Exploring Test Suite Coverage of Large Language Model–Enhanced Unit Test Generation

A Study on the Ability of Large Language Models to Improve the Understandability of Generated Unit Tests Without Compromising Coverage

Andrei Drăgoi[1]

Supervisors: Andy Zaidman[1]**, Amirhossein Deljouyi**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: Andrei Drăgoi
Final project course: CSE3000 Research Project
Thesis committee: Andy Zaidman, Amirhossein Deljouyi, Asterios Katsifodimos

Abstract

Automated software testing is a frequently studied topic in specialized literature. Search-based software testing tools, like EvoSuite, can generate test suites using genetic algorithms without the developer's input. Large Language Models (LLMs) have recently attracted significant attention in the software engineering domain for their potential to automate test generation. UTGen, a tool integrating LLMs with EvoSuite, produces more understandable tests than EvoSuite; however, the generated tests suffer a coverage drop.

To streamline bug detection by developers, we propose *UTGenCov*, a concept that focuses on improving the understandability of EvoSuite-generated tests without compromising on coverage. This approach builds upon UTGen by thoroughly analyzing the reasons behind the decrease in coverage and proposing an alternative approach.

Our investigation determined that the leading cause of coverage reduction in UTGen is LLM hallucination in the Understandability phase. UTGenCov aims to address hallucinations by providing the source code of the methods used in the test to the LLM. Yet, our experiment results indicate inconsistent performance and a further decrease in branch coverage of 0.74% compared to UTGen.

1 Introduction

Effective software testing is an integral part of the software development lifecycle. Search-based software testing (SBST) tools, such as EvoSuite [1], use generic algorithms to generate test suites that can achieve adequate coverage [2]. The growing popularity of Large Language Models (LLMs) is becoming increasingly evident, and researchers are actively pursuing techniques to automate test generation with the help of LLMs [3; 4]. Nonetheless, LLMs struggle to reach high coverage, and SBST produces tests that are not very readable [5].

To the best of our knowledge, limited research has been conducted on integrating SBST with LLMs to produce more understandable tests that can achieve high coverage. This research extends the work conducted by Deljouyi [6] on combining LLMs with SBST. This combined approach, which uses EvoSuite as a foundation, is called UTGen, for which the paper has not yet been published. However, the average branch coverage of the test suite generated by UTGen is 1% lower than EvoSuite's.

The goal of this research is to investigate and address the deficiencies in UTGen's achieved coverage, thereby developing automated test generation capable of producing meaningful tests with high coverage while remaining understandable and usable to developers.

Thus, our key research objective is to identify the causes behind coverage shortages in LLM-guided SBST compared to conventional SBST and, thereafter, address these shortfalls. This central objective is divided into the following research questions:

- **RQ**$_1$ Which of UTGen's phases impact the test suite coverage?
- **RQ**$_2$ How can the factors contributing to inferior coverage in UTGen be mitigated?

Our key contribution was identifying several critical areas negatively affecting the coverage of UTGen's generated tests as a result of two individual studies looking at eleven Java projects. Moreover, with a new proposed approach, *UTGenCov*, we deployed and evaluated a grounding technique [7] aimed to address these factors.

2 Background and Related Work

Literature serves as a valuable starting point for our research. Fan et al. [8] conducted a comprehensive literature survey outlining different use cases of LLMs in software engineering. When examining how LLMs can be used for software testing, one of the surveyed studies reports an increase in coverage of up to 80% [9], although "neither the studied LLMs could achieve more than 2% coverage on the EvoSuite SF110 dataset", which is the basis for our evaluation.

2.1 Search-based software testing

Search-based software testing (SBST) has proven to be a robust technique in software quality assurance by using genetic algorithms to generate test cases that achieve high code coverage and reveal faults [10; 11; 12].

One notable tool in this field is EvoSuite, an automated unit test generation framework for Java [1]. EvoSuite optimizes the generation of test cases by using evolutionary algorithms to maximize code coverage and minimize the number of test cases. This tool has been extensively evaluated, outperforming other testing methods in coverage metrics [13].

2.2 Large Language Models

Large Language Models (LLMs) like OpenAI's GPT and Meta's Code Llama are remodeling software testing practices. These models, trained on vast amounts of text data, show complex natural language understanding capabilities, making them suitable for generating and improving test cases [14].

Researchers are exploring the integration of LLMs with SBST to automate and enhance the test generation process [15]. While there have been promising advancements, there are still challenges in effectively integrating LLMs with SBST. One concern is ensuring that the improved tests do not negatively impact the generated test suite, such as reducing coverage or causing tests not to compile [5].

2.3 UTGen

UTGen combines EvoSuite with LLMs to improve the understandability of the generated test cases [6]. The LLM utilized by UTGen is Meta's Code Llama 7b Instruct, specifically trained for instruction-following code-related tasks [16].

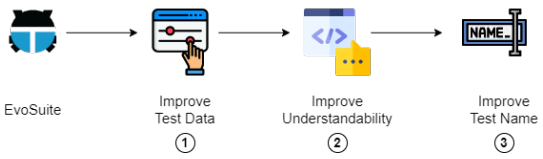

Figure 1: Overview of the stages of UTGen.

Figure 1 illustrates the phases that UTGen undergoes:

- EvoSuite uses a generic algorithm to produce a test suite for a given Java class.
- UTGen's Test Data improvement step (**1** in Figure 1) enhances test data, such as the parameters (e.g., `Strings`) used in method calls.
- UTGen's Understandability improvement phase (**2** in Figure 1) focuses on adding descriptive comments and improving variable names.
- UTGen's Test Name improvement stage (**3** in Figure 1) aims to give a descriptive name to each test.

3 UTGen Analysis *(RQ1)*

The initial phase of the research investigated the factors that negatively affect the coverage of the generated test suite in UTGen. This section will discuss the setup and results of our investigation.

3.1 Study setup

The dataset used as a basis for our analysis is a subset of EvoSuite's SF110 [17]. The subset was chosen based on the results of the UTGen experiments. In particular, classes that manifested a decrease in coverage in UTGen compared to EvoSuite were included in our investigations.

Table 1: List of projects from the SF110 dataset used in the Manual inspection and Phase isolation studies, and the difference in branch coverage between EvoSuite and UTGen.

Project	Branch coverage difference (in %)	Used in study
12_dsachat	1.3	Both
13_jdbacl	8.2	Manual inspection
15_beanbin	6.4	Manual inspection
17_inspireto	8.3	Manual inspection
19_jmca	0.2	Manual inspection
26_jipa	7.5	Both
33_javaviewcontrol	0.4	Manual inspection
45_lotus	26.6[1]	Both
60_sugar	3.3	Manual inspection
68_biblestudy	16.7	Manual inspection
93_quickserver	1.5	Both
101_netweaver	5.7	Phase Isolation

To understand the reasons for the decrease in coverage, we conducted two studies: Phase isolation and Manual inspection. Multiple classes were used in the investigations. Table 1 shows the classes used in both studies and the branch coverage difference between EvoSuite and UTGen. We will now discuss the setup of each study individually.

[1]For this project, we considered the instruction coverage; the branch coverage was slightly higher in UTGen than EvoSuite.

A) Phase isolation

As a first step, we investigated which of UTGen's phases affects coverage. To achieve this, we ran each phase of UTGen in isolation. Figure 2 details the setup: for each class under test, we ran EvoSuite once and used the generated tests as a basis in four independent runs of UTGen: only Test Data, only Understandability, only Test Name, and full UTGen. We then analyzed the coverage of each approach while also comparing it with EvoSuite as a baseline.

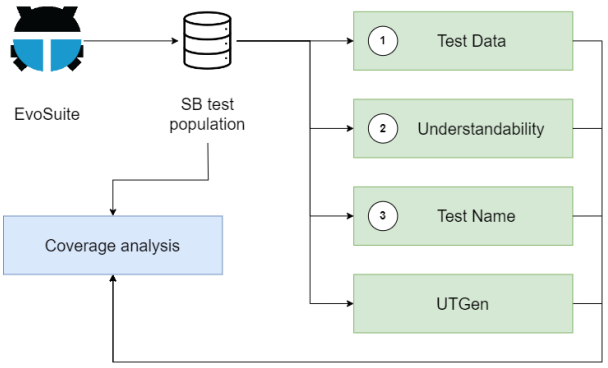

Figure 2: Study setup for running each of UTGen's phases in isolation, using the same test population as the foundation.

B) Manual inspection

To further understand the causes of the coverage drop, we manually compared the test suites generated in the initial UTGen experiment. These test suites were created for numerous projects from the SF110 dataset, preserving the EvoSuite- and UTGen-generated tests. Using JaCoCo reports as a starting point, we identified the methods and constructors with lower coverage in UTGen. We then looked for the EvoSuite-generated tests that used these methods or constructors, searched for similar test cases generated by UTGen that were missing them, and finally classified the changes into several categories based on the reason believed to cause the coverage drop.

3.2 Results

We will now discuss the results of each investigation individually and provide a summary of the UTGen analysis.

A) Phase isolation

Of the five projects in this stage, project 45 exemplified deviant behavior. The coverage increased when running Test Data and Understandability in isolation but decreased when running full UTGen. Upon manual inspection, we noticed a line had been removed from one test in both isolated runs of UTGen for the mentioned phases. The removal of this line covered an edge case, which led to increased coverage. For this reason, the aggregated results presented below exclude this project.

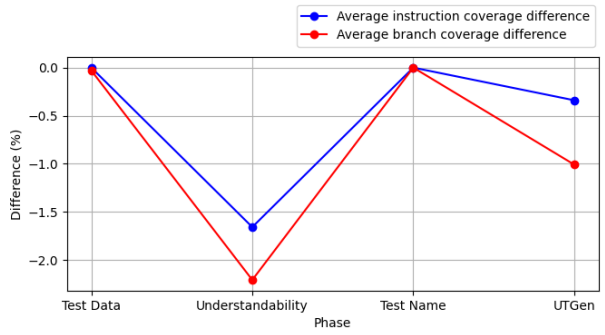

Figure 3: Coverage difference between EvoSuite and each phase of UTGen: Test Data, Understandability, Test Name, and full UTGen.

Figure 3 highlights that the biggest negative impact on coverage is caused by the Understandability phase of UTGen, with a 2.2% decrease in branch coverage compared to EvoSuite. The Test Data phase has a minor impact on the coverage, with a decrease of 0.03% in branch coverage. Lastly, the Test Name phase does not impact the test suite coverage.

B) Manual inspection

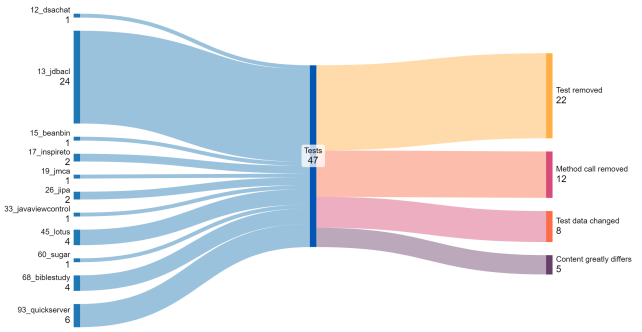

Figure 4: Manual inspection results: The 47 tests, part of 11 projects, identified to have a negative impact on coverage were categorized into groups based on the cause of the coverage drop.

Figure 4 shows the results of the manual inspection of the tests generated during UTGen's experiment. Out of all inspected tests, 47 of them from 11 projects had a negative impact on coverage. These tests have been classified according to the cause of the coverage drop. We categorized them based on the following criteria:

- **Test removed**: one of EvoSuite's phases (running both before and after UTGen) removes tests that are unstable and do not compile. 47% of the time, a coverage drop is generated by removed test cases.
- **Method call removed**: The test improved by UTGen was missing one or more method calls or constructors. About 25% of the flagged tests presented this behavior. An example of such a test can be seen in listing 1, where UTGen's Understandability phase added the comments on lines 3-4 and 9-10, but line 8 was removed compared to the EvoSuite test case.
- **Test data changed**: Parameters used for method calls are changed by UTGen, leading to a corner case no longer being covered by the test. These tests counted for about 17% of test cases. Listing 2 provides an example where line 3 was replaced by line 4. The missing nested quotes in the new String no longer covered the corner case.
- **Content greatly differs**: In this group, test logic was significantly changed in UTGen compared to EvoSuite; namely, different objects were instantiated, and assertions were different. 11% of test cases exemplified this behavior.

```
1  @Test
2  public void testRun1() throws Throwable  {
3    // Create a new instance of BlockingClientHandler
4    // and add an event to it
5      BlockingClientHandler blockingClientHandler = new
       ↪  BlockingClientHandler();
6      ClientEvent arg0 = ClientEvent.READ;
7      blockingClientHandler.addEvent(arg0);
8      blockingClientHandler.run();
9    // Run the handler and verify that it will
10   // clean up the resources
11     assertTrue(blockingClientHandler.getWillClean());
12 }
```

Listing 1: Diff showing a test generated by EvoSuite compared to the UTGen-improved version. The comments, highlighted in green, were added, and the line highlighted in red was removed by UTGen.

```
1 @Test
2 public void testProcessInstruction1() throws Throwable  {
3     Main.processInstruction("qBSfJjW\"b 7\"QIe,*");
4     Main.processInstruction("jump to next element");
5 }
```

Listing 2: Diff showing a test generated by EvoSuite compared to the UTGen-improved version. The string used as a parameter for the method call was changed.

In summary, from the two studies on the behavior of UTGen on 11 projects, we can draw the following conclusions: from the Manual inspection results, we can assess that the biggest factors negatively impacting coverage are tests, or lines from tests, being removed while the LLM attempts to improve understandability. As for the Phase isolation study, the Understandability phase, which replaces the entire test body, is shown to cause the biggest difference in coverage.

4 Factor mitigation *(RQ2)*

Based on the findings of *RQ1*, we constructed UTGenCov, an extension of UTGen. In this section, we will discuss this approach in detail and present the experiment setup and the results of our evaluation.

4.1 The UTGenCov Approach

To improve the coverage of the tests generated by UTGen, we employed a grounding technique [7], which aims to limit LLM hallucinations. Specifically, in the LLM prompt, we provided the source code of the methods called in the test to be improved.

```
<<SYS>>
  You are a Java developer optimizing JUnit tests to increase
  ↪  clarity.
<</SYS>>
Your task is to make a previously written JUnit test more
↪  understandable without changing test behavior. The returned
↪  understandable test must be between the [TEST] and [/TEST]
↪  tags.
Add comments to the code which explain what is happening and the
↪  intentions of what is being done. Only change variable names
↪  to make them more relevant, leaving the test data untouched.
Overall, it is the goal to have a more concise test that is both
↪  descriptive as well as relevant to the context.
The previously written test to improve is between the [CODE] and
↪  [/CODE] tags.
The source code is between the [SOURCE] and [/SOURCE] tags.
[CODE]
  The test code to be improved.
[/CODE]
[SOURCE]
  The source code of all methods used in the test above.
[/SOURCE]
```

Listing 3: Improved LLM prompt that provides the source code and the test. The changes to the original (UTGen) prompt are highlighted in light blue.

Based on the results of *RQ1*, we have decided to deploy this grounding strategy only in the Understandability phase of UTGen and not in the Test Data phase, which only had a marginal impact on coverage. The updated prompt can be seen in listing 3, where the addition is highlighted.

4.2 Experiment setup

To assess the impact of the changes outlined in section 4.1, we have set up an experiment where we look at the coverage of EvoSuite, compared to UTGen's Understandability phase ran in isolation and the new UTGenCov approach. To ensure the experiment was not affected by the other phases of UTGen, we disabled the Test Data and Test Name phases when running UTGenCov. Similar to the setup of *RQ1*, we use the same SB test population as the foundation for the two independent runs, as presented in Figure 5.

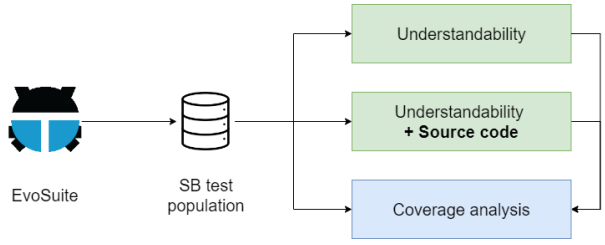

Figure 5: Experiment setup for comparing UTGen's Understandability phase with the improved approach from section 4.1, using the same test population as the foundation.

We attempted to run the experiment on all classes we identified as susceptible to a coverage drop from the Manual inspection phase. However, some classes resulted in EvoSuite errors and were removed. The classes used for this experiment are listed in Table 2.

Table 2: Classes selected for the experiment of *RQ2*

Project	Class
12_dsachat	Handler
17_inspirento	XmlElement
19_jmca	JMCAAnalyzer
26_jipa	Main
45_lotus	Game
68_biblestudy	Queue
93_quickserver	NonBlockingClientHandler
	BlockingClientHandler

4.3 Results

We will now analyze the results of our experiment per project. Figure 6 shows the difference in branch coverage between UTGen's Understandability phase and EvoSuite and between UTGenCov and EvoSuite. In Table 3, we look at the number of generated tests in each experiment, which will be used for further analysis of the results.

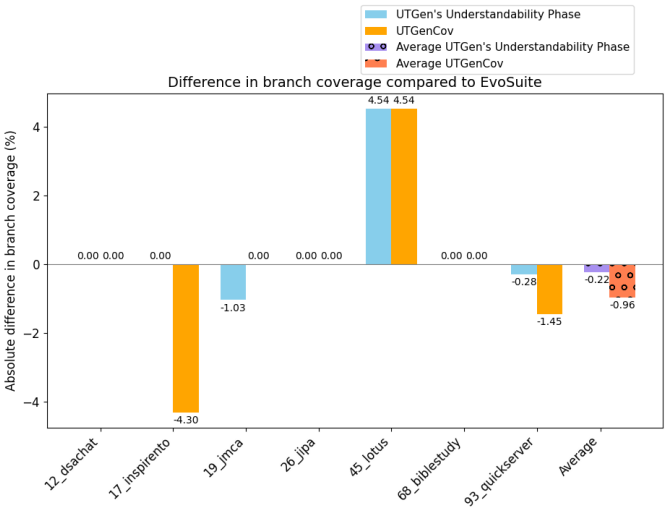

Figure 6: Branch coverage difference between UTGen's Understandability phase and UTGenCov compared to EvoSuite for all projects in the experiment

Table 3: Number of tests generated, tests with no comments added, and tests rolled back for each approach.

Project	Run type	# tests	# tests with no comments added	# tests rolled back
12_dsachat	EvoSuite	1	-	-
	Understandability	1	0	1
	UTGenCov	1	0	1
17_inspirento	EvoSuite	62	-	-
	Understandability	64	3	5
	UTGenCov	62	7	6
19_jmca	EvoSuite	4	-	-
	Understandability	4	0	1
	UTGenCov	4	1	1

Table continues on next page

Continuation of table

Project	Run type	# tests	# tests with no comments added	# tests rolled back
26_jipa	EvoSuite	60	-	-
	Understandability	61	28	1
	UTGenCov	61	28	1
45_lotus	EvoSuite	8	-	-
	Understandability	6	4	0
	UTGenCov	6	4	0
68_biblestudy	EvoSuite	17	-	-
	Understandability	17	17	0
	UTGenCov	17	17	0
93_quickserver	EvoSuite	66	-	-
	Understandability	66	40	0
	UTGenCov	66	11	2

For the classes from projects 12, 26, and 68, the addition of source code had no impact on the generated test and, therefore, no impact on the coverage. For project 12, EvoSuite generated one test. In both approaches, the improved test was rolled back to EvoSuite. This happens if the LLM-improved test fails to compile multiple times. In project 26, both UTGen and UTGenCov produced identical test suites, with 28 out of the 61 tests having no comments added. This behavior happens when the LLM fails to improve the test case, meaning that the LLM response is either in the incorrect format or the LLM call times out. For project 68, all 17 tests in both approaches concluded without added comments.

One class, Game, part of project 45, showed a coverage increase in both the UTGen and UTGenCov generated tests compared to EvoSuite. Upon manual analysis of the generated tests, we observed that both UTGen and UTGenCov presented the same behavior: one line of code was removed from the original test case, leading to an edge case being covered. It is worth noting that out of the eight tests that EvoSuite generated, both UTGen and UTGenCov produced only six tests, of which four had no comments added. Because of the coverage increase, project 45 is excluded from the average in Figure 6.

For project 19, the branch coverage of the UTGen tests decreased by 1% compared to EvoSuite, while UTGenCov had the same coverage as EvoSuite. While this is a favorable result, it should be noted that from the total of four tests generated by EvoSuite, both approaches resulted in one rolled back test, and UTGenCov had one test with no comments added, while UTGen did not, meaning that the improved prompt, with source code, lead to too many re-prompts.

UTGenCov performed worse regarding coverage in both projects 17 and 93. In project 17, branch coverage decreased by 4.3%, from 98.7% in EvoSuite to 94.3% in UTGenCov, while UTGen coverage was the same as EvoSuite. For the two classes from project 93, branch coverage was 1.2% lower in UTGenCov compared to UTGen. In 60% of UTGen's tests, no comments were added, while only 17% of UTGenCov's tests manifested this behavior. This means that while UTGenCov successfully improved the understandability of more tests, the improved tests' coverage was inferior.

In summary, the experiment showed mixed results across projects. Adding source code had no impact on projects 12, 26, 68, and 45 compared to UTGen. On project 19, UTGenCov had identical coverage to EvoSuite, but it could have been caused by a test that failed to be improved. However, UTGenCov had reduced coverage in projects 17 and 93 compared to EvoSuite and UTGen. Although UTGenCov successfully enhanced test understandability more frequently than UTGen, it resulted in, on average, 0.74% lower branch coverage than UTGen.

5 Discussion

In this section, we will discuss our results further, examine this research's limitations, and assess the threats to validity.

5.1 Revisiting the Research Questions

RQ1: Which of UTGen's phases impact the test suite coverage? Based on the Phase isolation study, we established that the Understandability phase is the primary cause of the coverage drop. These results align with our expectations based on the prompts given to the LLM for each phase:

- **Test Data**: In this phase, the LLM is tasked to "Improve the test data by changing the primitive values and Strings" and is only provided with one line of code; thus, it is improbable that the line is removed altogether or that hallucinations could affect it.
- **Understandability**: For this stage of UTGen, the entire method body is replaced with the improved code as a result of prompting the LLM with the following: "Your task is to make a previously written JUnit test more understandable". This prompt gives the LLM a lot of liberty in what it should do, which makes it more prone to hallucination errors.
- **Test name**: For this phase, only the test name is changed, not the method body, which makes it impossible for the phase to affect the coverage.

Furthermore, the Manual inspection has categorized the affected test cases into several groups. We observed that tests being removed were the most common cause of coverage drop, followed by changes in test content. Since a direct mapping between each inspected test and how much it affected the coverage is impossible, the impact was estimated based on the number of tests exemplifying the behavior.

It is noteworthy that from the Manual inspection, five of the tests exhibited clear signs of hallucination. This was evident from the presence of method calls not existent in the given test case or in the method under test, the latter not being provided to the LLM in any case.

However, the underlying cause remains uncertain. We hypothesize that all issues stem from hallucinations within the LLM, potentially arising from insufficient contextual information or the inherent constraints of the chosen LLM model. To this extent, *RQ2* aims to address hallucinations by providing more context to the LLM via the method bodies of the methods used by the test.

RQ2: How can the factors contributing to inferior coverage in UTGen be mitigated? Our chosen approach to mitigating UTGen's coverage shortfall, likely generated by hallucination, was to use a grounding technique, more precisely, adding the source code of the methods under test to the LLM prompt. While theoretically promising, this approach had no positive impact on coverage in most classes used in our experiment. Thus, as the result variation indicates, including the source code in the prompt may not always be beneficial.

We hypothesize that the selected LLM, Code Llama 7b Instruct, hallucinates due to inherent limitations, such as an insufficient context window (16k tokens) or model size (with only 7 billion parameters). A potential solution includes employing a different LLM with increased capabilities, such as Codestral[2], which features 22 billion parameters and a 32k token window, or GPT-4o[3], which has 175 billion parameters and a 128k context window. Alternatively, integrating advanced grounding techniques like Retrieval Augmented Generation (RAG) [18] could help with hallucinations.

5.2 Limitations

The scope of our research was limited by the short time frame and resource constraints. In the Phase isolation study, running one class took between two and six hours due to the substantial resource requirements of the LLM. Similarly, the Manual inspection phase required considerable time, particularly for test files containing over 80 tests that required analysis. These limitations could affect the findings by not presenting a sufficiently comprehensive overview.

5.3 Threats to validity

We will discuss two types of threats to the validity of this research: external validity, which looks into the generalizability of our research, and internal validity, which addresses other potential factors that can impact the outcomes of our research.

External validity An important threat to research is its generalizability. Given the nine-week project timeline, we attempted to mitigate this threat to the best of our ability. The main concern here is the size of the dataset used for our investigations. The number of selected classes is low compared to the size of the SF110 dataset used in the initial UTGen experiment. Nonetheless, they have been manually chosen to have the highest coverage difference, thus producing results relevant to our research questions.

We should likewise consider the temporal validity of this research. With the rapid advancements in the field of LLMs, it is important to consider that current research may become outdated as LLMs with superior capabilities become available. One such advancement is Codestral[2], a newly released LLM advertised to "outperform all other models in RepoBench, a long-range eval for code generation" [19].

Internal validity One principal factor that could have impacted our results is the nondeterminism of EvoSuite's genetic algorithm and of the LLM.

[2]https://mistral.ai/news/codestral/

[3]https://openai.com/index/hello-gpt-4o/

To mitigate this threat in EvoSuite, we used the same seed for all experiments, and for the Phase isolation study, as well as the UTGenCov experiment, the same test population was used as a basis for all runs on one class, as presented in Figures 2 and 5.

However, the nondeterminism of the LLM could not be mitigated. Sometimes, the LLM does not fully comply with the instructions, leading to an incorrectly formatted answer that UTGen cannot parse. This leads to a re-prompt of the LLM. If the LLM were deterministic, the answer would always be in an incorrect format, making improvements impossible.

6 Responsible Research

In this section, we will discuss the reproducibility of our research, examine the privacy and ethical implications of using LLMs, and address the use of generative AI for the project.

6.1 Research reproducibility

First, we should address the dataset used for this research, SF110, a publicly available dataset containing 110 open-source Java projects from SourceForge [17]. For the scope of our analysis, we have selected a subset of projects that exhibited the most significant reduction in coverage from EvoSuite to UTGen based on the results of the UTGen experiments. The paper about UTGen has not yet been published; however, it will include a replication package containing all experiment results, thus making the project selection reproducible. We can now discuss the reproducibility of each undertaking separately.

Phase isolation For this study, we adapted UTGen to run each phase separately. Although implementation details are deliberately not provided, section 3.1 provides a thorough description of the behavior of the revised approach using the same EvoSuite test population as the basis for all isolated runs of UTGen. This description offers substantial detail to ensure that the adaptation can be replicated.

Manual inspection As with the previous study, we describe the method used for this study in section 3.1 and further explain the criteria for the categorization used, along with examples in section 3.2. Of course, we must acknowledge that this is still a *manual* inspection, which may be subjective or biased. To the best of our ability, we sought to be objective and consistent for all inspected test suites. We deem sufficient detail was provided to ensure the reproducibility of this study.

UTGenCov experiment For this experiment, we adapted UTGen to include the source code of the methods used in the test case to be improved by the LLM. Section 4.1 explains the modified approach, which we call UTGenCov. Section 4.2 contains the classes used for the experiment and the experiment setup, where the same EvoSuite test population was used for both runs. Therefore, sufficient information about the setup is offered to make this experiment reproducible.

6.2 Privacy and ethical implications

For privacy, UTGenCov never stores user data or the class under test. However, the LLM is often deployed on a dedicated service, requiring data transfer to an outside party, potentially

with unclear or unfavorable privacy practices. To prevent any privacy concerns, users can run both the tool and the LLM locally, ensuring the confidentiality of their data.

One ethical factor is ensuring the quality and safety of the generated test cases. While attempting to enhance understandability, the LLM might generate code that does not follow good coding practices or overlooks security vulnerabilities. The developer is always responsible for performing thorough security reviews and code quality checks to mitigate such risks.

Another aspect we should consider is the potential misuse of the tool. Although UTGenCov aims to help developers detect bugs in their code by automatically generating understandable tests, it does not mean it could not be misused to create misleading tests. For instance, through a backdoor attack, the LLM might generate deceptive code comments or tests that could introduce vulnerabilities or malicious behaviors. To prevent this issue, the utilized LLM, Meta's Code Llama, is open source and allows developers to verify its behavior.

6.3 Use of AI

Generative AI tools, such as ChatGPT and Grammarly AI, were used to facilitate this research. In this section, we will discuss their usage.

Grammarly was used as a writing aid for proofreading. Grammarly automatically analyzes the text and suggests improvements. Additionally, its AI feature allows tone improvement with prompts such as "Improve it" and "Make it sound formal". Grammarly AI does not generate new ideas and was never used on more than one sentence at a time. Additionally, all responses were checked and modified accordingly.

ChatGPT was used for data analysis, error fixing, and creating visualizations. A list of prompt types can be found in Appendix A. For data analysis, all content produced by ChatGPT was manually verified for accuracy and correctness before it was used in the research process. ChatGPT was never used to produce new ideas, and its textual responses were never used verbatim.

7 Conclusion

This research investigated two key questions in the domain of automated software testing: which phases of UTGen, a tool combining Large Language Models with search-based software testing, impact test suite coverage, and how the factors contributing to inferior coverage in UTGen can be mitigated. Our study revealed that the Understandability phase of UTGen, which uses LLMs to improve test readability, is the primary cause of coverage reduction. This decline is primarily attributed to LLM hallucinations, where the model generates incorrect, incomplete, or irrelevant code.

To address this issue, we developed UTGenCov, an extension of UTGen that provides source code as context to the LLM during the Understandability phase. This grounding technique aims to reduce hallucinations and maintain coverage while improving test understandability. Our experiment with UTGenCov across multiple Java projects yielded mixed results. While it maintained or slightly improved coverage in some cases, it led to decreased coverage in others. On average, UTGenCov resulted in a 0.74% lower branch coverage than UTGen.

Our key contributions include analyzing coverage impacts in LLM-enhanced test generation, developing and evaluating UTGenCov, and highlighting the complexity of leveraging LLMs in software testing. While LLMs show potential in test understandability improvements, maintaining high code coverage remains a significant challenge. Our findings suggest that simply providing more context to LLMs may not be sufficient to mitigate coverage issues consistently.

These results emphasize the need for more refined techniques in LLM-guided test generation. Future work should consider experiments with a broader scope that use more projects to help strengthen the results and potentially reveal new insights. Additionally, an analysis of the impact on coverage of using different LLMs, such as Code Llama 70b Instruct or Codestral, should be performed. Given the inherent size difference of the LLMs mentioned earlier compared to the one currently used by UTGen and UTGenCov (Code Llama 7b Instruct), they could help with hallucinations and provide more understandable tests.

A LLM Prompts

The list below provides an overview of the types of prompts used for ChatGPT. Although this is not an exhaustive list, we covered all prompt templates. We did not include follow-up prompts in the list, as those only contained further information or questions related to the same topic.

- Please help me to format $\langle data \rangle$ in LaTeX.
- How can I fix this LaTeX/Python/Maven error: $\langle error \rangle$?
- Given this $\langle data \rangle$, can you perform an in-depth analysis? Please focus on $\langle request \rangle$ in your analysis.
- Can you please explain this piece of code: $\langle code \rangle$?
- I want to visualize this $\langle data \rangle$. Please provide the Python code to create a visualization for it.
- I am writing a section on $\langle context \rangle$ for my research paper. Is this structure logical?: $\langle structure \rangle$ How would you adjust it? Only provide a bullet point list of ideas.

References

[1] G. Fraser and A. 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*, 2011, pp. 416–419.

[2] ——, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, pp. 1–42, 2014.

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

and E. Wang, "Automated unit test improvement using large language models at meta," *arXiv preprint arXiv:2402.09171*, 2024.

[4] 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*, vol. 171, p. 107468, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584924000739

[5] Y. Tang, Z. Liu, Z. Zhou, and X. Luo, "Chatgpt vs sbst: A comparative assessment of unit test suite generation," *IEEE Transactions on Software Engineering*, 2024.

[6] A. Deljouyi, "Understandable test generation through capture/replay and llms (icse 2024 - doctoral symposium) - icse 2024."

[7] A. Addlesee, "Grounding LLMs to in-prompt instructions: Reducing hallucinations caused by static pre-training knowledge," in *Proceedings of Safety4ConvAI: The Third Workshop on Safety for Conversational AI @ LREC-COLING 2024*, T. Dinkar, G. Attanasio, A. C. Curry, I. Konstas, D. Hovy, and V. Rieser, Eds. Torino, Italia: ELRA and ICCL, May 2024, pp. 1–7. [Online]. Available: https://aclanthology.org/2024.safety4convai-1.1

[8] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," 2023, Conference paper, p. 31 – 53, all Open Access, Green Open Access. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85185604518&doi=10.1109%2fICSE-FoSE59343.2023.00008&partnerID=40&md5=8940cca3aaa53cdab79b08a8ef247214

[9] S. Hashtroudi, J. Shin, H. Hemmati, and S. Wang, "Automated test case generation using code models and domain adaptation," *arXiv preprint arXiv:2308.08033*, 2023.

[10] M. Harman *et al.*, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, pp. 1–61, 2012.

[11] P. McMinn, "Search-based software test data generation: a survey," *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[12] A. Panichella *et al.*, "Automated test case generation as a many-objective optimisation problem," in *Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering*. IEEE, 2018, pp. 123–133.

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

[14] S. Minaee, T. Mikolov, N. Nikzad, M. Chenaghlu, R. Socher, X. Amatriain, and J. Gao, "Large language models: A survey," 2024.

[15] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," 2023, Conference paper, p. 919 – 931. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85168753555&doi=10.1109%2fICSE48619.2023.00085&partnerID=40&md5=4ac3b83ac2ab896fa76d07bd76fb82dc

[16] Meta AI, "Introducing Code Llama, a state-of-the-art large language model for coding," 8 2023. [Online]. Available: https://ai.meta.com/blog/code-llama-large-language-model-coding/

[17] EvoSuite, "SF110 Corpus of Classes — EvoSuite." [Online]. Available: https://www.evosuite.org/experimental-data/sf110/

[18] Z. Jiang, F. F. Xu, L. Gao, Z. Sun, Q. Liu, J. Dwivedi-Yu, Y. Yang, J. Callan, and G. Neubig, "Active retrieval augmented generation," 2023.

[19] Mistral AI, "Codestral: Hello, world!" 6 2024. [Online]. Available: https://mistral.ai/news/codestral/