Evaluating the Effectiveness of Meta Llama3 70B for Unit Test Generation

Reinier Schep

Supervisor(s): Annibale Panichella, Mitchell Olsthoorn

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

Evaluating the Effectiveness of Meta Llama3 70B for Unit Test Generation

Reinier Schep
Delft University of Technology
Delft, The Netherlands

Abstract

The automated generation of test suites is crucial for enhancing software quality and efficiency. Manually writing tests is time-consuming and accounts for about 15% of project time while tests generated by automated tools like EvoSuite and Pynguin often lack readability and comprehensibility. Recent research suggests that Large Language Models (LLMs) might offer a promising alternative. This paper investigates the effectiveness of Llama3 70B in generating unit test cases for Java and Python projects. We compared Llama3 against EvoSuite and Pynguin by measuring the mutation score of test suites generated for a corpus of 20 Java and 20 Python classes. Our findings reveal that EvoSuite significantly outperforms Llama3 in terms of mutation score, achieving an average mutation score of 81.05% versus Llama3's 66.26%. Conversely, Llama3 surpasses Pynguin, with scores of 51.95% and 42.73% respectively. These results highlight that while Llama3 is not superior to EvoSuite, it shows potential as a viable tool for test generation, especially for dynamically typed languages like Python. Further, empirical observations indicate that Llama3 requires significantly more time to generate tests compared to EvoSuite and Pynguin. This study underscores the need for continued research to optimize LLMs for software testing and improve their efficiency and accuracy.

Keywords

Automated Test Suite Generation, Software Quality, Large Language Models (LLMs), Llama3 70B, Unit Test Cases, Java, Python, Mutation Testing, EvoSuite, Pynguin

1 Introduction

The automated generation of test suites plays a crucial role in software development, facilitating the detection of bugs and improving code quality. Tests can be manually written but this is time intensive in general [7] and takes up about 15% of the project on average [3]. Search Based Software Testing (SBST) tools are meant to alleviate this by automatically generating tests, but they are often not used in projects because they are less readable and meaningful than manually written tests [1]. In other words, humans are struggling to comprehend what behaviour these automated tests are asserting.

Reinier Schep, June 23, 2024, Delft

This paper has examined how effective the "meta/meta-llama-3-70b-instruct"[1] model is at generating unit test cases. The effectiveness of Llama3 has been measured by acquiring a Java and Python corpus of 20 classes each and letting Llama3 generate 12 test suites for each class. Then for the Java corpus, EvoSuite is used to generate corresponding test suites and Pynguin fulfills the same role for the Python corpus. Then for each test suite, we measured its quality by calculating the mutation score. Then, this resulted in having 12 mutation scores per class per tool. Then, we compared the different tools using statistical analysis, namely the Wilcoxon signed-rank test[2]. It is used to determine if these samples come from a significantly different distribution and the Vargha-Delaney $\hat{A}_{12}$ standarized effect size is used to measure how big this difference is. The Java corpus is obtained from SF110[3] and the Python corpus is a subset of the corpus used in a study by Stephan Lukasczyk and Gordon Fraser [10] which also involved Pynguin. The EvoSuite tool is run for 120s per class per test suite since research by Panichella et al. [14] supports the hypothesis that this is sufficient. Pynguin is run for 30s per class per test suite due to time constraints under the hypothesis that this is sufficient time, a small experiment in the result section confirms this. A large scale study by Panichella et al. [15] and another study by Stephan Lukasczyk et al. [12] found that DynaMOSA was the best performing search algorithm for EvoSuite, therefore both EvoSuite and Pynguin will be run with this search algorithm. Llama3 is used to generate 8 passing test cases per class per test suite.

Our results show that EvoSuite significantly outperforms Llama3 for 13 out of the 20 Java classes and Llama3 outperforms EvoSuite for only 3 out of 20 Java classes. Also, if the average over all Java classes and all 12 independent runs is considered, EvoSuite attains an average mutation score of 81.05% and Llama3 attains 66.26%. The average difference in mutation score between the two is thus 14.79%. For the majority of Java classes, 8 test cases is sufficient as they have converged in mutation score after 8 tests. Our results also show that Pynguin outperforms Llama3 for 4 out of the 20 Python classes and Llama3 outperforms Pynguin for 9 out of 20 Python classes. For the majority of Python classes, 8 test cases are sufficient as mutation score has converged after 8 tests. Our experiment has shown that running Pynguin for 30s is not worse than running it for 90s in terms of mutation score, and both outperform Pynguin when it runs for 60s. The average mutation score for Pynguin over all classes and over all 12 runs was 42.73%. Llama3 achieved an average mutation score of 51.95%, thus a difference of roughly 9.22%.

These results back up the hypothesis that EvoSuite is better than Llama3 at generating test suites with regard to mutation score. It

[1] https://github.com/meta-llama/llama3
[2] https://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test
[3] https://www.evosuite.org/experimental-data/sf110/

also supports the hypothesis that it is easier to generate test suites with high mutation score for statically typed languages rather than dynamically typed ones since Llama3 achieved a higher mutation score for the Java corpus than for the Python corpus. Thus, Llama3 is not more effective than EvoSuite at generating unit tests with regards to mutation score, but it does outperform Pynguin.

One limitation of generating unit tests with Llama3 is that it takes significantly longer compared to EvoSuite and Pynguin. In the best case, Llama3 will take around a minute to generate 8 passing unit tests for some class, but in a bad case this process could take 10 minutes or more.

The rest of the paper is structured as follows. Directly after this section follows section 2 which explains the background of this research and discusses related work. Section 3 contains the approach which outlines how the main research question will be answered, it also explains certain decisions in the research process. Thereafter, section 4 specifies the setup of the experiment based on the approach chosen in section 3 and it is followed up by section 5 which shows the results of this experiment. Section 6 then discusses the ethical considerations of this research. This is followed by section 7 which reasons about the results, the threats to the validity of the research as well as the limitations of the approach used. Then, section 8 concludes the research and gives a definitive answer to the research question. Finally, section 9 shares ideas for future research in this field.

2 Background & related work

2.1 Mutation testing

Mutation testing is a way of measuring the quality of a test suite by calculating the mutation score. It involves injecting artificial faults (mutations) in the source code of the program for which the test suite is made and them measure how many of these remain undetected when running the test suite.

2.2 Use of LLMs in testing

This section discusses multiple papers which suggest that LLMs are a promising way of generating tests. Research by Tang et al. [20] have compared ChatGPT with EvoSuite for unit test generation for Java projects. They found that EvoSuite outperforms ChatGPT by 18.8% for code coverage and by 5% for bug detection. Ryan et al. [17] has used the 'SymPrompt' approach to provide guidance to LLMs for test generation. The SymPrompt approach entails that for each method in a class under test, and then for each execution path possible in that method, they construct a prompt which asks the LLM to generate a test for that execution path. They found that code coverage improved by a factor of 2x over baseline prompting strategies when applied to GPT-4 and to Python projects. Liu et al. [9] used the 'AID' approach which consists of using an LLM and differential testing. They found that the recall, precision, and F1 score of AID outperforms the best existing method by up to 1.80×, 2.65×, and 1.66× respectively. Yuan at el. [22] has investigated ChatGPT's capability to generate unit tests with regard to correctness, sufficiency, readability, and usability. They found that some tests had compilation errors and assertion errors, but the ones that passed had comparable coverage to manually written tests. Consequently, they concluded that it's a promising way of generating tests if the correctness would be improved. Wang et al. [21] found that current techniques for automated unit test generation (excluding LLM techniques since those are largely unexplored) are far from satisfactory. In their paper, they analysed 102 other papers which used LLMs for software testing.

According to Johnsson [5], the Llama2 7B model is useful for generating tests, this makes the Llama3 70B model an interesting LLM to research. To the best of our knowledge, no prior research has investigated Llama3 70B with regard to test generation. This paper will try to fill that knowledge gap.

3 Approach

This study aims to evaluate the effectiveness of Llama3[4] at generating unit tests. To determine the effectiveness of Llama3, we need to quantify the performance of the test suites it generates. Various metrics exist to measure this, some examples include: code coverage, mutation score and test execution time. Out of these metrics, mutation score is the best metric since it directly measures the test suite's ability to catch bugs in the program under test. Having high code coverage is good, but technically one could have 100% code coverage without being able to catch a single bug if bad assertions are made, with 100% mutation score this cannot happen. A high mutation score gives high confidence that good assertions are made in the test suite.

Now we are able to quantify the performance of the test suites generated by Llama3 but to determine how effective it is it should be compared against a benchmark. One approach would be to take a large sample of classes with high cyclomatic complexity and corresponding manually written test cases and compare those (in terms of mutation score) with the ones the Llama3 generates. However, finding classes which adhere to both requirements might be hard which would reduce the available classes for this research and that could hurt the validity of this research. Another approach is to use a state-of-the-art SBST tool to generate a test suite for some arbitrary class. These automatically generated test suites are also often more effective than manually written ones and have fewer test smells [16]. This widens the possible classes that can be chosen for this research which potentially improves the validity of this research. A class with high cyclomatic complexity and without manually written test cases can now be included. Now for those classes we can generate a test suite by the SBST tool and a test suite generated by Llama3 and compare mutation scores of these test suites to determine which one is more effective.

The generation of tests by both Llama3 and SBST tools is a non-deterministic process. This means that the non-deterministic process of generating tests should be repeated multiple times with the calculation of their corresponding mutation scores. Then statistical tests can be employed to judge if the two samples of mutation scores come from the same distribution or not. A total of 12 test suites per class with corresponding mutation score for both Llama3 and the SBST tool will be generated and then the Wilcoxon signed-rank test[5] will be used to determine the probability of observing both samples under the null hypothesis that they were drawn from

[4] https://github.com/meta-llama/Llama3
[5] https://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test

the same distribution. In total, there are 12 mutation scores per class for both Llama3 and the SBST tool. The Wilcoxon test is preferred over other methods since it does not assume normality of data and because it has been used in related papers [2] [19]. As recommended by existing guidelines in empirical software engineering, if this probability (p-value) is less than 5% ($\alpha = 0.05$) we will reject the null hypothesis with at least 95% confidence. If a significant difference in distribution for a certain class is observed, the Vargha-Delaney $\hat{A}12$ standarized effect size will be used to quantify how large this difference is which is also used in related papers [13] [2].

Obviously, at least one programming language which has a SBST tool available needs to be included in this research. Including two programming languages would be even better since results can be compared across languages. Now it would also be interesting to pick two languages which differ from each other in some fundamental aspect, i.e. functional vs. imperative or statically typed vs dynamically typed. Java is a good language to include because it has the best performing SBST tool available, namely EvoSuite. A paper by Panichella et al. [14] found that EvoSuite scored best in a unit testing competition with other SBST tools. Java is statically typed and imperative, so now it would be interesting to include either a functional language or a dynamically typed language. It turns out that the only programming languages that have an SBST tool available are: Java (EvoSuite), Python (Pynguin), C# (Randoop) , C++ (KLEE/CBCMC). None of these languages are purely functional and Python is the only dynamically typed one. Thus, Python is the second language of choice with Pynguin as the SBST tool since it is the only tool which can generate tests in a completely automatic way for Python [10].

Both programming languages need a corpus of classes for evaluation. For Java this will be a subset of the SF110[6] corpus since these classes are not on GitHub which means Llama3 is likely not trained on these classes. This matters because Llama3 likely performs much better for classes it is trained on. SF110 has also been commonly used for benchmarking and is used in various related papers [6] [4] [8]. 20 Java classes from SF110 with a cyclomatic complexity[7] of at least 4 and with few dependencies will be used as the Java corpus. A subset of 20 Python classes will be chosen from the corpus used in the paper by Stephan Lukasczyk and Gordon Fraser [10] their results can potentially be compared against the results of this paper. In short, the corpus for this research contains 20 Java classes and 20 Python classes.

SBST tools generate some finite amount of tests within their time limit. For Llama3 however, we could theoretically ask for an infinite amount of test cases. However, at some point the mutation score will just converge because it is either maxed out at 100% or because of the limited capability of Llama3 to generate test cases which further improve mutation score. We will generate exactly 8 passing test cases for each class in the corpus per run (there will be 12 runs in total per class per tool as stated earlier). The number 8 is based on empirical evidence observed during preliminary research. To support this hypothesis that 8 tests are indeed sufficient, an experiment will be conducted where the mutation score is measured for each test added to a test suite for any class in the corpus. The convergence or divergence of this data will support or refute this hypothesis.

To elicit test cases from Llama3, prompts are needed which contain the source code of the class under test. Figure 1 shows the flow of prompts. It starts off by sending the source code of the class under test without any dependencies. The reason dependencies are not sent is because it has been observed from preliminary research that this does not improve performance.

In Figure 1, there are some placeholders between curly brackets and they will be briefly explained here. The placeholder {code} for the source code of the class under test, {imports} means the imports used in the class under test plus any additional needed imports related to the testing framework used. Placeholder {extra} means any additional language specific instructions needed to get working tests. The exact contents of this {extra} placeholder is also shown in Figure 1 and have been obtained through preliminary research. Placeholder {error} is to be filled with the error message of why the test case failed. The depicted cycle in Figure 1 is continued per class until 8 passing test cases have been obtained.

4 Study Design

This section details the experiment that has been conducted to assess the effectiveness of Llama3 for generating unit tests. This experiment is steered by the following research question:

RQ1 *How effective is Llama3 70B at generating unit tests with regards to mutation score?*

4.1 Corpus

The Java corpus used in this experiment is listed in Figure 2. The metrics have been defined and computed by CK[8]. PITest[9] in combination with Gradle[10] is used to calculate mutation score for the Java corpus.

The Python corpus that will also be used in this experiment is listed in Figure 3, it also includes the version of the project. This was not applicable for the Java classes since they come from the SF110[11] corpus. All metrics in Figure 3 have been defined and computed by Radon[12]. All Python classes can be found on PyPI[13] by their project name from Figure 3. The tool mut.py[14] will be used to calculate mutation scores for the Python corpus. The replication package is publicly available on Github[15] to ensure reproducibility.

4.2 Llama3 provider

For this research, we will use the "meta/meta-llama-3-70b-instruct" instance hosted by Replicate[16]. Conversations with this model are not directly supported, so they have been simulated by wrapping a user request in '[INST] [/INST]' block since this worked for "meta/llama-2-70b-chat" as shown by an example in Figure 4. So

[6] https://www.evosuite.org/experimental-data/sf110/
[7] https://en.wikipedia.org/wiki/Cyclomatic_complexity
[8] https://github.com/mauricioaniche/ck
[9] https://pitest.org/
[10] https://gradle.org/
[11] https://www.evosuite.org/experimental-data/sf110/
[12] https://radon.readthedocs.io/
[13] https://pypi.org/
[14] https://github.com/mutpy/mutpy
[15] https://github.com/flazedd/bug-busters
[16] https://replicate.com/meta/meta-llama-3-70b-instruct

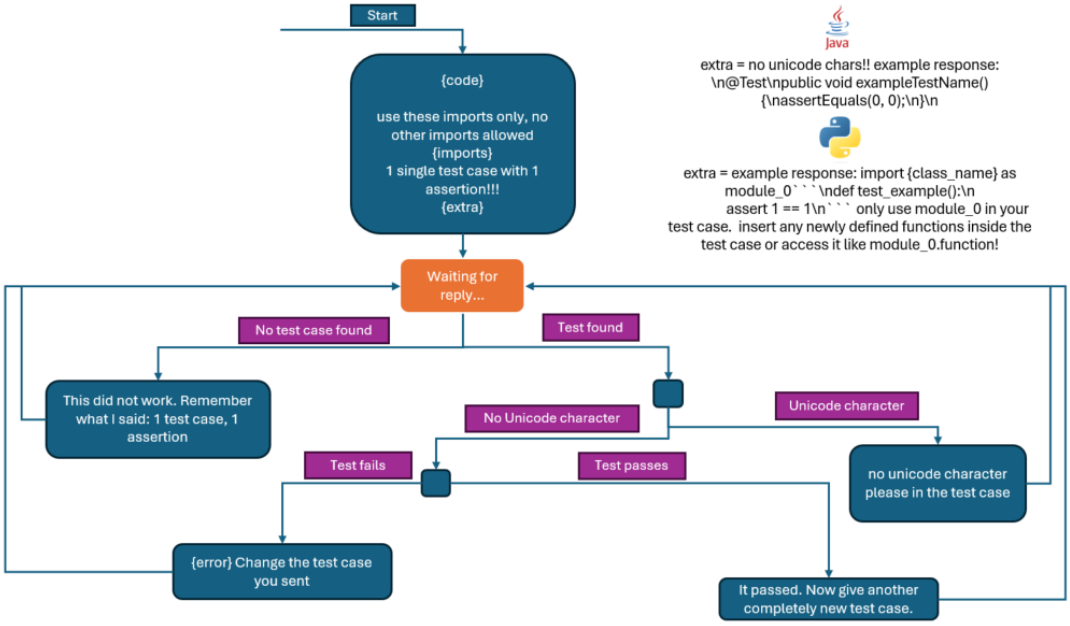

Figure 1: Flowchart illustrating how the agent communicates with Llama3 to elicit test cases. Blue boxes are messages sent. Purple boxes are results of local computations based on replies from Llama3.

	Project	Class	CC	Largest CC of a method	LOC	Method count	Dependency
0	1_tullibee	com.ib.client.Util	22	11	45	7	No
1	1_tullibee	com.ib.client.Contract	26	23	90	4	Yes
2	5_templateit	org.templateit.OpMatcher	27	8	90	5	Yes
3	7_sfmis	com.hf.sfm.crypt.Base64	22	9	103	9	No
4	11_imsmart	com.imsmart.servlet.HTMLFilter	8	8	27	1	No
5	15_beanbin	net.sourceforge.beanbin.search.WildcardSearch	11	10	42	2	No
6	24_saxpath	org.saxpath.Axis	28	14	89	2	No
7	33_javaviewcontrol	com.pmdesigns.jvc.tools.Base64Coder	35	17	81	7	No
8	33_javaviewcontrol	com.pmdesigns.jvc.tools.HtmlEncoder	14	14	40	1	No
9	35_corina	corina.util.NaturalSort	47	18	87	8	No
10	35_corina	corina.util.Sort	29	11	79	8	No
11	35_corina	corina.util.StringComparator	7	6	20	2	No
12	35_corina	corina.util.StringUtils	26	11	83	7	No
13	73_fim1	osa.ora.server.utils.StringEncoder64	38	10	139	9	No
14	36_schemaspy	net.sourceforge.schemaspy.util.Version	14	5	40	4	No
15	51_jiprof	org.objectweb.asm.jip.ByteVector	33	14	158	11	No
16	52_lagoon	nu.staldal.util.Utils	25	11	69	5	No
17	66_openjms	org.exolab.jms.util.CommandLine	24	8	80	10	No
18	68_biblestudy	bible.util.Queue	28	7	113	13	No
19	72_battlecry	bcry.bcWord	27	10	76	9	No

Figure 2: Overview of all the Python classes included in the experiment. Cyclomatic complexity (CC) is the total cyclomatic complexity of all methods. LOC is defined as lines of code excluding empty lines and comments. Dependency is defined as any dependency which is not included in standard libraries.

every time the entire conversation is sent back to the model with the addition of the user request.

4.3 Settings SBST tools

EvoSuite and Pynguin are the SBST tools used in this study. This section specifies the settings and hardware on which they will run. Arguably the two most important settings for the SBST tools are the search algorithm used for generating test cases and the time constraint imposed on this algorithm. A bad algorithm or little time will obviously lead to bad test cases. To get a good benchmark we should strive to use the optimal settings. A large scale study by Panichella et al. [15] compared different search algorithms for automated test

	Project	Version	Class	CC	Largest CC of a method	LOC	Method count	Dependency
0	codetiming	1.4.0	timers	21	3	77	12	No
1	dataclasses_json	0.6.7	stringcase	11	3	130	6	No
2	docstring_parser	0.16	common	49	5	228	14	No
3	docstring_parser	0.16	epydoc	56	39	268	4	No
4	docstring_parser	0.16	google	84	27	408	11	No
5	docstring_parser	0.16	numpydoc	123	38	532	21	No
6	docstring_parser	0.16	parser	11	6	98	3	No
7	docstring_parser	0.16	rest	59	25	259	4	No
8	flutils	0.7	txtutils	66	31	424	15	Yes
9	flutils	0.7	validators	10	10	82	1	No
10	httpie	3.2.2	status	6	5	40	1	No
11	isort	5.13.2	comments	7	5	32	2	No
12	pymonet	0.12.0	immutable_list	36	5	166	14	No
13	pyutils	0.0.14	bst	123	23	352	34	Yes
14	pyutils	0.0.14	centcount	63	8	373	20	No
15	pyutils	0.0.14	logical_search	86	22	306	18	Yes
16	pyutils	0.0.14	money	70	9	370	21	No
17	pyutils	0.0.14	rate	26	6	131	16	No
18	pyutils	0.0.14	trie	48	9	398	16	No
19	typesystem	0.4.1	unique	17	8	60	4	No

Figure 3: Overview of all the Python classes included in the experiment. Cyclomatic complexity (CC) is the total cyclomatic complexity of all methods. LOC is defined as lines of code excluding empty lines and comments. Dependency is defined as any dependency which is not included in standard libraries.

```
prompt = """\
[INST] Hi! [/INST]
Hello! How are you?
[INST] I'm great, thanks for asking. Could you help me with a task? [/INST]"
```

Figure 4: Example dialog of a user with Llama2.

case generation for Java classes. They did this for over 180 Java classes and found that the algorithm DynaMOSA outperformed the rest in terms of effectiveness and efficiency where only branch

coverage was measured. Another paper by Stephan Lukasczyk et al [12] found that DynaMOSA performed best for Pynguin out of all algorithms available for Pynguin at that time. Therefore, the DynaMOSA algorithm will be used for both EvoSuite and Pynguin. In another paper by Panichella et al. [14], where mutation score was measured on test suites generated by EvoSuite, it is empirically observed that little to no classes improve in mutation score if run for 60s compared to 180s. Therefore, a search budget of 120s for EvoSuite to generate a single test suite for a single class is deemed adequate. The EvoSuite test suites are generated by running it on two AMD EPYC 7H12 64-Core processors which equates to 128 cores 512 threads since hyperthreading is enabled. The frequency of this CPU is between 1.5Ghz and 2.6Ghz. The CPU is accompanied by 256GB of RAM and the operating system is Ubuntu 22.04.

Pynguin will be run for 30s per class per test suite since this is deemed adequate through preliminary research. Preliminary research concluded that there was a lack of evidence that the mutation score of the test suites generated by Pynguin would increase if ran for longer than 30s even though numerous papers [10] [12] [11] have run Pynguin for 600s. We include an experiment to support this hypothesis in the results where the median mutation score per class is taken where all runs have used 30s. Then, Pynguin will be ran a single time with 60s and a single time with 90s and mutation scores are compared. Pynguin is run on an AMD Ryzen 5 3600 6-core processor with 12 threads since hyperthreading is enabled, 16GB of RAM and on Windows 11.

4.4 Statistical analysis

For each of the 20 classes in the Java corpus, 12 test suites have been generated by EvoSuite and 12 test suites by Llama3. For each test suite the mutation score has been computed, resulting in 12 mutation scores per tool (EvoSuite and Llama3). Then the Wilcoxon signed-rank test[17] has been used to compute the probability of observing the obtained data under the null hypothesis that two related paired samples come from the same distribution. If this probability (p-value) is less than 0.05 we reject the null hypothesis because we are at least 95% confident that the observed difference between the samples is not due to random variation alone, but because of a true difference in the distributions being compared. Then, if the distributions differ significantly, Vargha and Delaney A[18] is used to quantify how big this difference is. The output domain of this function is $[0, 1]$ where a value of less than 0.50 indicates the second group (Llama3) dominates the first group (EvoSuite) and a score larger than 0.50 means the exact opposite of this. A score of 0.5 means there's stochastic equality between the two groups. This approach is repeated for the 20 Python classes except that EvoSuite is swapped out for Pynguin.

5 Results

Figure 5 shows the median mutation scores of both EvoSuite and Llama3 with their corresponding p-values from the Wilcoxon test and Vargha and Delaney $\hat{A}_{12}$ score. Whenever the p-value is below 0.05 the cell is colored red with its corresponding Vargha and Delaney $\hat{A}_{12}$ score if the EvoSuite mutation scores are larger than the Llama3 mutation scores. The cells are colored green for the opposite cases. Our statistical analysis reveals that EvoSuite statistically outperforms Llama3 for 13 out of 20 Java classes. Llama3 outperforms EvoSuite for only 3 out of 20 Java classes.

	Project	Class	EvoSuite Median Mutation Score	Meta_Llama_3_70B_Instruct Median Mutation Score	p-value Wilcoxon	Vargha-Delaney effect size
0	battlecry_72	bcWord	79.0	63.0	0.0049	L (0.8438)
1	beanbin_15	WildcardSearch	96.0	62.0	0.0005	L (0.9896)
2	biblestudy_68	Queue	82.0	74.0	0.0342	M (0.7118)
3	corina_35	NaturalSort	75.0	43.0	0.001	L (0.9722)
4	corina_35	Sort	29.0	71.0	0.0005	L (0.0)
5	corina_35	StringComparator	100.0	81.5	0.0005	L (1.0)
6	corina_35	StringUtils	87.5	78.0	0.064	M (0.7326)
7	fim1_73	StringEncoder64	79.5	63.0	0.0005	L (1.0)
8	imsmart_11	HTMLFilter	100.0	100.0	1.0	- (0.5)
9	javaviewcontrol_33	Base64Coder	94.0	94.5	0.3804	S (0.3993)
10	javaviewcontrol_33	HtmlEncoder	69.5	78.0	0.0342	L (0.2326)
11	jiprof_51	ByteVector	31.0	17.0	0.0005	L (1.0)
12	lagoon_52	Utils	83.0	46.0	0.0005	L (1.0)
13	openjms_66	CommandLine	88.0	67.5	0.0005	L (1.0)
14	saxpath_24	Axis	100.0	50.0	0.0005	L (1.0)
15	schemaspy_36	Version	84.0	58.0	0.0005	L (1.0)
16	sfmis_7	Base64	77.5	92.0	0.001	L (0.0625)
17	templateit_5	OpMatcher	72.0	69.0	0.3013	M (0.6875)
18	tullibee_1	Contract	100.0	72.0	0.0005	L (1.0)
19	tullibee_1	Util	100.0	43.0	0.0005	L (1.0)

Figure 5: Median mutation score achieved by our approach of generating tests with Llama3 and the baseline EvoSuite over 12 independent runs. Furthermore, we show the p-value (Wilcoxon test) and the Vargha-Delaney effect size $\hat{A}_{12}$. We labeled the effect size with S, M or L to denote small, medium and large effect size. Colored cells indicate that the p-value is significant ($\alpha = 0.05$). Green indicates that Llama3 performed better and red indicates that EvoSuite performed better.

The same statistics have been obtained for the Python corpus in Figure 6 where Pynguin is used instead of EvoSuite. In Figure 6, our statistical analysis reveals that Llama3 statistically outperforms Pynguin for 9 out of 20 Python classes. Pynguin outperforms Llama3 for only 4 out of 20 Java classes.

	Project	Class	Pynguin Median Mutation Score	Meta_Llama_3_70B_Instruct Median Mutation Score	p-value Wilcoxon	Vargha-Delaney effect size
0	codetiming	timers	39.45	42.1	0.6221	- (0.4688)
1	dataclasses_json	stringcase	100.0	100.0	0.7334	- (0.5347)
2	docstring_parser	common	50.0	27.1	0.0161	L (0.7639)
3	docstring_parser	epydoc	16.75	57.2	0.0024	L (0.1597)
4	docstring_parser	google	14.9	42.0	0.0005	L (0.0)
5	docstring_parser	numpydoc	13.05	17.2	0.0015	L (0.1181)
6	docstring_parser	parser	50.0	42.9	0.9697	M (0.6875)
7	docstring_parser	rest	15.2	51.05	0.0005	L (0.0104)
8	flutils	txtutils	49.55	41.3	0.0068	L (0.7847)
9	flutils	validators	65.85	70.0	0.6772	S (0.3889)
10	httpie	status	64.7	73.55	0.001	L (0.097)
11	isort	comments	37.5	75.0	0.0034	L (0.1806)
12	pymonet	immutable_list	33.3	63.7	0.0015	L (0.0556)
13	pyutils	bst	12.9	27.95	0.0342	L (0.1111)
14	pyutils	centcount	54.5	27.55	0.0005	L (1.0)
15	pyutils	logical_search	0.0	100.0	0.0034	L (0.1285)
16	pyutils	money	57.2	28.5	0.0005	L (1.0)
17	pyutils	rate	47.65	47.65	0.7334	- (0.5417)
18	pyutils	trie	40.9	50.0	0.5186	- (0.5)
19	typesystem	unique	81.65	91.65	0.064	M (0.2986)

Figure 6: Median mutation score achieved by our approach of generating tests with Llama3 and the baseline Pynguin over 12 independent runs. Furthermore, we show the p-value (Wilcoxon test) and the Vargha-Delaney effect size $\hat{A}_{12}$. We labeled the effect size with S, M or L to denote small, medium and large effect size. Colored cells indicate that the p-value is significant ($\alpha = 0.05$). Green indicates that Llama3 performed better and red indicates that Pynguin performed better.

For this research, we made the decision to generate 8 compiling and passing test cases with Llama3 for each class in the corpus. The hypothesis was that for most classes the addition of any further test cases will likely not increase mutation score by much if at all. In

[17] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.wilcoxon.html
[18] https://gist.github.com/jacksonpradolima/f9b19d65b7f16603c837024d5f8c8a65

Figure 7 and in Figure 8, we show that most classes have converged in mutation score or are slowly improving which supports this hypothesis. From empirical inspection, there is a negative correlation for classes in Figure 2 between their cyclomatic complexity and mutation score in Figure 7 and 8.

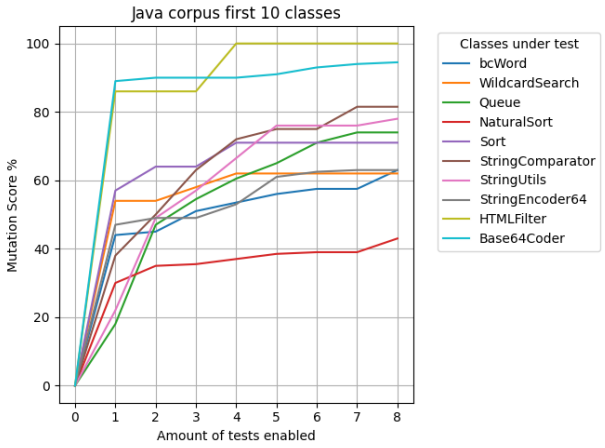

Figure 7: Improvement in mutation score when more tests are added to the test suite for the first 10 classes of the Java corpus

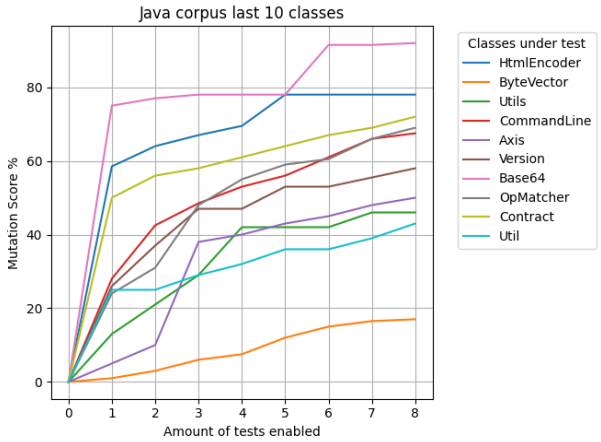

Figure 8: Improvement in mutation score when more tests are added to the test suite for the last 10 classes of the Java corpus

For the Python corpus in Figure9 and 10, we observe that the mutation score has converged or is slowly improving for most classes. From empirical inspection, there is a negative correlation for classes in Figure 3 between their cyclomatic complexity and mutation score in Figure 9 and 10.

Figure 11 shows the median mutation score of the 12 runs used in the research and the mutation score for a single run of 60s and 90s. The highest mutations scores per class are marked in green, in the case of a tie, multiple boxes are colored green. We observe

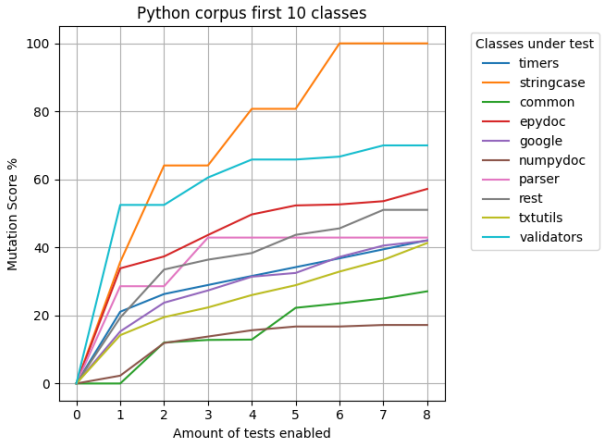

Figure 9: Improvement in mutation score when more tests are added to the test suite for the first 10 classes of the Python corpus

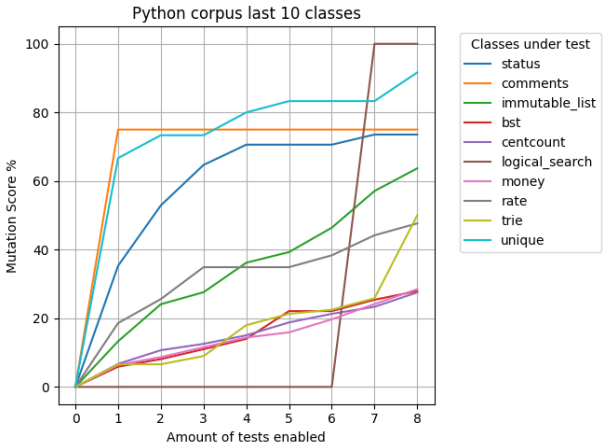

Figure 10: Improvement in mutation score when more tests are added to the test suite for the last 10 classes of the Java corpus

that performance does not strictly improve when running Pynguin for a longer duration. This supports the hypothesis that running Pynguin for 30s is not worse than 60s or 90s.

Figure 12 shows the average mutation score over all Java classes over all 12 independent runs. We observe that EvoSuite attains a higher mutation score (81.05%) than Llama3 (66.26%). In Figure 13, the same statistic is calculated for the Python corpus with Pynguin, but here Llama3 (51.95%) performs better on average than Pynguin (42.73%).

6 Responsible Research

LLMs are nondeterministic and are known to make mistakes. An ethical concern is that if tests are automatically generated by an LLM for critical software, then one should not blindly trust these generated tests and conclude that the software works as intended

	Project	Class	Pynguin 30s Median Mutation Score	Pynguin 60s Mutation Score	Pynguin 90s Mutation Score
0	codetiming	timers	39.45	47.4	42.1
1	dataclasses_json	stringcase	100.0	90.0	100.0
2	docstring_parser	common	50.0	44.0	38.5
3	docstring_parser	epydoc	16.75	18.1	53.9
4	docstring_parser	google	14.9	14.2	15.6
5	docstring_parser	numpydoc	13.05	14.8	15.6
6	docstring_parser	parser	50.0	16.7	16.7
7	docstring_parser	rest	15.2	21.8	37.0
8	flutils	txtutils	49.55	43.1	68.0
9	flutils	validators	65.85	75.0	70.6
10	httpie	status	64.7	76.5	58.8
11	isort	comments	37.5	70.0	70.0
12	pymonet	immutable_list	33.3	21.9	25.8
13	pyutils	bst	12.9	100.0	99.2
14	pyutils	centcount	54.5	50.0	48.6
15	pyutils	logical_search	0.0	0.0	0.0
16	pyutils	money	57.2	51.4	53.5
17	pyutils	rate	47.65	39.5	32.6
18	pyutils	trie	40.9	96.1	18.2
19	typesystem	unique	81.65	60.0	85.7

Figure 11: Mutation score achieved by Pynguin for different durations (30s, 60s, 90s)

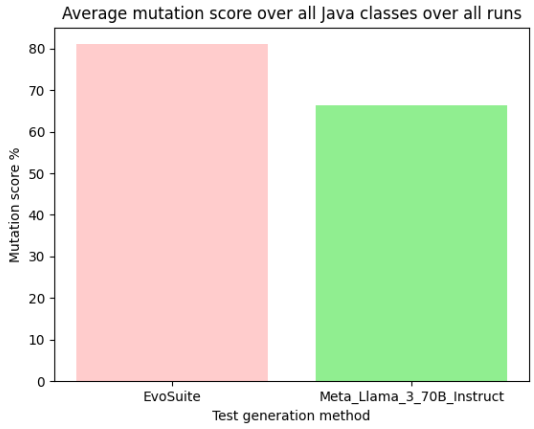

Figure 12: Arithmetic average of mutation score over all 20 Java classes over all 12 independent runs for EvoSuite and Llama3

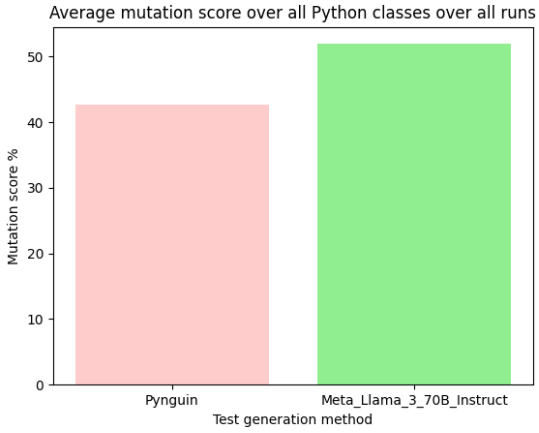

Figure 13: Arithmetic average of mutation score over all 20 Python classes over all 12 independent runs for Pynguin and Llama3

because the tests from the LLM passed. A human should always validate test cases generated by LLMs to ensure they are asserting the intended behaviour. If a test from an LLM is accepted, a human should be held accountable if the software still fails and/or the test is not asserting the right behaviour.

7 Discussion

7.1 Results

In the results, we observed that there was a negative correlation between the cyclomatic complexity (CC) of a class and the mutation score. This makes sense because classes with a higher CC can have more mutants to be killed and thus mutation score is usually lower when the same amount of tests are used.

Another interesting observation is that the entire Java corpus is not available on GitHub but the entire Python corpus is. Remarkable discrepancies have been shown between performance of LLMs on datasets that are available on GitHub compared to ones that are not available on GitHub [18]. So, we expected that Llama3 would perform better on the Java corpus. Contrary to our expectation, Llama3 achieved a higher mutation score (66.26%) on the Java corpus than on the Python corpus (51.95%). We believe this difference in performance is caused by the statically typed nature of Java compared to the dynamically typed nature of Python. We believe that the reason for this is that static typing is providing extra information in the code which is helpful to Llama3.

An interesting observation made during the generation of tests is that when the LLM has a passing test it will quickly produce more passing tests since it knows what already works. So, 5 failures followed by a passing test case is usually followed by mostly passing test cases.

7.2 Threats to validity

One threat to the validity of the research is that Llama3 is closed-source and therefore it is unknown if Llama3 has been trained on the Java and Python corpus used in this research. If Llama3 was trained on those classes it would perform better in this experiment than if it had not been. However, it is unlikely that Llama3 has been trained on the Java corpus since it does not exist on GitHub which mitigates this risk.

Another threat to the validity is the randomness of Llama3, since it will not always produce the same output for the same input (nondeterminism). This is mitigated though by taking 12 samples per class per tool.

7.3 Limitations

A limitation of generating tests with Llama3 is that it is quite time consuming compared to generating tests with either EvoSuite or Pynguin. Generating 8 compiling and passing test cases with Llama3 takes slightly under a minute if everything passes instantly. It takes slightly over a minute if there are some failures (i.e. 4 failures and 8 passing ones and thus 12 total). For some classes the LLM is struggling to generate passing cases and then it could take 10 minutes of restarting the agent before 8 passing tests are achieved. Difficult classes are apparently classes which need a defined function or class inside the test case to run. The LLM first assumes the function exists which is not the case, then it does generate the function but

not inside the test case. After some tries it does put the function inside and the test case passes.

8 Conclusion

In this paper, we have gathered a corpus of 20 Java and 20 Python classes and generated test suites with EvoSuite and Pynguin, respectively. Then for each class in the corpus we also generated a test suite of 8 unit tests with Llama3 and thereafter measured mutation scores of all obtained test suites. The results showed that EvoSuite outperforms Llama3 in terms of mutation score since EvoSuite performed significantly better than Llama3 for 13 out of 20 Java classes and Llama3 only performed significantly better than EvoSuite for 3 Java classes. Considering all the Java classes over all 12 independent runs, the average mutation score for EvoSuite was 81.05% while that of Llama3 was 66.26%. Both these findings support the hypothesis that EvoSuite is more effective than Llama3 for test suite generation in terms of mutation score. The results also showed that Llama3 performs significantly better than Pynguin for 9 out of 20 Python classes and that Pynguin also performed better than Llama3 for 4 Python classes. Considering all the Python classes over all 12 independent runs, the average mutation score for Pynguin was 42.73% while that of Llama3 was 51.95%. Both findings indicate that Llama3 is more effective than Pynguin for test suite generation in terms of mutation score.

9 Future work

In future work, different LLMs can be explored such as ChatGPT-4o and then results could be compared to this paper. Also, more and different programming languages could be considered to see if results generalize to different languages. Java and Python are the two most popular languages[19] which means there is likely more training data available for those languages than for less popular languages. This means that LLMs would have more training data for Java and Python and thus perform better than for less popular languages. This hypothesis remains to be verified by future work.

Another interesting approach would be to research a functional language like Haskell to see how results generalize across different programming paradigms. Furthermore, a topic which needs attention is to figure out how to optimally prompt an LLM to achieve optimal performance. In this research, a simple prompt was chosen that seemed to work from preliminary research, but it would be interesting to see what prompting techniques lead to the optimal desired response. We hypothesize this will be different for different LLMs based on the findings in this research.

References

[1] M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. 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)*. 263–272. https://doi.org/10.1109/ICSE-SEIP.2017.27

[2] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2014. Automated unit test generation for classes with environment dependencies. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 79–90. https://doi.org/10.1145/2642937.2642986

[3] 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

[4] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.* 24, 2, Article 8 (dec 2014), 42 pages. https://doi.org/10.1145/2685612

[5] N. Johnsson. 2024. *An In-Depth Study on the Utilization of Large Language Models for Test Case Generation*. Ph. D. Dissertation. Umeå University, Faculty of Science and Technology, Department of Computing Science.

[6] Wanida Khamprapai, Cheng-Fa Tsai, Paohsi Wang, and Chi-En Tsai. 2021. Performance of Enhanced Multiple-Searching Genetic Algorithm for Test Case Generation in Software Testing. *Mathematics* 9, 15 (2021). https://doi.org/10.3390/math9151779

[7] Claus Klammer and Albin Kern. 2015. Writing unit tests: It's now or never!. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 1–4. https://doi.org/10.1109/ICSTW.2015.7107469

[8] Dongcheng Li, W. Eric. Wong, Shenglong Li, and Matthew Chau. 2022. Improving Search-based Test Case Generation with Local Search using Adaptive Simulated Annealing and Dynamic Symbolic Execution. In *2022 9th International Conference on Dependable Systems and Their Applications (DSA)*. 290–301. https://doi.org/10.1109/DSA56465.2022.00047

[9] Kaibo Liu, Yiyang Liu, Zhenpeng Chen, Jie M. Zhang, Yudong Han, Yun Ma, Ge Li, and Gang Huang. 2024. LLM-Powered Test Case Generation for Detecting Tricky Bugs. arXiv:2404.10304 [cs.SE]

[10] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated Unit Test Generation for Python. In *ICSE'22: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. ACM, 168–172. https://doi.org/10.1145/3510454.3516829

[11] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2020. *Automated Unit Test Generation for Python*. Springer International Publishing, 9–24. https://doi.org/10.1007/978-3-030-59762-7_2

[12] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2022. An Empirical Study of Automated Unit Test Generation for Python. arXiv:2111.05003

[13] Francis Palma, Tamer Abdou, Ayse Basar, John Maidens, and Stella Liu. 2018. An Improvement to Test Case Failure Prediction in the Context of Test Case Prioritization. 80–89. https://doi.org/10.1145/3273934.3273944

[14] Annibale Panichella, José Campos, and Gordon Fraser. 2020. EvoSuite at the SBST 2020 Tool Competition. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (Seoul, Republic of Korea) *(ICSEW'20)*. Association for Computing Machinery, New York, NY, USA, 549–552. https://doi.org/10.1145/3387940.3392266

[15] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information and Software Technology* 104 (2018), 236–256. https://doi.org/10.1016/j.infsof.2018.08.009

[16] A. Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent Hellendoorn. 2022. Test Smells 20 Years Later: Detectability, Validity, and Reliability. *Empirical Software Engineering* 27, 7 (2022). https://doi.org/10.1007/s10664-022-10207-5

[17] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware Prompting: A study of Coverage Guided Test Generation in Regression Setting using LLM. arXiv:2402.00097 [cs.SE]

[18] 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

[19] Beatriz Souza and Patrícia Machado. 2020. A Large Scale Study On the Effectiveness of Manual and Automatic Unit Test Generation. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering* (<conf-loc>, <city>Natal</city>, <country>Brazil</country>, </conf-loc>) *(SBES '20)*. Association for Computing Machinery, New York, NY, USA, 253–262. https://doi.org/10.1145/3422392.3422407

[20] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. 2024. ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation. *IEEE Transactions on Software Engineering* (2024), 1–19. https://doi.org/10.1109/TSE.2024.3382365

[21] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing with Large Language Models: Survey, Landscape, and Vision. arXiv:2307.07221 [cs.SE]

[22] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. arXiv:2305.04207 [cs.SE]

[19] https://spectrum.ieee.org/the-top-programming-languages-2023