assume the standard character set or make a poor attempt to use the alternative set, leading to the majority of alt scheme tests failing. In the one case where the class suceeded to work with the alt scheme, there was only a single test method that simply encoded and decoded a string and compared with the value it started with.

Therefore, some of the analysed examples indicate that GPT-4o tends to sometimes make incorrect assumptions about the code that lead to failing assertions. Removing them significantly reduced the percentage of mutants killed in some of the classes. This would also explain the greater standard deviation compared with EvoSuite.

5.3 Examples of possible bugs or corner cases found by GPT-4o

It is also worth noting that a failing assertion is not necessarily a bad assertion. It could also mean that the test suite detected a bug

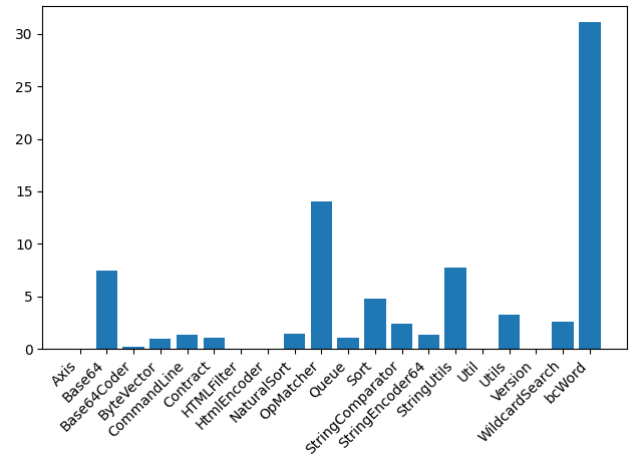

Figure 2: Increase in mean mutation scores after fixing the failing assertions

or a corner case that the developer missed. A small fraction of the failing assertions were caused by this exact reason.

All of the following bugs were caught by failing tests. When fixing the failing tests, these were purposefully kept out of the test suite as that would imply accepting the unintentionally incorrect behavior of the program. There have been cases when the intended output was not clear but after a deeper dive into the purpose and usages of the code, those instances were deemed correct or given the benefit of the doubt. As for the remaining few, some of the noteworthy examples will be discussed in the following paragraphs.

One of the simpler bugs found by the failing assertions is in *OpMatcher*. Even though at first glance *matchTemplateEnd* can handle *null* input, calling it with *null* causes an exception to be thrown. The reason for that is missing parentheses in the three-part boolean statement. Even if the first part is *false*, the third part is still evaluated because the logical OR expression can still return *true*. Thus, the program calls the *indexOf* method on a value that is *null*, leading to an exception. This bug was caught in 4 out of 10 *OpMatcherTest* classes.

```
1  public static boolean matchTemplateEnd(String
       text)
2  {
3      return text != null && (text.indexOf("
           @template_end") != -1)
4              || (text.indexOf("@tend") != -1);
5  }
```

WildcardSearch is also demonstrating erroneous behavior on some of the inputs. The class functions as a Regex checker for text with wildcards. Although the class appears to be functioning correctly with wildcard symbols, unexpected results could be achieved by using it without them. The test below manages to point out two bugs with two assertions. The first one fails because *doesMatch* returns *true* for all strings that start with the search string ("hello" in this case). Even though a correct regex checker would return *false*. As for the second one, an input string shorter than a search string produces an exception. In all fairness, these can also be considered corner cases because the project from which this class was taken has only uses this class with a wildcard symbol.

```
1      @Test
2      void testDoesMatch_noMatchDifferentLengths() {
3          WildcardSearch search = new WildcardSearch
               ("hello");
4          assertFalse(search.doesMatch("helloo"));
5          assertFalse(search.doesMatch("hell"));
6      }
```

Yet another example can be found in *Queue*. Calling the dequeue method on an empty queue causes the variable *numItems* to be negative. Line 15, where the count of elements is decreased, gets executed even if the queue is empty.

```
1  public synchronized Object dequeue() {
2    Object obj = null;
3
4    if(isEmpty()) {
5      System.out.println("Cannot remove when queue
           is empty");
6
7    } else if(first == last) { // first see if we
         only have one item in the queue
8      obj = first.value;
9      first = null;
10     last = null;
11   } else {
12     obj = first.value;
13     first = first.next;
14   }
15   numItems--;
16   return obj;
17 }
```

6 THREATS TO VALIDITY

6.1 External validity

The dataset size is relatively small. The sampled classes are strongly biased towards stale projects written using older versions of Java. As a result, we do not know if using other programming languages or using different datasets would alter the findings. To account for that we gathered classes whose total cyclomatic complexity ranges between 7 and 47 and the number of mutants varies from 7 to 177. In addition, we sampled classes with different purposes, such as base64 encoding, sorting, string manipulation, data structures, etc. Overall, the samples are from 15 different projects. This way the results included more diversity which should help with generalisability.

6.2 Internal validity

Manual work was used to first remove and then fix the failing assertions. Therefore, if others replicate the same experiment, they might obtain different results as they could have a different interpretation of what the LLM was trying to do. Even though for the majority of failing test cases fixing the expected value was sufficient, the efforts to curb this problem also included inferring what the model aimed to achieve using the test method names, comments, the actions it took, and the source code. Some of the more complex cases required

analyzing the context and usages of the class within its respective project. On the other hand, if it appeared that the program had a bug, the test case would be left out as testing for incorrect behavior defeats the purpose of testing.

6.3 Construct validity

Another threat is that the LLMs have enormous training sets. Since the projects used for this paper are open-source, there is a considerable likelihood that at least some were used in the training set of GPT-4o. Especially given the possibility that OpenAI trained the model even on private GitHub repositories. To mitigate this, we chose the SF110 Corpus of Classes which is not hosted on GitHub (even though some of its projects are).

6.4 Conclusion validity

One of the main problems that research with LLMs might face is non-determinism. The same prompt is highly unlikely to produce the same results. There has also been research claiming that OpenAI continuously adjusts their models which might affect performance [21], which they acknowledge in their documentation [17]. To account for this, we prompted the LLM 10 times for each of the classes within the dataset before performing statistical analysis and provide the metadata of the queries in the following section.

7 RESPONSIBLE RESEARCH

The generated test code may contain elements from private codebases. Thus, blindly following the described approach might lead to the model reciting copyrighted code. Therefore, developers using LLMs to generate code should verify the code before using it in other projects. In addition, datasets often tend to have biases. Artificial intelligence models trained on such data tend to display biased behavior. Despite OpenAI's efforts to curb it as much as possible, it is impossible to fully remove them.

To ensure reproducibility, the project used to run the experiment is published online along with the results, including the raw response messages, extracted code, and code coverage reports [3]. Nevertheless, this does not guarantee replicability as LLMs are not deterministic. Thus, repeating the same experiment with the same setup might yield different results. The experiment described in the paper is run on version *gpt-4o-2024-05-13*, the first public version of GPT-4o. Since the models are constantly fine-tuned, it is recommended to mention the date of the queries as well [21], which in this case was the 28th of May, 2024.

8 CONCLUSION & FUTURE WORK

In this paper, we took a static approach to generate JUnit tests using GPT-4o. The results were evaluated with regards to mutation score using Pitest and compared against EvoSuite. Our approach performed slightly worse than EvoSuite in terms of mutation score but improved upon some of the limitations of SBST, such as poor readability, the time it takes to generate a test class, and the assumption that the class under test is behaving correctly.

GPT-4o appears to be a viable option for creating unit tests. However, the generated code has to be verified and often contains hallucinations or other mistakes. The quality of the results is influenced by how well the model understands the class under test.

Despite the shortcomings, we demonstrated a viable strategy that could aid a developer in building a robust test suite. Furthermore, the results have shown that the model might find a different perspective and discover corner cases not yet considered by the developer.

As for future work, multiple directions could be insightful. For example, analyzing different programming languages that have mutation testing frameworks such as Python, JavaScript, and PHP. In addition, there are numerous other LLMs whose performance might rival GPT-4o. It might be worthwhile to compare them against each other or test out different prompts. Moreover, the current approach can be applied to different datasets to obtain a more diverse set of results. Alternatively, the dynamic approach can be used to automatically further improve tests after their generation.

ACKNOWLEDGMENTS

To my supervisor Mitchell and responsible professor Annibale for their insights and support. To my project teammates Arda, Reiner, Roelof, and Stefan for forming a collaborative and interactive environment.

REFERENCES

[1] Maurício Aniche. 2015. *Java code metrics calculator (CK).* Available in https://github.com/mauricioaniche/ck/.

[2] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250. https://doi.org/10.1002/stvr.1486

[3] Adomas Bagdonas. 2024. GPT-4Test. https://github.com/AdoBag/GPT-4Test. Accessed: 2024-06-23.

[4] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[5] Frederick P. Brooks. 1995. *The mythical man-month (anniversary ed.).* Addison-Wesley Longman Publishing Co., Inc., USA.

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165

[7] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. 2024. A Survey on Evaluation of Large Language Models. *ACM Trans. Intell. Syst. Technol.* 15, 3, Article 39 (2024), 45 pages. https://doi.org/10.1145/3641289

[8] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E. Gonzalez, and Ion Stoica. 2024. Chatbot Arena: An Open Platform for Evaluating LLMs by Human Preference. arXiv:2403.04132 [cs.AI]

[9] Ermira Daka and Gordon Fraser. 2014. A Survey on Unit Testing Practices and Problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 201–211. https://doi.org/10.1109/ISSRE.2014.11

[10] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. 2024. Effective test generation using pre-trained Large Language Models and mutation testing. *Information and Software Technology* 171 (2024), 107468. https://doi.org/10.1016/j.infsof.2024.107468

[11] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179

[12] Gordon Fraser and Andrea Arcuri. 2014. A Large Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 8.

[13] Gregory Gay. 2024. Improving the Readability of Generated Tests Using GPT-4 and ChatGPT Code Interpreter. In *Search-Based Software Engineering*. Springer Nature Switzerland, 140–146. https://doi.org/10.1007/978-3-031-48796-5_11

[14] Mark Harman, Yue Jia, and William B. Langdon. 2011. Strong higher order mutation-based test data generation *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 212–222. https://doi.org/10.1145/2025113.2025144

[15] Gunel Jahangirova and Valerio Terragni. 2023. SBFT Tool Competition 2023 - Java Test Case Generation Track. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. 61–64. https://doi.org/10.1109/SBFT59156.2023.00025

[16] Kefan Li and Yuan Yuan. 2024. Large Language Models as Test Case Generators: Performance Evaluation and Enhancement. (2024). https://doi.org/10.48550/arXiv.2404.13340

[17] OpenAI. 2024. OpenAI API Documentation. https://platform.openai.com/docs/ Accessed: 2024-06-22.

[18] OpenAI. 2024. OpenAI Python Library. https://github.com/openai/openai-python Accessed: 2024-06-22.

[19] Ali Parsai and Serge Demeyer. 2020. Comparing mutation coverage against branch coverage in an industrial setting. *Int J Softw Tools Technol Transfer* 20 (2020), 365–388. https://doi.org/10.1007/s10009-020-00567-y

[20] Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, Danielle Gonzalez, and Mehdi Mirakhorli. 2021. DeepTC-enhancer: improving the readability of automatically generated tests. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 287–298. https://doi.org/10.1145/3324884.3416622

[21] June Sallou, Thomas Durieux, and Annibale Panichella. 2024. Breaking the Silence: the Threats of Using LLMs in Software Engineering. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER'24)*. ACM. https://doi.org/10.1145/3639476.3639764

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

[23] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105. https://doi.org/10.1109/TSE.2023.3334955

[24] András Vargha and Harold D. Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. https://doi.org/10.2307/1165329

[25] Peng Zhang, Yang Wang, Xutong Liu, Yibiao Yang, Yanhui Li, Lin Chen, Ziyuan Wang, Chang ai Sun, and Yuming Zhou. 2022. Test suite effectiveness metric evaluation: what do we know and what should we do? arXiv:2204.09165

[26] Yucheng Zhang and Ali Mesbah. 2015. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 214–224. https://doi.org/10.1145/2786805.2786858