Reducing LLM Hallucinations with Retrieval Prompt Engineering

Minimising the Need for Re-prompting in Automatic Understandable Test Generation

Angelika Mentzelopoulou[1]

Supervisors: Andy Zaidman[1]**, Amir 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: Angelika Mentzelopoulou
Final project course: CSE3000 Research Project
Thesis committee: Andy Zaidman, Amir Deljouyi, Asterios Katsifodimos

Abstract

Automated test generation is the means to produce correct and usable code while maintaining an efficient and effective development process. UTGen is a tool that utilizes a Large Language Model (LLM) to improve the understandability of a test suite generated by a Search-Based Software Testing tool, namely EvoSuite. Often while the LLM attempts to improve a given test case, it generates code that is too far from the original, changing the test's purpose. Alternatively, it may generate code that does not compile. Such behaviour is called "LLM Hallucination".

The current hallucination handling of UTGen is time-consuming and resource-expensive. To address this, we propose two alternative approaches that use information retrieval prompt engineering techniques to minimise hallucinations. Our respective techniques include incorporating the source code under test and the errors thrown by the latest generated test case to the LLM prompt. We assess our methods through a comparison study against the base UTGen version. We observe that source code retrieval enhances the generation of compilable test cases for complex classes. Error code retrieval shows similar hallucination performance to base UTGen, with a decrease in the number of re-prompts for classes with a high normalised Lack of Cohesion of Methods (*LCOM).

Index Terms - Automated Test Generation, Large Language Models (LLMs), LLM Hallucination, Prompt Engineering

1 Introduction

Thorough software testing is necessary to produce correct code that fits a client's requirements, is usable and maintainable code [1]. The automation of test generation aims for an efficient and effective test development process [2]. Search-Based Software Testing (SBST) tools are one of the main test generation tool types, that focus on creating a suite that achieves a high coverage [3]. A common issue amongst such tools is the limitation of the generated tests' understandability [4]; there is no focus by the tools on generating human-intuitive test data, test method names and code comments. To resolve that issue while maintaining the high coverage and efficiency advantages of SBST, UTGen [1] was developed. UTGen combines an SBST tool, namely EvoSuite [5], and Large Language Models (LLMs) to enhance the understandability of the automatically generated test cases in the three factors mentioned above.

While UTGen successfully improved the understandability of many of the tests compared to the ones generated by EvoSuite, the LLM usage to accomplish that is ineffective. The involvement of the LLM slightly decreased the overall coverage achieved by the test suite. Additionally, the LLM often "hallucinated" throughout the test generation process by producing invalid responses. Hallucination is a larger challenge within language generation models and refers to generations that are "nonsensical or unfaithful to the provided source content" [6]. Specifically in the UTGen context, it produced code that did not compile, or it altered the context of the EvoSuite-generated test too much, changing its purpose in the test suite. UTGen includes verification processes for each area, where re-prompting currently handles LLM hallucinations. There is a maximum amount of iterations for which each verification process can be repeated. If that maximum amount is reached and no valid response has been generated, UTGen uses the original EvoSuite test for the result, defeating its purpose of improving understandability.

Figure 1 showcases this process and the result for two tests generated by UTGen that were discarded. On the left-hand side example, even though the understandability was greatly improved by the explicit comments added by the LLM, the test needed to be reverted due to the fact that the 'Robot' object constructor missed one argument. Respectively, the right-hand side showcases the same for a generated test that was too different to the original EvoSuite test. Again here, comments have been added, but one method call is missing, dropping the similarity score between the Evosuite-generated and the LLM-improved test cases significantly. In both cases, the LLM has already been prompted to the maximum amount allowed for each respective process, and therefore, time and expenses have been wasted without achieving any improvement.

As said, handling hallucinations with re-prompting leads to a time-consuming and expensive test generation process, as every step may be repeated multiple times. In this research, we focus on decreasing the need for re-prompting to handle hallucinations by grounding the LLM using prompt engineering to generate contextful test cases. Prompt engineering involves developing effective prompting techniques that enable foundation models to solve specific tasks better [7]. Specifically, we are investigating the change in performance using information retrieval methods, where knowledge from the input is extracted and inserted in the prompt to the LLM. We focus on the Post-Processing stage of UTGen, where the LLM is prompted to add understandable identifiers and comments to the test case. That is the only phase where the test code's structure may be altered, and therefore the mentioned hallucinations may occur.

In this project, we explore the following research questions:

1. *RQ1: Is it possible to reduce the hallucination of LLMs used in UTGen during the Post-Processing phase, to minimise the need for re-prompting, using prompt engineering with source code retrieval?*
2. *RQ2: Is it possible to reduce the hallucination of LLMs used in UTGen during the Post-Processing phase, to minimise the need for re-prompting, using prompt engineering with error code retrieval?*

The key contributions of our paper are outlined as follows:

- Our approach that integrates the source code to the LLM

[1] A paper on UTGen has not yet been published.

Figure 1: Examples of processes resulting in a reverted test and enhancement stagnation

prompt UTGen utilises to enhance the understandability of generated unit tests.

- Our approach that integrates the error codes produced by the latest generated test to the LLM prompt.
- The application of our two approaches on 3 classes that have shown ineffectiveness in improving understandability, to examine the understandability and effectiveness of the generated unit tests.
- A comparison study between the UTGen prompt and the prompts used by our approaches to improve understandability on the same base Evosuite-generated unit tests.
- We release a publicly available package with our implementation and experiment [8].

2 Background & Related Work

In this section, we provide background information on the term "LLM hallucinations" and how UTGen functions. We also briefly compare how our approach compares to relevant other initiatives for reducing the need for re-prompting LLMs in automatic test generation.

2.1 LLM Hallucinations

Outside of LLMs, hallucination is a psychological term, as defined by Blom [9] as "a percept, experienced by a waking individual, in the absence of an appropriate stimulus from the extracorporeal world", an unreal phenomenon that feels real. Similarly, in the context of LLMs used for code generation, hallucinated code may appear syntactically valid, despite using inexistent variables or methods.

LLM hallucinations for code generation arise from various fundamental principles of LLM functionality [10]. The causes can be rooted in an incomplete, biased or flawed dataset used to train the model and the training architecture itself [11]. Yet, hallucinations can also be caused by a prompt that gives false information or a prompt phrased with informal language [12]. This variety and especially the fact that hallucinations may be independent of training, indicate that hallucination cannot be directly prevented; but the chances of it occurring may be decreased.

Existing research explores different avenues to decrease the possibility of an LLM hallucinating; by utilising software-specific insights for fine-tuning and training [13], or using grounding techniques. Grounding involves providing the LLM with additional information during generation time. The provided information is related only to the specific prompt and is retrieved accordingly, possibly from source code, API references, or databases [14; 15].

2.2 UTGen

UTGen combines search-based software testing, namely EvosSuite [5], and LLMs to enhance the understandability of automatically generated test cases.

UTGen is an enhancement built on top of EvoSuite. The input to the tool is comprised of the source code files of the classes we desire to generate tests for, the same as for the base version of EvoSuite. The tests are generated by EvoSuite using search-based techniques. After the EvoSuite test suite is available, UTGen incorporates the LLM to improve its understandability in three steps, which are also illustrated in Figure 2:

1. **Test Data Improvement:** UTGen prompts the LLM to generate new test data that is more understandable. It uses an enhanced parser of the LLM output to ensure that each line that would cause an error in the new test case is replaced by the original line.

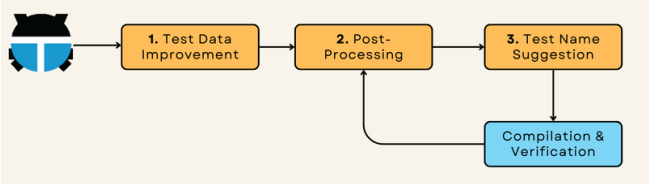


Figure 2: Overview of UTGen's test generation process

2. **Post Processing: Enhancement of Variable Identifiers and Comments** After the test data has been finalised, UTGen moves to the Post-Processing phase. The modifications on the code from now onwards aim purely to improve understandability and clarity, without altering the functionality of the test. To ensure the purpose of the test remains unchanged, the CodeBLEU metric is used. CodeBLEU effectively assesses syntactic and semantic similarities between two sequences [16].
3. **Test Method Name:** Finally, the focus of UTGen is on generating an understandable test method name. The LLM is re-prompted until it produces a name unique to the test suite.

Hallucinations within UTGen appear in two main forms; the LLM may produce code that does not compile, e.g., add extra parameters to a method call. Alternatively, it may generate lines that deviate from the original test case changing the test's purpose. UTGen's original approach does not particularly attempt to decrease hallucinations, but does include hallucination handling. Specifically, after a test case is generated, UTGen attempts to compile it. If any issues are raised there, the LLM is re-prompted till a valid case is generated. There is a budget for the maximum amount of re-prompts for a test case; if it is reached, the test case is discarded. The same process is followed when the generated test by the LLM deviates from the EvoSuite test above a given threshold, evaluated by the enhanced parser or CodeBLEU according to the generation phase.

2.3 Retrieval Prompt Engineering for Code Generation

With Retrieval Prompt Engineering, we refer to any prompt engineering technique where, according to the task to be executed, the prompt is adapted by including relevant information retrieved from a resource. The additional information may aid in grounding the LLM. Said resource may vary, as different such grounding techniques may make use of a database or previous LLM responses. In our case, the data is retrieved from the source code to be tested and the execution results of the tests last generated by UTGen.

Eghbali and Pradel [14] introduced De-Hallucinator, a code completion technique that iteratively augments the prompt by including API references in decreasing relevance to the code to be completed. Contrary to our techniques, the prompt here is augmented instead of altered after every failed attempt of the LLM to generate the code. Additionally, a point to consider is that De-Hallucinator executes a completion task, while UTGen generates new independent pieces of code.

Nashid et al. introduce CEDAR [15], a systematic approach to retrieve the most effective code examples from a pool, to include as part of a few-shot engineered prompt, that follows the notion of learning from the desired task description, along with a few examples [17]. Similarly, Parvez et al. [18] suggest REDCODER, a framework that incorporates related code to the prompt for code generation and summarization tasks. Both papers suggest methods that require a database of code, while our suggestions retrieve information only from the current code generation task to enhance the prompts to the LLM.

3 The Proposed Approaches

The three-stage communication bridge with the LLM raises the need for re-prompting, often due to hallucinations. This makes the process more time-consuming and expensive. In this research, we focus on the Post-Processing phase, specifically in the enhancement of variable identifiers and comments. This interaction phase is the only stage that may cause a test to be reverted or stagnate enhancement and, therefore, cause the severest drawbacks to understandability. The rationale behind this is that it represents the sole phase during which the structure of EvoSuite test code can be modified, unlike the test data and test method name phases.

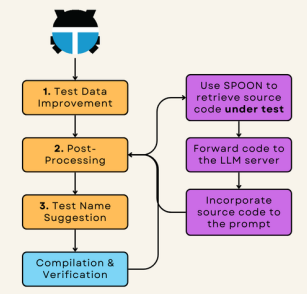
3.1 Source code retrieval

We investigated whether incorporating source code in the prompt can reduce the hallucination of LLMs in UTGen. When researching what information we can retrieve for the prompt, source code stood out due to its consistent reliability and availability. Additionally, source code is inherently relevant to the domain under test, as it is used to execute it.

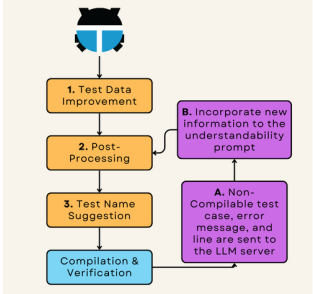
The implementation was built on top of the latest UTGen version. We retrieve the source code of all methods the unit test generated by EvoSuite is calling, from the class under test. Those methods are forwarded to the LLM server. For the method code extraction, we used the source code analysis library for Java 'SPOON' [19]. In the LLM server, the methods are incorporated into the prompt simply by stating that the source code of the methods under test is available and placing it between `[SRC_CODE]/[/SRC_CODE]` tags. These tags adhere to the existing prompt format for the Post-Processing phase of UTGen. This integration process is illustrated in Figure 3a.

Listing 1 showcases the template for the understandability prompt of the base version of UTGen. In red is the adjustment made to the base UTGen prompt to include the relevant source code:

```
<<SYS>>
   You are a Java developer optimizing JUnit tests for clarity.
<</SYS>>
Your task is to make a previously written JUnit test more understandable.
The returned understandable test must be between the [TEST] and [/TEST] tags
Add comments with the Given, When, Then Structure 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 which 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 snippet being tested is between the [SRC_CODE] and [/SRC_CODE
      ] tags.
[CODE]
```



(a) Source Code



(b) Errors from latest test case

Figure 3: Overview of UTGen incorporating information to the understandability prompt

```
   (TEST CODE)
[/CODE]
[SRC_CODE]
   (SOURCE CODE OF RELATED METHOD 1)
   (SOURCE CODE OF RELATED METHOD 2)
   ...
[/SRC_CODE]
```

Listing 1: Source Code Retrieval Approach Prompt

Error code retrieval We explored the effect of incorporating test compilation error codes to the understandability prompt on the hallucination of LLMs in UTGen. This research was motivated by the phenomenon that a large part of the hallucinations source from the generation of non-compilable code. The main idea of our approach is to decrease the number of re-prompts needed to make the code compile by providing the error codes thrown by the previously generated test case to the LLM and asking it to fix those directly.

Figure 3b showcases the overview of our approach. After a generated test case fails to compile during UTGen's 'Compilation & Verification' phase, the diagnostics from the compilation tool are gathered. All diagnostics contain an error message, a source file at which the error was thrown, as well as the line number where the error was thrown. We use the source file and the line number to retrieve the exact code from the test case that threw the error. The message, the erroneous line of code, as well as the whole generated test case, are then sent to the LLM server, which in turn incorporates them into the prompt. The final prompt is shown in Listing 2, where we emphasize the main changes from the base prompt in red.

```
<<SYS>>
   You are a Java developer optimizing JUnit tests for clarity.
<</SYS>>
This is the prompt you received earlier in this role:
"""
Your task is to make a previously written JUnit test more understandable. 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.
Overall, it is the goal to have a more descriptive test.
"""
Unfortunately, the test you previously generated raised some errors.
The test case you previously generated is between the [CODE] and [/CODE] tags
      .
The errors and the lines of code they were raised in are between the [ERRORS]
       and [/ERRORS] tags.
With that information, try again, **while fulfilling the task requirements
      from your initial prompt**
Tip: if there is a "cannot find symbol" error, then remove that line.
Add comments with the Given, When, Then Structure 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 which is both descriptive
       as well as relevant to the context.
[CODE]
   (PREVIOUSLY GENERATED TEST CODE)
[/CODE]
[ERRORS]
   (ERROR 1)
   (ERROR 2)
   ...
[/ERRORS]
```

Listing 2: Error Code Retrieval Approach Prompt

4 Experimental Setup

In this section, we describe the methodology of evaluation of our approach. We investigated the following research questions:

***RQ*$_1$** Is it possible to reduce the hallucination of LLMs used in UTGen during the Post-Processing phase, to minimise the need for re-prompting, using prompt engineering with source code retrieval?

***RQ*$_2$** Is it possible to reduce the hallucination of LLMs used in UTGen during the Post-Processing phase, to minimise the need for re-prompting, using prompt engineering with error code retrieval?

The experiment setup was similar for both questions, while the same metrics were measured to answer each question.

4.1 Hallucination Metrics

To identify and measure LLM hallucination during UTGen's test generation process for a given class under test, we defined the following metrics:

Number of re-prompts: The number of times the LLM needed to be re-prompted. This is identifiable through the log files of the LLM server. Specifically, we count the number of times the comment *"Trying again with the same prompt"* is logged by the server.

Enhancement Stagnation rate: During the Post-Processing phase we are investigating, part of the LLM's role is to add comments to the test code. If the LLM reaches the maximum number of re-prompts allowed by UTGen without generating a test that is similar to the EvoSuite test case according to CodeBLEU, the comment *"no comments added"* is added to the original test case instead. Therefore, for the Enhancement Stagnation rate, we measure the percentage of tests that contain the mentioned replacement comment in their code.

Reverted Tests rate Similarly to Enhancement Stagnation, there exists handling by UTGen for when the LLM fails to produce compilable code within the set amount of prompts allowed for the respective verification. In that case, the test in the resulting test suite is rolled back to the original EvoSuite test case, and UTGen adds the comment *"rollbacked to EvoSuite"* to it. For the Reverted Tests rate metric, we, therefore, measure the percentage of tests containing the comment mentioned in their code.

Index	Project	Class
1	51_jiprof	MethodWriter
2	86_at-robots2-j	Robot
3	51_jiprof	ClassReader
	# Re-Prompts	# Enhancement Stagnation
1	9	14
2	40	6
3	80	14
	# Reverted Tests	# Tests
1	35	86
2	6	61
3	11	34

Table 1: Hallucination-Relevant Metrics of the High-Hallucination Performing Classes

4.2 LLM

For our experiments, the prompt component of UTGen uses the `code-llama:7b-instruct model` from Meta [20] as provided by Hugging Face [2]. We selected this model because the base version of UTGen also uses it; this allows for a more reliable comparison with the benchmarks. Contrary to the benchmarks, we decided to source and run it on Hugging Face since it would allow faster execution of our experiments' iterations.

4.3 Dataset

UTGen developers utilized the DynaMOSA dataset composed of 346 non-trivial Java classes from 117 open-source projects [21]. The classes are selected from four different benchmarks, with the primary source being the 204 non-trivial classes of SF110 [22]. We use the resulting test suites and logs of UTGen's execution on these projects as our benchmarks for measuring LLM hallucination.

4.4 Subjects

Three classes were deemed the ideal number of classes to run our experiment on, considering the tight time constraints of our project. We selected our classes from two categories: classes with many reverted tests and tests to which enhancement stagnation occurred, therefore many problematic tests, and classes with a low rate of the former two types of tests. For the measurements required for the selection, the benchmarks available from the evaluation of UTGen were used. An overview of the chosen classes and their values for the hallucination metrics are shown in Table 1.

Firstly we wanted to select two classes with a high number of problematic test cases in the benchmarks. The reason we prioritised this selection metric was the opportunity to investigate the application of our approaches to a wider range. This allows us to observe to what extent our approaches are consistent and not context-specific. To select the exact classes, we ran a Python script that measured the number of *reverted tests* and *enhancement stagnation tests* of all available classes from the benchmarks. The mean and standard deviation of each of the metrics were then calculated, and we isolated the classes in the third high standard deviation for both metrics. This left us with four classes. To avoid prioritising either metric, we randomly selected two classes from that set: `MethodWriter` and `ClassReader`.

For our third experiment class, we selected one with low rates of problematic tests compared to the total number of tests in its test suite. The motivation for investigating this class was to assess whether our approaches could lower the understandability performance of classes with good benchmark results. Using a Python script, we measure the total amount of tests and calculate the enhancement stagnation and reverted test rates. By narrowing down the classes to only ones with low rates of problematic test cases, we identified ones with a relatively high number of total tests. The reason was again to have a larger range on which to test our approaches. Out of the resulting set, the `Robot` class was selected.

4.5 Experimental Procedure

The experimental procedure specific to each RQ is described in this section.

RQ1 We apply two versions of UTGen to our dataset: the base version of UTGen originally developed and the one that includes relevant source code to the LLM prompt. The two versions are run on the same base EvoSuite-generated tests. This means that only after the LLM is involved in improving understandability do our two test suites begin to differ. It is important to note we do not run the test data improvement and test method name phases for either experiment. We then compare the two methods by measuring 1) the number of re-prompts, 2) the enhancement stagnation rate, and 3) the reverted test rate.

The same experiment is repeated thrice for each class for us to generate more objective and unbiased results. We ensure that the three iterations use distinct randomness seeds for EvoSuite's search-based test generation. To compare the hallucination performance between the two UTGen versions, we analogise the average metrics values for each class over all three iterations.

We also correlate these values to the code complexity of each class, specifically by measuring the Lines Of Code (LOC), Normalised Lack of Cohesion of Methods (*LOCM) and Weighed Method Count (WMC) from the CK metrics [23]. To measure the code complexity metrics for each class, we use the CK tool by Maurício Aniche [3].

RQ2 We apply two versions of UTGen to our dataset: the base version of UTGen originally developed and the one that includes the errors thrown by the latest generated test to the LLM prompt. The experimental process, setup and evaluation follow the exact same methods as for RQ1.

5 Results

In this section, we present and discuss our experiment results per research question.

[2] https://huggingface.co/

[3] https://github.com/mauricioaniche/ck

5.1 RQ1: Source Code Retrieval

Figure 4 showcases the average values of the metrics from three iterations of our experiment. The metrics are shown for base UTGen and for the version that includes the source code in the prompt, respectively. The aim of the research was to see a decrease from the base version to our approach for all of the number of re-prompts, reverted tests rate, and enhancement stagnation rate.

During the three iterations of our experiment, UTGen successfully generated a total of 1128 tests. Out of the total number of tests, 311 of them were reverted to the original EvoSuite test case, because the LLM did not achieve producing compilable code within the allowed amount of tries. Additionally, a total of 276 tests of the resulting suites did not have understandable comments, because the LLM did not manage to produce understandable code that was similar enough to the original EvoSuite test case, according to the CodeBLEU metric.

The rates of problematic test cases are not uniform amongst the three classes, as can be seen in Figure 4. Firstly, `ClassReader`, one of the two classes with high problematic test cases in the benchmarks, shows improvement in all three areas when using our source code approach, with the greatest decrease in the reverted tests rate, from almost 50% to 10%. `MethodWriter`, the second highly problematic test case class, has a decrease in reverted test rates but an increase in both other metrics. It presents the greatest change in the number of re-prompts, with our source code approach increasing that measure by an average count of 111 re-prompts. Finally, the `Robot` class, our third experiment class with a low rate of problematic test cases compared to the complete test suite in the benchmarks, shows an increase in all three metrics. Our approach increases LLM hallucinations for `Robot` in all three areas, with the highest average change appearing in the Enhancement Stagnation rate, with an increase of approximately 17%.

We also plot the correlation matrix between the increase in hallucinations produced by the LLM on a class according to the three metrics and the class' code complexity in Figure 5. We observe the strongest correlations with code complexity to appear with the Reverted Tests Rate. For classes with a high *LCOM, the base version of UTGen results in less reverted tests. Conversely, our approach decreases the reverted test rate for classes with high LOC or WMC. Nevertheless, for the same category of classes, the number of re-prompts is lower with the base version of UTGen. There are no strong positive or negative correlations concerning the increase in enhancement stagnation rate.

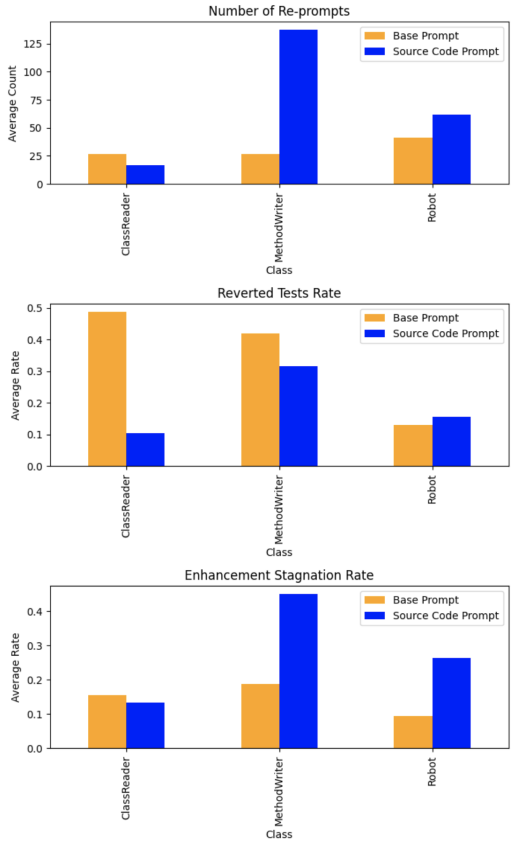


Figure 4: Comparison of Metrics between Base UTGen and Source Code retrieval version

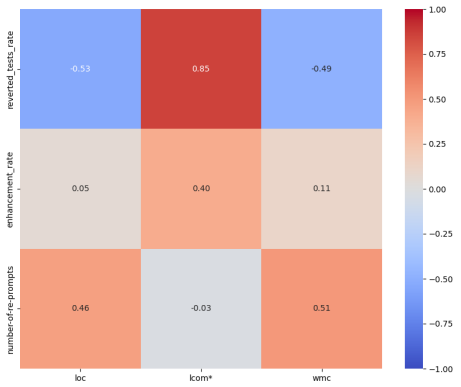


Figure 5: Correlation Matrix of Code Complexity and Increase in Average Hallucination Metrics

5.2 RQ2: Error Code Retrieval

Figure 6 showcases the average values of the metrics from three iterations of our experiment. We now compare base UTGen with the version that includes the error codes in the prompt in the case where the LLM produces non-compilable code. Here, we aim to see a decrease in the reverted test rate and the number of re-prompts since they are the metrics relevant to compilation.

During the three iterations of this experiment, UTGen successfully generated a total of 1138 tests. Of the total number of tests, 398 were reverted to the original EvoSuite test case, while a total of 118 tests of the resulting suites did not have understandable comments.

Our error code retrieval approach shows considerably more uniform results than when including source code in the prompt. Overall, `MethodWriter` and `ClassReader` show no or slight increase in all metrics of the average count of

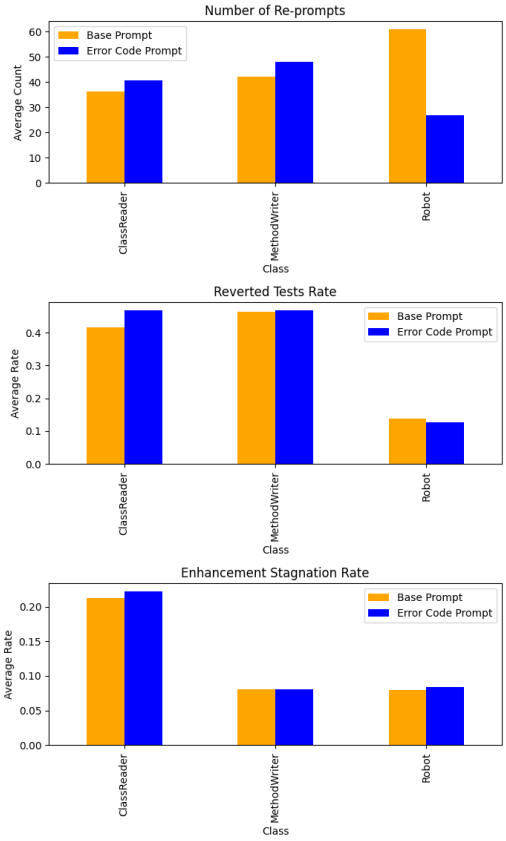


Figure 6: Comparison of Metrics between Base UTGen and Source Code retrieval version

re-prompts and average rates of reverted tests and enhancement stagnation. Nevertheless, with our approach, there are fewer hallucinations regarding the number of re-prompts and reverted test rate when generating tests for `Robot`, compared to base UTGen. Especially for the former, we see the greatest difference between our two approaches over our entire experiment; the average decrease in the number of re-prompts from base UTGen to our error code retrieval version for `Robot` is approximately 34.3%.

We correlate the increase in hallucinations the LLM encounters during generating tests for a given class and the class' code complexity in Figure 7. We observe strong correlations between class complexity and the increase in number of re-prompts, as our approach appears to benefit classes with a high *LCOM. The positive correlation between high WMC and LOC and the number of re-prompts indicates that base UTGen is the favoured version for such classes. The same correlations apply with the reverted test rate but to a weaker extent. Again, we observe no strong positive or negative correlations concerning the increase in enhancement stagnation rate.

6 Responsible Research

In this section, we discuss the ethical considerations taken into account while conducting our research. Additionally, we

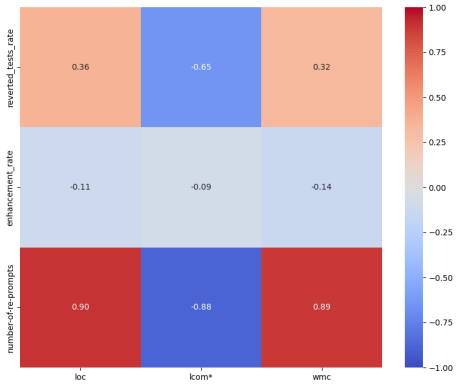


Figure 7: Correlation Matrix of Code Complexity and Increase in Average Hallucination Metrics

concern the possible ethical implications our approaches may introduce.

6.1 Reproducibility

For the reproducibility of our research, we provide a replication package which includes: the implementation of our two approaches (UTGen with source code and error code retrieval grounding), our resulting test cases and the server logs from their generation, and the scripts used to measure our metrics.

The replication package has been conducted adhering to the FAIR framework [24]. All information is *findable* since the data used for our experiments are publicly available as part of the DynaMOSA dataset [21]. The base UTGen-generated results used as our benchmarks are freely available as part of our replication package. Our implementation is *accessible* as we require no authentication for the accessing of our replication package. Relevant data is grouped in common folders with a common format, allowing for their easy interaction, making our data *interoperable*. Finally, we ensure the *reusability* of our package by formulating a clear folder structure with an understandable file naming system that allows for the identification of relevant data necessary for reuse.

6.2 Possible Ethical Implications

There are possible ethical implications concerning our research that should be taken into consideration.

LLM Training Set There may be bias sourced from the training set of the LLM. For that reason, we decided to conduct our measurements to evaluate our approaches with an open-source LLM.

Information Retrieval Retrieval techniques allow for the input of external information in the prompt used by UTGen to enhance test understandability. Especially in the source code retrieval approach, we must note that a malicious party can include malicious content in the prompt through the source code files and manipulate the LLM response.

Privacy All of the EvoSuite generated test cases, the source code and the error codes may comprise sensitive information

of UTGen's users. It shall be considered that any data that is part of the prompt is shared with the LLM.

7 Discussion

In this section, we discuss our results, possible future improvements, and the limitations faced during our research.

7.1 Revisiting the Research Questions

RQ1: *Is it possible to reduce the hallucination of LLMs used in UTGen during the Post-Processing phase, to minimise the need for re-prompting, using prompt engineering with source code retrieval?* When comparing our source code retrieval approach to the base version of UTGen, the change in hallucination varies. We observe that our approach decreases the reverted test rate for classes with low *LCOM. We hypothesize that supplying the LLM with source code directs its focus towards generating compilable code, thereby reducing the rate of reverted tests. Additionally, the enhancement stagnation rate increases for classes with different complexity and benchmark performance. We hypothesize that this phenomenon may result from the same shift in focus towards producing compilable code; the LLM prioritizes compilation over improving existing test cases. Consequently, the similarity scores by CodeBLEU decrease, resulting in high enhancement stagnation rates.

We analyzed the LLM's mentioned shift in focus towards producing compilable code and determined it may be related to the increased length of the input prompt. LLMs experience accuracy limitations with longer prompts because the ability to batch multiple examples from their training set and produce a response is constrained by limited available memory [25]. A possible avenue to explore is the switch to a model trained on a higher number of parameters, such as the `code-llama:70b-instruct model` [20]. Such models are likely to provide more appropriate responses to longer prompts due to their increased capacity. Additionally, we recommend further research on prompt engineering to assess the possibility of constructing a more concise prompt that includes source code while remaining focused on improving an initial given test case. A possible route to explore is limiting the provided source code to a single method.

RQ2: *Is it possible to reduce the hallucination of LLMs used in UTGen during the Post-Processing phase, to minimise the need for re-prompting, using prompt engineering with error code retrieval?* We compared base UTGen and our version that includes error codes in the prompt on their hallucination performance while attempting the understandability enhancement of the same EvoSuite-generated test suite. The experiment resulted in very similar average performance by the two UTGen versions, with the exception of our approach minimising the number of re-prompts for classes with high *LCOM, compared to base UTGen. We investigated the LLM server logs to see how it behaves after receiving the prompt with the previously generated test case and the compilation errors it generated. We observed that a common occurrence is for the LLM to mention in the text response that the issues have been resolved but still provide the erroneous test code.

We hypothesize that the issue arises from the high complexity of our prompt, which contains additional information beyond what the LLM directly requires to fix the generated test case. Specifically, we include information about the initial prompt to enhance the understandability of a given EvoSuite test case. We recommend that future researchers adopt our approach using a simplified prompt that solely requests the correction of a non-compilable test case based on the retrieved errors.

Another area of interest is examining the similarity between the final test suite generated by our approach and the initial EvoSuite test suite using CodeBLEU. In our implementation, the response to the prompt that includes the errors is not directly compared to the EvoSuite test case. Instead, we use CodeBLEU to compare the new response to the previous LLM-generated test case. This decision aims to maintain a concise prompt and aligns with the UTGen implementation, which checks CodeBLEU similarity on the LLM server. We therefore recommend further investigation to assess the effectiveness of the final test suite.

7.2 Limitations

Here, we discuss our general limitations while conducting our experiments and suggest how future researchers can avoid them.

Lack of Documentation UTGen is a tool built directly on the EvoSuite source code, comprising over 3000 Java source code files. Additionally, it lacks proper documentation for each functionality, making it challenging to integrate our approaches since the exact means of its functionality are not immediately transparent.

Time needed for Test Generation The initial approach to executing our experiments used the `code-llama:7b-instruct model` provided by Ollama [4]. The execution of a test suite generation for one Java class by UTGen with this model had an average duration of 8 hours on the available computer of a 2,3 GHz 8-Core Intel Core i9 processor. This led to the decision to narrow the experiments down to three classes.

Incompatibility of Ollama model with GPU Even though UTGen successfully completed its process on the available computer with the local Ollama version of the LLM model, the results generated by that configuration were deemed invalid. The reason for that is the incompatibility of the mentioned version of the model with the available computer's graphical processing unit (GPU), AMD Radeon Pro 5500M. Ollama could only run on the CPU, often failing to meet timeout constraints due to limited resources and eventually producing unrealistic results.

Financial Limitations The GPU incompatibility and the extensive time needed for execution led to our decision to use an external version of the model from Hugging Face. This reduced the time needed to generate a test suite for a single class to an average of 1 hour. The number of classes needed to remain at three due to the high financial cost of using Hugging Face and our limited resources.

[4] https://https://ollama.com/

Measuring Number of Re-Prompts The number of re-prompts is a crucial metric we employed in our research due to its relevance to our goal of reducing re-prompts. However, the manner in which UTGen's LLM server handles error logging meant that the message we used to count re-prompts was not always pertinent to LLM hallucinations. Specifically, the same message was printed in the event of a timeout while awaiting the LLM response, making the two events indistinguishable. We advise future researchers to refine UTGen's error logging by creating a message that specifically addresses hallucinations detected after the CodeBLEU evaluation or the failure of the generated test case to compile.

8 Conclusions and Future Work

UTGen aims to enhance the understandability of automatically generated test suites by utilizing LLMs. The primary obstacle to achieving this effectively and efficiently is LLM hallucinations, where the LLM may produce code that is not compilable or significantly diverges from the initial test case UTGen intended to improve. To address this, we propose two approaches to prompt engineering that incorporate information retrieval, where the additional information can better guide the LLM towards a valid response. These approaches involve including the source code under test or the compilation errors generated by the latest LLM-produced test case in the prompt.

We evaluated the effectiveness of our approaches by comparing the hallucination performance to the base version of UTGen while attempting to enhance the understandability of the same EvoSuite test suite. The comparison was performed for three Java classes that have caused high hallucinations in the past. Including relevant source code in the prompt shows an improvement in the reverted test rate for classes with high LOC and WMC, while base UTGen performs better for ones with high *LCOM. On the other hand, adjusting the prompt by incorporating error codes minimizes the number of re-prompts and the reverted test rate for classes with high *LCOM. Still, classes with high LOC and WMC hallucinate less with base UTGen.

Further research shall explore how our approaches may be adapted to minimize the need for re-prompting more effectively. We recommend research in prompt engineering with a focus on making the prompts of either approach more concise and concentrated on the goal to be achieved. Additionally, we suggest the use of an LLM trained on more parameters for more accurate responses.

References

[1] Shalini Joshi and Indra Kumari. Analyses of Software Testing Approaches. In *2022 International Interdisciplinary Humanitarian Conference for Sustainability (IIHC)*, pages 1276–1281, November 2022.

[2] Kent Beck. *Test-driven Development: By Example*. Addison-Wesley Professional, 2003. Google-Books-ID: CUlsAQAAQBAJ.

[3] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation. *IEEE Transactions on Software Engineering*, 36(6):742–762, November 2010.

[4] Andrea Arcuri. An Experience Report On Applying Software Testing Academic Results In Industry: We Need Usable Automated Test Generation. *Empirical Software Engineering*, 23(4):1959–1981, August 2018. arXiv:1901.03865 [cs].

[5] 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.

[6] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Bang, Delong Chen, Ho Shu Chan, Wenliang Dai, Andrea Madotto, and Pascale Fung. Survey of Hallucination in Natural Language Generation. *ACM Computing Surveys*, 55(12):1–38, December 2023. arXiv:2202.03629 [cs].

[7] Harsha Nori, Yin Tat Lee, Sheng Zhang, Dean Carignan, Richard Edgar, Nicolo Fusi, Nicholas King, Jonathan Larson, Yuanzhi Li, Weishung Liu, Renqian Luo, Scott Mayer McKinney, Robert Osazuwa Ness, Hoifung Poon, Tao Qin, Naoto Usuyama, Chris White, and Eric Horvitz. Can Generalist Foundation Models Outcompete Special-Purpose Tuning? Case Study in Medicine, November 2023. arXiv:2311.16452 [cs].

[8] Angelika Mentzelopoulou. BSc-Thesis-LLM-Hallucination, June 2024. Available: https://doi.org/10.5281/zenodo.12511024.

[9] Jan Dirk Blom. *A Dictionary of Hallucinations*. Springer International Publishing, Cham, 2023.

[10] Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Bing Yin, and Xia Hu. Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond, April 2023. arXiv:2304.13712 [cs].

[11] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions, November 2023. arXiv:2311.05232 [cs].

[12] Vipula Rawte, Prachi Priya, S. M. Towhidul Islam Tonmoy, S. M. Mehedi Zaman, Amit Sheth, and Amitava Das. Exploring the Relationship between LLM Hallucinations and Prompt Linguistic Nuances: Readability, Formality, and Concreteness, September 2023. arXiv:2309.11064 [cs].

[13] Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, and Vincent J. Hellendoorn. CAT-LM: Training Language Models on Aligned Code And Tests, October 2023. arXiv:2310.01602 [cs].

[14] Aryaz Eghbali and Michael Pradel. De-Hallucinator: Iterative Grounding for LLM-Based Code Completion, January 2024. arXiv:2401.01701 [cs].

[15] Noor Nashid, Mifta Sintaha, and Ali Mesbah. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2450–2462, May 2023. ISSN: 1558-1225.

[16] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis, September 2020. arXiv:2009.10297 [cs].

[17] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing, July 2021. arXiv:2107.13586 [cs].

[18] Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Retrieval Augmented Code Generation and Summarization, September 2021. arXiv:2108.11601 [cs].

[19] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. SPOON: A library for implementing analyses and transformations of Java source code. *Software: Practice and Experience*, 46(9):1155–1179, 2016. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2346.

[20] 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. arXiv:2308.12950 [cs].

[21] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, February 2018.

[22] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. Large Language Models for Software Engineering: Survey and Open Problems, November 2023. arXiv:2310.03533 [cs].

[23] Talha Burak Alakus, Resul Das, and Ibrahim Turkoglu. An Overview of Quality Metrics Used in Estimating Software Faults. In *2019 International Artificial Intelligence and Data Processing Symposium (IDAP)*, pages 1–6, September 2019.

[24] Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E. Bourne, Jildau Bouwman, Anthony J. Brookes, Tim Clark, Mercè Crosas, Ingrid Dillo, Olivier Dumon, Scott Edmunds, Chris T. Evelo, Richard Finkers, Alejandra Gonzalez-Beltran, Alasdair J. G. Gray, Paul Groth, Carole Goble, Jeffrey S. Grethe, Jaap Heringa, Peter A. C. 't Hoen, Rob Hooft, Tobias Kuhn, Ruben Kok, Joost Kok, Scott J. Lusher, Maryann E. Martone, Albert Mons, Abel L. Packer, Bengt Persson, Philippe Rocca-Serra, Marco Roos, Rene van Schaik, Susanna-Assunta Sansone, Erik Schultes, Thierry Sengstag, Ted Slater, George Strawn, Morris A. Swertz, Mark Thompson, Johan van der Lei, Erik van Mulligen, Jan Velterop, Andra Waagmeester, Peter Wittenburg, Katherine Wolstencroft, Jun Zhao, and Barend Mons. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data*, 3(1):160018, March 2016. Publisher: Nature Publishing Group.

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.