Using local LLMs in constrained environments for increasing mutation score

Roelof van der Geest[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 25, 2024

Name of the student: Roelof van der Geest
Final project course: CSE3000 Research Project
Thesis committee: Panichella, A., Olsthoorn, M., Poulsen, C.

Abstract

Mutation testing is a way to test the effectiveness of a test suite for catching bugs in a given piece of code. Writing these tests manually can be cumbersome and time-consuming. Automated tools can be used to generate tests that achieve a high mutation score. The output of these tools is often very hard to understand for humans, and therefore rarely used as actual test suites for software programs. Because LLMs have been shown to be able to generate programs that can be more easily understood by humans, we ask if these LLMs can be used for improving or generating tests for the purpose of mutation testing. Some LLMs run in the cloud, while others run locally. Cloud-based LLMs such as ChatGPT or Copilot are not always an option because of privacy concerns, speed, or regulations, but do not require possession of hardware. Local LLMs do not have the privacy concerns, but sometimes require large amounts of hardware to be available. This paper will focus on local LLMs that can be run in a computationally restricted environment. We present an automated approach to use a local LLM to improve the mutation score of existing test suites. We compare three different models (DeepSeek Coder, Code Llama and Codestral), evaluated on publicly available datasets. Using this approach, we were able to successfully generate unit tests that, combined with the existing manually written tests, are able to increase the mutation score around one third to half of the time depending on the model.

1 Introduction

As more and more of society becomes reliant on software, having proper tests for this software becomes very important to assert its quality [4]. Writing tests for software can be cumbersome and time consuming [11], which often leads to developers not writing tests at all [15]. To solve this, Search Based Software Testing (SBST) tools such as EvoSuite [19] and Randoop [34] have been created to automatically generate tests. SBST is an effective way to automatically generate tests as shown in SBST competitions [29]. One downside of the tests generated by these tools is that they are often very complex [30] and hard to understand by humans [2, 23]. Furthermore, due to the oracle problem [8], these tests assume a correct code base and therefore still have to be verified by a human developer.

Recent LLMs (Large Language Models) are able to generate comprehensible tests [21], as well as generate tests from scratch [37]. Using LLMs to generate or improve tests has its own problems however. Technical problems exist such as generating failing, flaky [27, 30], or non-compiling tests [3]. The resources required to run LLMs can be prohibitively expensive. For example, the LLAMA2-70B model already requires almost €100,000 in hardware to run [41]. In cases where private source code is involved, it may not be an option to provide the source code to external parties running the LLM. When using externally hosted models, the data provided to these models is often stored and used for training purposes by the hosting party [18]. In the past, private information has already leaked via inference or extraction attacks [44]. When externally hosted models are an option, these services may become slow or unresponsive due to increasing popularity and query volume [17]. Running the LLM locally ensures that any source code provided to the LLM will not end up in its training set, preventing leaks via the aforementioned attacks.

This research aims to investigate the effectiveness of a solution using a local LLM running in a computationally constrained environment to overcome these problems. In our approach we compare three different models to answer the main research question:

- How effective are local LLMs running in constrained environments at increasing mutation score?

For the models we use Deepseek Coder 6.7B [26], Code Llama 2 13B [35] and Codestral 22B [1]. This provides us with models of different sizes, which can still be run on commodity consumer hardware. The largest model (Codestral) can run on systems with at least 20GB of video memory. This includes consumer graphics cards such the single Nvidia RTX 3090 used for this research, or an Apple Macbook with at least 24GB of unified memory.

Our approach uses these models with a 1-shot approach to generate new test cases, given the source code of the class under test. The resulting tests are extracted and added to the test suite. We then run the new test suite, removing failing or non-compiling tests until a working test suite is present. To calculate the mutation score of the resulting test suite, we use pitest. As a benchmark, we selected classes from GitHub, as well as the SF110 corpus of classes [20] to form a representative set of classes. The classes under test are selected based on cyclomatic complexity [31] and lines of code to avoid selecting classes that are too simple or too large for the context size of the models. With this approach, we were able to increase the mutation score compared to the manually written test cases for 4 to 6 out of 12 classes in the benchmark, depending on the model.

In the rest of this paper we present our method of using these models in more detail, and provide a statistical analysis of its performance.

2 Background and Related Work

The concept of test generation is not new, going back to at least the late 1970's [13]. Since then, multiple different ways of going about test generation have been developed such as fuzzing [32]. Fuzzing aims to compromise a program by sending in random, unexpected or invalid data. Other approaches are search based such as EvoSuite [19] and Randoop [34]. These techniques can also be combined with AI. Blasi et al. [9] were able to combine Randoop with Natural Language Processing to improve unit test generation.

Most of this research, however, uses coverage metrics to evaluate the quality of the generated tests. Past research has shown that the ability of a test suite to detect bugs is only loosely related to metrics such as line- or branch coverage [10, 22, 28]. Still, tools for generating tests with high mutation score already exist. EvoSuite, for example, has a parameter setting to specifically search for tests that increase mutation score. The downside of such tools is that the tests they generate are often very complex [30] and hard to understand by humans [2, 23]. This is where LLMs could come in, as they have demonstrated the ability to generate tests that are easily understandable by humans [21].

Using LLMs for test generation is far more novel, as it has only been recently that LLMs have become adept at a variety of tasks. This has caused a surge in LLM-based research, including software test generation. Tools that use LLMs for test generation have been shown to be able to successfully generate tests that improve on the existing code base [3]. Furthermore, large cloud-based models such as ChatGPT are able to produce unit tests without errors or human intervention [24].

This is not to say that no research on LLMs with regards to mutation testing has been done. It has been shown that LLMs can be effectively used to suggest mutations, which results in being able to generate test suites that cover mutants which cannot be produced using standard mutation operators [39]. Feeding surviving mutants into an LLM has also been shown to be an effective way to increase mutation score [16]. So far, no research has been done on using publicly available LLMs which can run in computationally restricted environments, specifically for increasing mutation score.

3 Approach

We propose a flexible system, where the model used for test generation can be easily interchanged. This enables the user to change to a different model, for instance depending on the release of new models or hardware specific limitations. A high-level overview of the workings of the model is given in Figure 1. The sytem takes as input some Java source code, including manual tests. The source code of the class under test is provided to the model, which is then prompted to generate extra tests. The tests are extracted from the model output and added to the original code corpus. For every generated test, we run the new test suite to find any compilation errors or failing tests. These are then removed, after which the test suite is run again. This eventually yields a new test suite, with added tests generated by the model. Mutation scores are then calculated for the manual tests, generated tests, and a test suite containing both sets of tests.

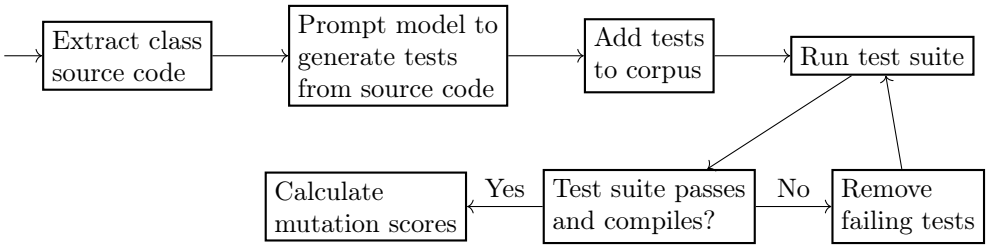

Figure 1: Flowchart of the proposed system

For the models we selected three options with different hardware requirements:

- A "small" model: Deepseek Coder 6.7B [26] version 9e221e6 with 8-bit quantization (approximately 8GB of video memory).
- a "medium" model: Code Llama 2 13B [35] version 82f1dd9 with 8-bit quantization (approximately 16GB of video memory).
- A "large" model: Codestral 22B [1] version 0e6abe1 with 6-bit K-quantization (approximately 24GB of video memory).

These models were selected because of their high performance compared to other models of the same size on benchmarks such as HumanEval [12] and MBPP [7].

We use a simple prompt to generate the tests: "Generate junit 4 tests for the following Java class: <source code here>". This prompt does not contain any instructions for the LLM to specifically target mutation score when generating tests. Preliminary testing showed that the models do not seem to have a notion of mutation score or mutation testing. Including this in the prompt made no difference to the generated tests, and was therefore omitted.

When the model has generated new source code, the tests are extracted and added to the original corpus, yielding a test suite containing both the manually written and automatically generated tests. These tests may contain false assertions or may not compile. Preliminary testing was done to explore the effectiveness of a dynamic system, where the output of the generated test suite is fed back to the model to make improvements or fix errors. During testing, the models were found to not be suitable for this approach. Out of 15 attempts, the models were able to fix a non-compiling test 2 times, although the test still did not pass

after that point. The most common reason for this is the limited context length of these models. Feeding back information into the model can quickly exceed this limit, causing the model to "forget" the initial prompt or class under test. In the vast majority of cases, the model started outputting random characters, only stopping when the generation was manually terminated. We therefore opted to implement a static 1-shot approach and take the corresponding speed benefits, foregoing a dynamic approach. In the static approach, we only run the new test suite to discover which tests are problematic. Tests with failing assertions and tests that cause compilation errors are then removed until a working test suite is left.

4 Study Design

This section describes the methods and benchmark we will be using to evaluate the performance of our system. This evaluation was chosen in such a way that it answers the main research question:

- How effective are local LLMs running in constrained environments at increasing mutation score?

4.1 Performance evaulation

To evaluate the performance of our approach, we will mainly be looking at the resulting mutation score (higher is better). Multiple mutation scores are calculated. The mutation score with only the manually written tests, the mutation score of only the generated tests, and the mutation score achieved when combining both test suites. Afterwards, we will take the mutation score of the manually written tests as a baseline, and the mutation score of the combined test suites as the performance of the system.

Research has shown that weak mutation is sufficient for obtaining relevant results for research purposes [33]. For calculating the mutation score, we use pitest with the default parameters and mutators, which covers weak mutation. This process is repeated for every class in the corpus.

In order to be able to perform meaningful statistical analysis, we repeat the described process 6 times for every model and class. To measure the statistical significance of the results, it is important to provide suitable statistical evaluation. We will therefore provide the results of Welch's t-test as well as the Vargha-Delaney effect size [40], which have been shown to be suitable for this research [6].

We also measure the runtime of the system. To remain consistent, we run all computations on the same machine sequentially, keeping interfering processes to a minimum. The system used for this research contains the following: AMD Ryzen 7 5800X 8-Core (16 threads) processor @3.8-4.7GHz, 32GB of memory @3200MHz, and a single Nvidia RTX 3090 with 24GB of video memory.

4.2 Benchmark

For the benchmark we want to use a sufficiently complex and representative set of classes. Testing the system on very simple classes will give a misrepresentation of the performance of the system, and testing on unrepresentative classes makes the results poorly generalizable to other classes. To avoid selecting classes that are too simple, we only selected classes

with a McCabe/cyclomatic complexity [31] above 10. To mimimize the risk of selecting unrepresentative classes, we selected classes from multiple different projects. Furthermore, code hosted on GitHub is almost certainly part of the training set of the models, while the SF110 corpus of classes [20] is most likely not (it is available on SourceForge instead). Past research has already shown that LLMs perform better when writing unit tests for Java code that is posted on GitHub, versus Java code that is not (SF110) [38]. To avoid giving the models an unfair advantage, we selected a combination of classes hosted on GitHub (Apache Commons) and the SF110 corpus of classes [20].

Because local LLMs have a limited context length, we can not select classes that are too large. Larger models generally support larger context sizes, but we want to avoid selecting classes that can not be processed by all three models. Therefore, we also impose a line limit of 250 lines to the classes in the benchmark. The collected corpus used as a benchmark can be found in Figure 2.

Class	**Package**	**Source**	**CC**	**LOC**
Util	org.apache.commons.cli	Apache Commons	16	33
ArrayFill	org.apache.commons.lang3	Apache Commons	17	50
Distance	org.jcvi.jillion.assembly.ca.frg	SF110	23	95
IlluminaUtil	org.jcvi.jillion.trace.fastq	SF110	31	106
Vector	jigl.math	SF110	53	139
AbstractNFeAdaptadorBean	br.com.jnfe.base.adapter	SF110	25	21
StringUtils	org.jsecurity.util	SF110	49	157
UnsyncBufferedInputStream	com.liferay.portal.kernel.io.unsync	SF110	48	173
PrefixParser	org.pdfsam.console.utils.perfix	SF110	71	219
DateTime	dk.statsbiblioteket.summa.plugins	SF110	22	97
StringMap	dk.statsbiblioteket.summa.common.util	SF110	25	80
OpMatcher	org.templateit	SF110	27	90

Figure 2: Overview of the corpus of classes used as a benchmark with cyclomatic complexity (CC) and lines of code (LOC) per class.

5 Results

In this section we present the results of using our system to generate tests. We provide an overview of the mutation score achieved on the benchmark corpus, the runtime of the system, and a statistical evaluation.

5.1 Mutation score

In this section we peresent the mutation scores that were achieved with our approach. Detailed results with exact mutation scores are available in Appendix A. A mutation score of 0% means that the model was not able to generate any compiling tests.

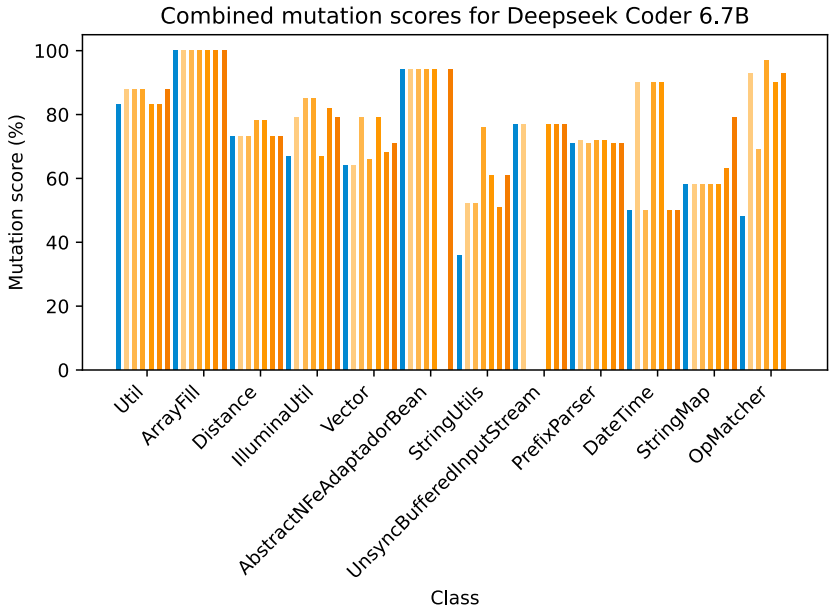

Figure 3: Combined mutation scores for Deepseek Coder 6.7B. The blue bar indicates the base mutation score, the orange bars indicate the mutation score for runs 1-6.

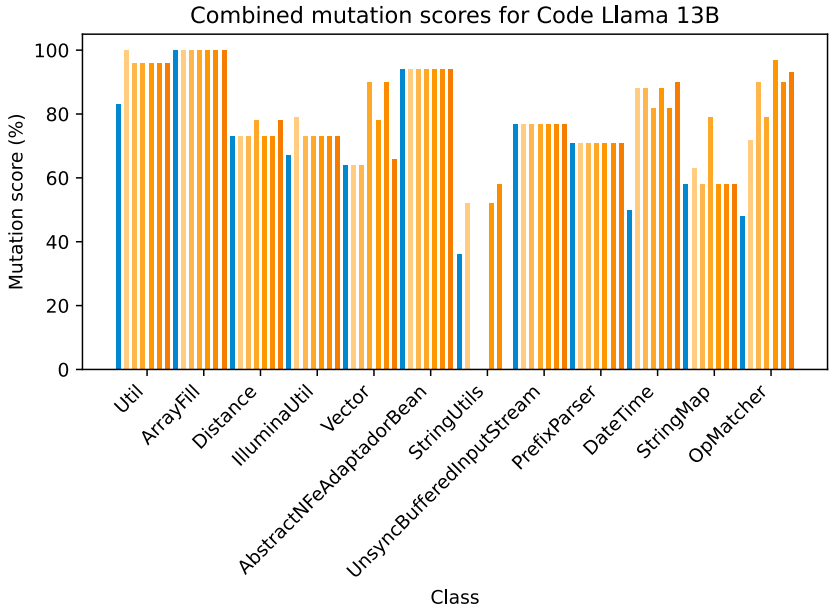

Figure 4: Combined mutation scores for Code Llama 13B. The blue bar indicates the base mutation score, the orange bars indicate the mutation score for runs 1-6.

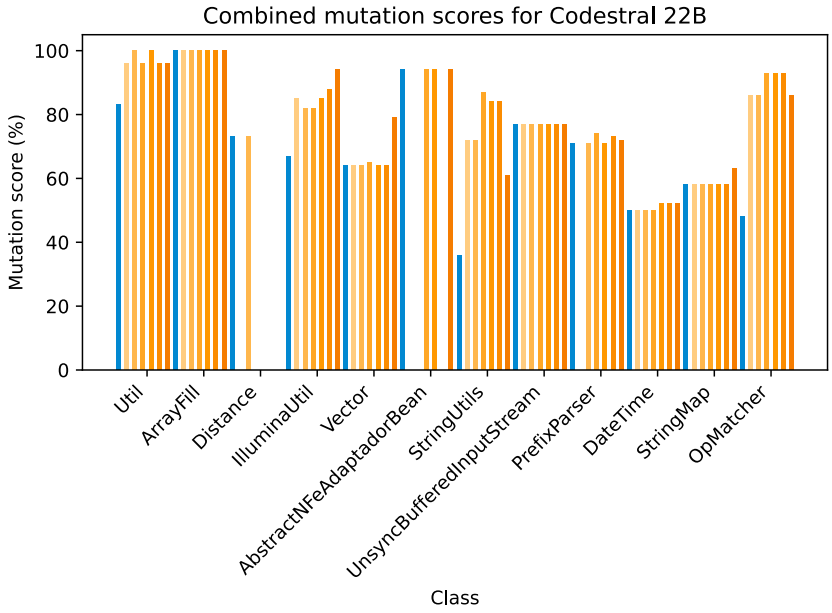

Figure 5: Combined mutation scores for Codestral 22B. The blue bar indicates the base mutation score, the orange bars indicate the mutation score for runs 1-6.

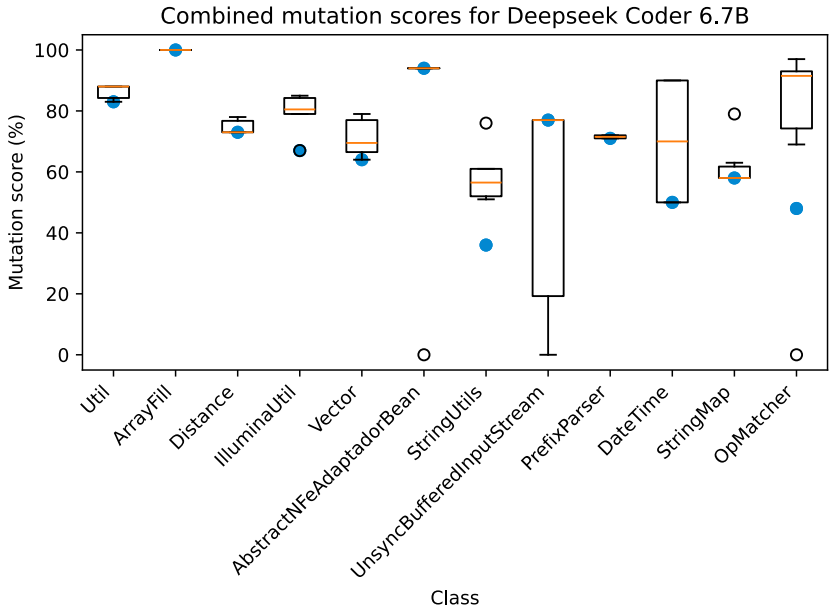

Figure 6: Combined mutation scores for Deepseek Coder 6.7B. The solid blue dot indicates the baseline.

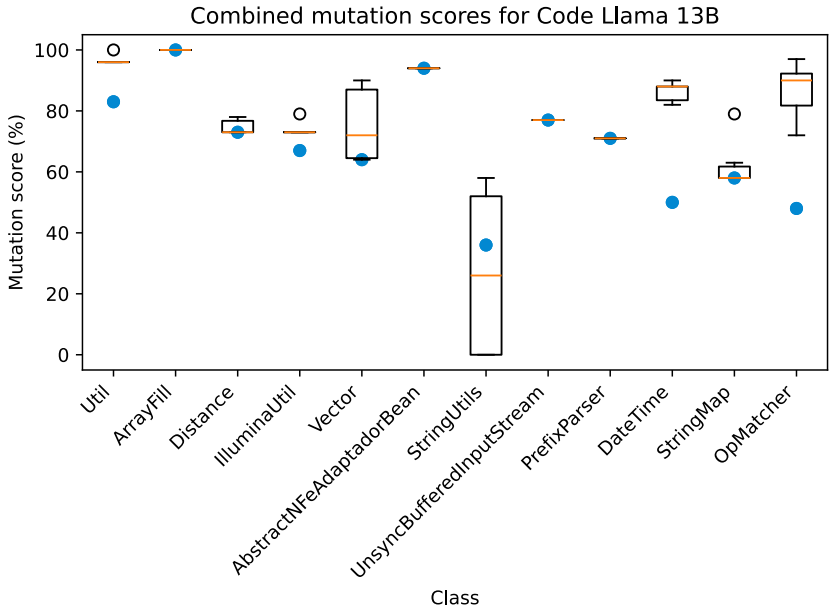

Figure 7: Combined mutation scores for Code Llama 13B. The solid blue dot indicates the baseline.

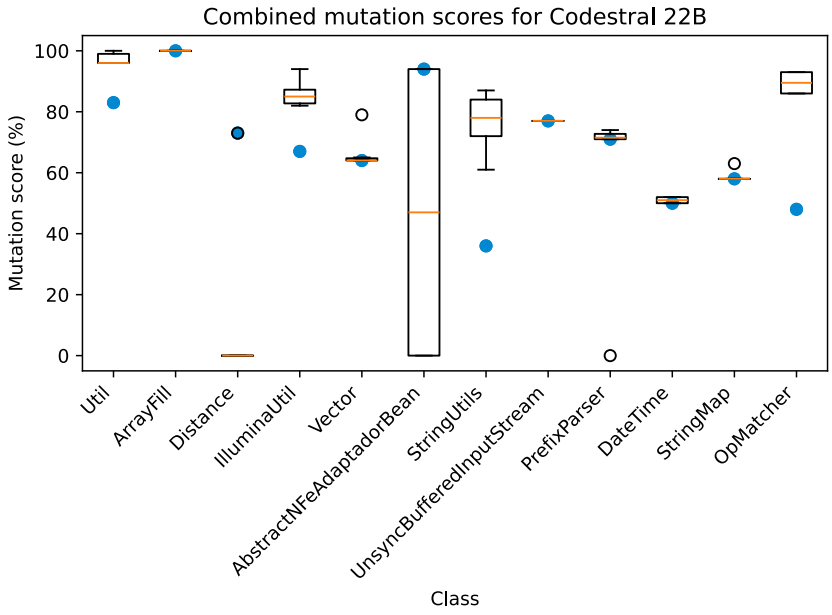

Figure 8: Boxplot with combined mutation scores for Codestral 22B. The solid blue dot indicates the baseline.

5.2 Runtime

Below we present the runtime of generating and running tests with our approach. The runtime measures the time taken for generating the tests, inserting them into the test suite, and running the test suite (possibly multiple times) to remove problematic tests.

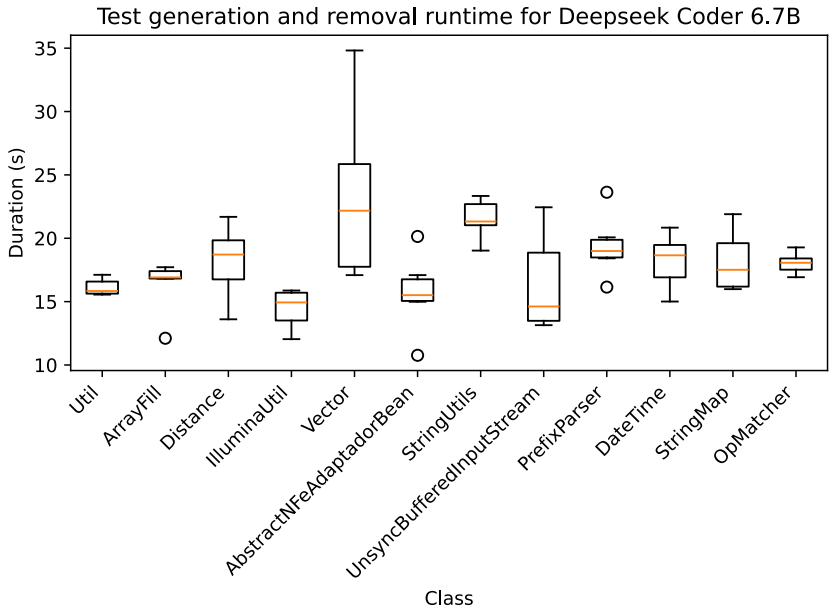

Figure 9: Test generation and removal runtime for Deepseek Coder 6.7B.

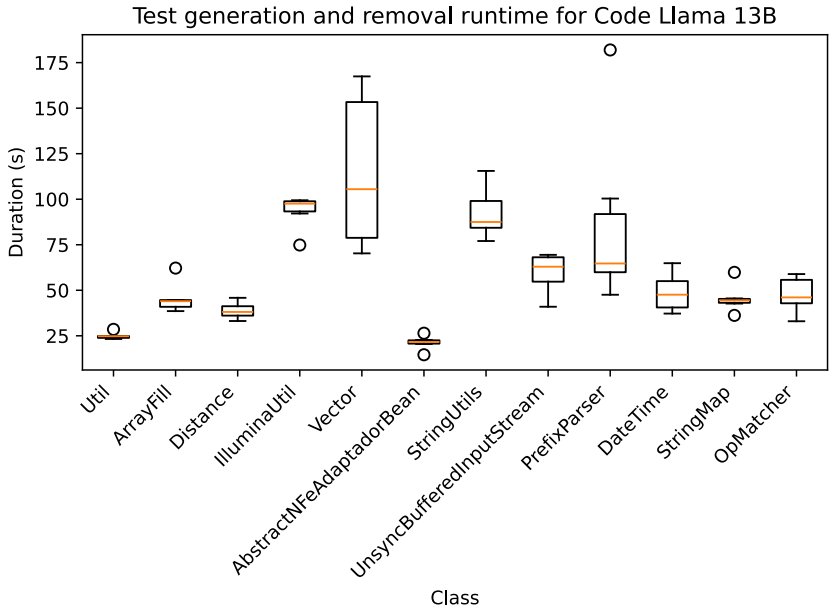

Figure 10: Test generation and removal runtime for Code Llama 13B.

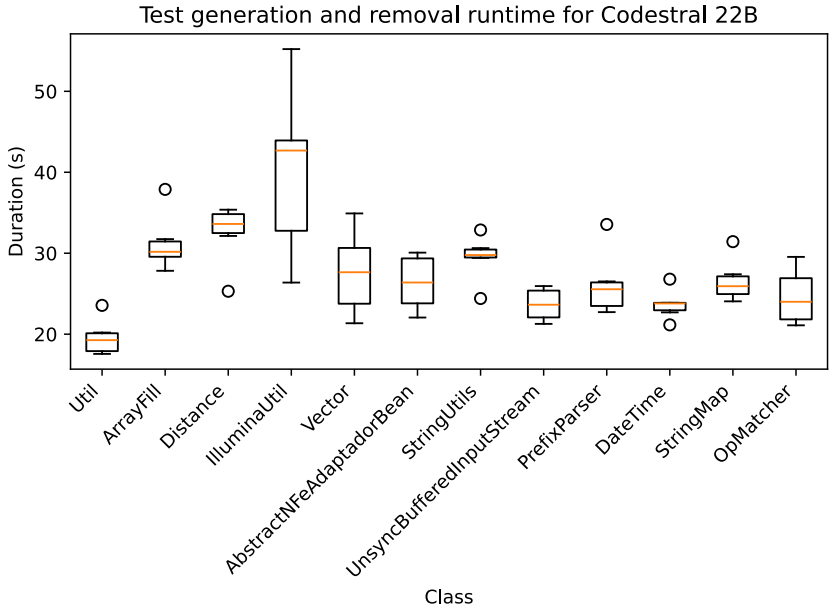

Figure 11: Test generation and removal runtime for Codestral 22B.

5.3 Statistical analysis

Below we present the computed Welch's t-test and Vargha-Delaney effect size [40].

Class	Model								
	Deepseek Coder 6.7B			Code Llama 13B			Codestral 22B		
	T-statistic	p-value	$\hat{A}_{12}$	T-statistic	p-value	$\hat{A}_{12}$	T-statistic	p-value	$\hat{A}_{12}$
Util	-3.16	2.50e-02	L(0.83)	-19.0	7.44e-06	L(1.00)	-15.7	1.84e-05	L(1.00)
ArrayFill	-	-	−(0.50)	-	-	−(0.50)	-	-	−(0.50)
Distance	-1.58	0.175	M(0.67)	-1.58	0.175	M(0.67)	5	4.10e-03	L(0.08)
IlluminaUtil	-4.58	5.95e-03	L(0.92)	-7.00	9.17e-04	L(1.00)	-10.3	1.48e-04	L(1.00)
Vector	-2.70	4.26e-02	L(0.92)	-2.24	7.53e-02	L(0.83)	-1.08	0.330	M(0.67)
AbstractNFeAdaptadorBean	1.00	0.363	S(0.42)	-	-	−(0.50)	2.24	7.56e-02	L(0.25)
StringUtils	-5.89	2.00e-03	L(1.00)	0.716	0.506	−(0.50)	-10.1	1.63e-04	L(1.00)
UnsyncBufferedInputStream	1.58	0.175	M(0.33)	-	-	−(0.50)	-	-	−(0.50)
PrefixParser	-2.24	7.56e-02	L(0.75)	-	-	−(0.50)	0.897	0.411	M(0.67)
DateTime	-2.24	7.56e-02	L(0.75)	-29.0	9.14e-07	L(1.00)	-2.24	7.56e-02	L(0.75)
StringMap	-1.27	0.259	M(0.67)	-1.27	0.259	M(0.67)	-1.00	0.363	S(0.58)
OpMatcher	-1.66	0.158	L(0.83)	-10.3	1.44e-04	L(1.00)	-26.8	1.34e-06	L(1.00)

Figure 12: Welch's t-test and Vargha-Delaney effect size calculated for the results of every model and class. "-" means that both result sets were constant.

6 Discussion

In this section we discuss the results of our study and answer the main research question formulated in Section 4.

All three models were able to improve the mutation score to a varying degree. Across all 12 classes, the models were able to increase mutation score in 4 to 6 cases depending on the model. The measured increases range from as low as 5 percentage point as can be seen in Figures 3 and 13 to as high as 51 percentage point as shown in Figures 5 and 18.

Interestingly, the difference in size between the models does not appear to have a large impact on the performance. This suggests that the ability of a model to write unit tests for a particular program is not closely related to its amount of parameters, or that this relation requires more extreme values than the range of 6.7 to 22 billion parameters (say, 0.1B or 70B).

As can be seen in Figures 3, 4 and 5, some classes have results in a large range. This can be attributed to a model not being able to generate a passing or compiling test, which can cause either no improvement, or a 0% mutation score.

A good example of a class where the models were not able to generate any passing tests is `AbstractNFeAdaptadorBean`. Upon further inspection, we discover that this class is written in Spanish. LLMs can be thrown off by semantic-preserving changes [36], such as changing the name of a variable [14], as shown in various studies [5, 43, 45]. Because the models are most likely mainly trained on code written in English, we believe the same behaviour is displayed in our study. A similar argument can be made for the `Distance` class, which represents genomic library mate distances instead of euclidean distances or some other more conventional distance metric, possibly confusing the model.

We also notice that the classes from the Apache Commons project appear to perform better than the classes from the SF110 project, which is in line with previous research [38]. For example the `ArrayFill` class already has 100% mutation score with only the manually written tests, but even the generated tests by themselves can reach this score as can be seen in Appendix A Figures 13, 15 and 17. The models were not able to achieve this score on any other class.

Another observation is that comments appear to be relevant relevant to the performance of the LLM. Classes without comments such as `UnsyncBufferedInputStream` perform significantly worse than classes with comments such as `StringUtils`.

The runtime of the models seems to be fairly consistent for most classes. Classes where the range of runtimes is larger are classes where the model is not often able to produce working classes. This causes the system to re-run the test suite multiple times until all problematic tests have been removed, increasing the total runtime.

7 Threats to Validity

The prompts used for generating tests might not be optimal. Because of the black-box nature of LLMs, one can not known which prompt will yield the best results. Because the quality of the model output is heavily dependent on the quality of the prompt [42], this can have an impact on the results. Even though prompt engineering is outside the scope of this research, we performed preliminary testing to arrive at a prompt which yielded good results.

The test corpus used for evaluation might not be representative of the real world. Because testing all existing Java classes is not possible, we used a limited set of classes. This means that the test corpus might not generalise to other classes. To minimise this risk, we attempted to make the corpus as varied as possible, using classes of varying complexity, language, and application. Additionally, we have used performance evaluation metrics which have been shown to be suitable for this type of research [6]. We used the Vargha-Delaney $\hat{A}_{12}$ effect size [40] and Welch's t-test to determine the relevance of our results.

The classes may be part of the training set of the model, artificially increasing performance. Previous research has shown that models perform better on classes which are part of the training set [38]. This is also in line with the results of our study. We believe however, that the SF110 corpus of classes [20] is most likely not part of this training set [36]. Therefore, we assembled a benchmark that contains mostly classes from SF110 to minimise this risk.

8 Responsible Research

In this section we reflect on the ethical aspects of this research, and discuss the reproducibility of the proposed system.

8.1 Resources

The resources used for this project are all publicly available. The classes under test were mostly taken from the SF110 corpus [20], and a few from Apache Commons. The SF110 was specifically chosen because it is most likely not part of the training set of the used models. We know that public code on GitHub is part of the training set of these models,

which can give an unfair advantage [38]. We used our own personal hardware for running the experiments, no sponsorship or other hardware was provided.

8.2 Reproducibility

The code required for reproducing this research is available in a public repository on GitHub. This repository contains the prompts, source code, classes under test, and code for calculating the performance of the models. For running the models, an OpenAI compatible API server running the model is used. For this research we used the built-in Local Server of LM Studio through a proxy (also available in aforementioned repository) to provide some basic authentication.

Because LLMs are innately nondeterministic, the results obtained by performing this research may not be exactly the same. In order to make the research as reproducible as possible, we ran the program multiple times for every class under test.

9 Conclusions and Future Work

The models were often able to improve the mutation score, being able to do so in 4 to 6 out of 12 cases depending on the model. For most classes classes, the models were also able to reliably produce passing and compiling tests. We believe the success rate of a model can be attributed to a few factors. Some classes have very brief or no comments, which seemed to negatively impact the model performance. The language of the class also seems to be important, as a class that is not written in English (`AbstractNFeAdaptadorBean` is written in Spanish) performed very poorly across all models. This suggests that the LLMs perform better if they are able to also understand the intention of the code instead of only the semantic and syntactic meaning. The runtime of the models did not seem to be very significantly different depending on what class is being tested. Finally, in line with previous research [38], classes that are part of the training set appear to perform better than classes which are not.

Future research could focus on (i) more extensive prompt engineering, which has been shown to be able to yield significant improvements of the effectiveness of these models [42], and (ii) exploring different models such as Microsoft's Phi [25] or OpenAI's Codex [12].

10 Acknowledgements

I would like to express my gratitude to my supervisors Dr. Annibale Panichella and Dr. Mitchell Olsthoorn for their guidance and helpful feedback during the writing of this thesis. In addition, I would also like to thank my fellow research group members Adomas Bagdonas, Arda Turhan, Reinier Schep and Stefan Creastă for their helpful discussions and ideas.

A Detailed results from all runs of the system

This section contains a detailed result of all runs of the system with mutation scores per model, per clas, per run. If the generated tests ("Generated only" column) have a score of 0%, this means no passing tests were generated. If the combined tests ("Combined score" column) also have a score of 0%, this means no compiling tests were generated.

Classname	Mutants	Baseline	Run	Combined score	Generated only
Util	24	83%	1	88%	38%
			2	88%	38%
			3	88%	38%
			4	83%	33%
			5	83%	33%
			6	88%	38%
ArrayFill	24	100%	1	100%	100%
			2	100%	100%
			3	100%	88%
			4	100%	100%
			5	100%	29%
			6	100%	100%
Distance	49	73%	1	73%	0%
			2	73%	0%
			3	78%	10%
			4	78%	14%
			5	73%	0%
			6	73%	0%
IlluminaUtil	33	67%	1	79%	21%
			2	85%	27%
			3	85%	18%
			4	67%	0%
			5	82%	15%
			6	79%	12%
Vector	97	64%	1	64%	0%
			2	79%	55%
			3	66%	21%
			4	79%	46%
			5	68%	13%
			6	71%	46%
AbstractNFeAdaptadorBean	16	94%	1	94%	6%
			2	94%	6%
			3	94%	0%
			4	94%	31%
			5	0%	0%
			6	94%	6%

Continued on next page in Figure 14

Figure 13: Deepseek Coder 6.7B mutation scores per class over 6 runs per class. Both with (combined) and without (generated) manually written tests. X/0: no passing tests generated. 0/0: no compiling test generated.

Classname	Mutants	Baseline	Run	Combined score	Generated only
StringUtils	67	36%	1	52%	21%
			2	52%	21%
			3	76%	61%
			4	61%	25%
			5	51%	19%
			6	61%	30%
UnsyncBufferedInputStream	95	77%	1	77%	2%
			2	0%	0%
			3	0%	0%
			4	77%	2%
			5	77%	4%
			6	77%	3%
PrefixParser	97	71%	1	72%	3%
			2	71%	0%
			3	72%	2%
			4	72%	2%
			5	71%	0%
			6	71%	0%
DateTime	40	50%	1	90%	40%
			2	50%	0%
			3	90%	40%
			4	90%	40%
			5	50%	0%
			6	50%	0%
StringMap	19	58%	1	58%	11%
			2	58%	5%
			3	58%	42%
			4	58%	42%
			5	63%	32%
			6	79%	79%
OpMatcher	29	48%	1	93%	72%
			2	69%	48%
			3	97%	72%
			4	90%	69%
			5	93%	72%
			6	0%	0%

Figure 14: Continuation of Deepseek Coder 6.7B mutation scores per class over 6 runs per class. Both with (combined) and without (generated) manually written tests. X/0: no passing tests generated. 0/0: no compiling test generated.

Classname	Mutants	Baseline	Run	Combined score	Generated only
Util	24	83%	1	100%	92%
			2	96%	79%
			3	96%	88%
			4	96%	79%
			5	96%	88%
			6	96%	88%
ArrayFill	24	100%	1	100%	67%
			2	100%	50%
			3	100%	100%
			4	100%	100%
			5	100%	67%
			6	100%	67%
Distance	49	73%	1	73%	0%
			2	73%	0%
			3	78%	6%
			4	73%	0%
			5	73%	6%
			6	78%	6%
IlluminaUtil	33	67%	1	79%	48%
			2	73%	33%
			3	73%	36%
			4	73%	64%
			5	73%	33%
			6	73%	6%
Vector	97	64%	1	64%	9%
			2	64%	9%
			3	90%	70%
			4	78%	45%
			5	90%	70%
			6	66%	10%
AbstractNFeAdaptadorBean	16	94%	1	94%	0%
			2	94%	0%
			3	94%	0%
			4	94%	0%
			5	94%	0%
			6	94%	0%

Continued on next page in Figure 16

Figure 15: Code Llama 13B mutation scores per class over 6 runs per class. Both with (combined) and without (generated) manually written tests. X/0: no passing tests generated. 0/0: no compiling test generated.

Classname	Mutants	Baseline	Run	Combined score	Generated only
StringUtils	67	36%	1	52%	16%
			2	0%	0%
			3	0%	0%
			4	52%	37%
			5	58%	43%
			6	0%	0%
UnsyncBufferedInputStream	95	77%	1	77%	26%
			2	77%	0%
			3	77%	32%
			4	77%	0%
			5	77%	31%
			6	77%	32%
PrefixParser	97	71%	1	71%	0%
			2	71%	0%
			3	71%	0%
			4	71%	0%
			5	71%	0%
			6	71%	0%
DateTime	40	50%	1	88%	38%
			2	88%	38%
			3	82%	32%
			4	88%	38%
			5	82%	32%
			6	90%	40%
StringMap	19	58%	1	63%	63%
			2	58%	0%
			3	79%	79%
			4	58%	58%
			5	58%	37%
			6	58%	58%
OpMatcher	29	48%	1	72%	48%
			2	90%	66%
			3	79%	55%
			4	97%	72%
			5	90%	41%
			6	93%	69%

Figure 16: Continuation of Code Llama 13B mutation scores per class over 6 runs per class. Both with (combined) and without (generated) manually written tests. X/0: no passing tests generated. 0/0: no compiling test generated.

Classname	Mutants	Baseline	Run	Combined score	Generated only
Util	24	83%	1	96%	83%
			2	100%	88%
			3	96%	62%
			4	100%	88%
			5	96%	62%
			6	96%	58%
ArrayFill	24	100%	1	100%	100%
			2	100%	100%
			3	100%	100%
			4	100%	100%
			5	100%	100%
			6	100%	100%
Distance	49	73%	1	0%	0%
			2	73%	0%
			3	0%	0%
			4	0%	0%
			5	0%	0%
			6	0%	0%
IlluminaUtil	33	67%	1	85%	85%
			2	82%	21%
			3	82%	30%
			4	85%	85%
			5	88%	73%
			6	94%	94%
Vector	97	64%	1	64%	0%
			2	64%	0%
			3	65%	21%
			4	64%	0%
			5	64%	0%
			6	79%	44%
AbstractNFeAdaptadorBean	16	94%	1	0%	0%
			2	0%	0%
			3	94%	6%
			4	94%	6%
			5	0%	0%
			6	94%	6%

Continued on next page in Figure 18

Figure 17: Codestral 22B mutation scores per class over 6 runs per class. Both with (combined) and without (generated) manually written tests. X/0: no passing tests generated. 0/0: no compiling test generated.

Classname	Mutants	Baseline	Run	Combined score	Generated only
StringUtils	67	36%	1	72%	40%
			2	72%	61%
			3	87%	76%
			4	84%	52%
			5	84%	70%
			6	61%	30%
UnsyncBufferedInputStream	95	77%	1	77%	0%
			2	77%	0%
			3	77%	0%
			4	77%	0%
			5	77%	0%
			6	77%	0%
PrefixParser	97	71%	1	0%	0%
			2	71%	1%
			3	74%	24%
			4	71%	35%
			5	73%	23%
			6	72%	3%
DateTime	40	50%	1	50%	5%
			2	50%	5%
			3	50%	20%
			4	52%	8%
			5	52%	8%
			6	52%	8%
StringMap	19	58%	1	58%	0%
			2	58%	11%
			3	58%	0%
			4	58%	0%
			5	58%	0%
			6	63%	47%
OpMatcher	29	48%	1	86%	62%
			2	86%	62%
			3	93%	72%
			4	93%	72%
			5	93%	72%
			6	86%	62%

Figure 18: Continuation of Codestral 22B mutation scores per class over 6 runs per class. Both with (combined) and without (generated) manually written tests. X/0: no passing tests generated. 0/0: no compiling test generated.

References

[1] Mistral AI. Codestral: Hello, World! https://mistral.ai/news/codestral/, May 2024.

[2] M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In *2017 IEEE/ACM 39th International Conference on Software Engi-*

neering: Software Engineering in Practice Track (ICSE-SEIP), pages 263–272, May 2017.

[3] Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated Unit Test Improvement using Large Language Models at Meta, February 2024.

[4] Paul Ammann and Jeff Offutt. Introduction to Software Testing. https://www.cambridge.org/highereducation/books/introduction-to-software-testing/95E57CCADEA697EC8594F03729F47311, December 2016.

[5] Leonhard Applis, Annibale Panichella, and Ruben Marang. Searching for Quality: Genetic Algorithms and Metamorphic Testing for Software Engineering ML. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '23, pages 1490–1498, New York, NY, USA, July 2023. Association for Computing Machinery.

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

[7] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program Synthesis with Large Language Models, August 2021.

[8] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015.

[9] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. Call Me Maybe: Using NLP to Automatically Generate Unit Test Cases Respecting Temporal Constraints. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, pages 1–11, New York, NY, USA, January 2023. Association for Computing Machinery.

[10] Xia Cai and Michael R. Lyu. The effect of code coverage on fault detection under different testing profiles. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, May 2005.

[11] Mariano Ceccato, Alessandro Marchetto, Leonardo Mariani, Cu D. Nguyen, and Paolo Tonella. Do Automatically Generated Test Cases Make Debugging Easier? An Experimental Assessment of Debugging Effectiveness and Efficiency. *ACM Transactions on Software Engineering and Methodology*, 25(1):5:1–5:38, December 2015.

[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford,

Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code, July 2021.

[13] L.A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, September 1976.

[14] Rhys Compton, Eibe Frank, Panos Patros, and Abigail Koay. Embedding Java Classes with code2vec: Improvements from Variable Obfuscation. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 243–253, June 2020.

[15] Ermira Daka and Gordon Fraser. A Survey on Unit Testing Practices and Problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211, November 2014.

[16] 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. *Information and Software Technology*, 171, 2024.

[17] Nobel Dhar, Bobin Deng, Dan Lo, Xiaofeng Wu, Liang Zhao, and Kun Suo. An Empirical Analysis and Resource Footprint Study of Deploying Large Language Models on Edge Devices. In *Proceedings of the 2024 ACM Southeast Conference*, ACM SE '24, pages 69–76, New York, NY, USA, April 2024. Association for Computing Machinery.

[18] M. Dubiel, Y. Barghouti, K. Kudryavtseva, and L.A. Leiva. On-device query intent prediction with lightweight LLMs to support ubiquitous conversations. *Scientific reports*, 14(1):12731, 2024.

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

[20] Gordon Fraser and Andrea Arcuri. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Transactions on Software Engineering and Methodology*, 24(2):8:1–8:42, December 2014.

[21] Gregory Gay. Improving the Readability of Generated Tests Using GPT-4 and ChatGPT Code Interpreter. In Paolo Arcaini, Tao Yue, and Erik M. Fredericks, editors, *Search-Based Software Engineering*, pages 140–146, Cham, 2024. Springer Nature Switzerland.

[22] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 72–82, New York, NY, USA, May 2014. Association for Computing Machinery.

[23] Giovanni Grano, Simone Scalabrino, Harald C. Gall, and Rocco Oliveto. An Empirical Investigation on the Readability of Manual and Generated Test Cases. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 348–3483, May 2018.

[24] Vitor Guilherme and Auri Vincenzi. An initial investigation of ChatGPT unit test generation capability. In *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing*, SAST '23, pages 15–24, New York, NY, USA, October 2023. Association for Computing Machinery.

[25] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks Are All You Need, October 2023.

[26] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence, January 2024.

[27] Mark Harman and Peter O'Hearn. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–23, September 2018.

[28] Hadi Hemmati. How Effective Are Code Coverage Criteria? In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 151–156, August 2015.

[29] Gunel Jahangirova and Valerio Terragni. SBFT Tool Competition 2023 - Java Test Case Generation Track. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, pages 61–64, May 2023.

[30] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 643–653, New York, NY, USA, November 2014. Association for Computing Machinery.

[31] T.J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[32] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[33] A. Jefferson Offutt and Stephen D. Lee. How strong is weak mutation? In *Proceedings of the Symposium on Testing, Analysis, and Verification*, TAV4, pages 200–213, New York, NY, USA, October 1991. Association for Computing Machinery.

[34] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, New York, NY, USA, October 2007. Association for Computing Machinery.

[35] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo

Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open Foundation Models for Code, January 2024.

[36] June Sallou, Thomas Durieux, and Annibale Panichella. 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*, pages 102–106, April 2024.

[37] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation, December 2023.

[38] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. Using Large Language Models to Generate JUnit Tests: An Empirical Study, March 2024.

[39] Frank Tip, Jonathan Bell, and Max Schäfer. LLMorpheus: Mutation Testing using Large Language Models, April 2024.

[40] András Vargha and Harold D. Delaney. A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[41] Yuling Wang, Changxin Tian, Binbin Hu, Yanhua Yu, Ziqi Liu, Zhiqiang Zhang, Jun Zhou, Liang Pang, and Xiao Wang. Can Small Language Models be Good Reasoners for Sequential Recommendation? In *Proceedings of the ACM on Web Conference 2024*, WWW '24, pages 3876–3887, New York, NY, USA, May 2024. Association for Computing Machinery.

[42] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT, February 2023.

[43] Zhou Yang, Jieke Shi, Junda He, and David Lo. Natural Attack for Pre-trained Models of Code. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1482–1493, May 2022.

[44] Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Zhibo Sun, and Yue Zhang. A Survey on Large Language Model (LLM) Security and Privacy: The Good, The Bad, and The Ugly. *High-Confidence Computing*, 4(2):100211, June 2024.

[45] Noam Yefet, Uri Alon, and Eran Yahav. Adversarial Examples for Models of Code, October 2020.