Leveraging Large Language Models for Enhancing the Understandability of Generated Unit Tests

Amirhossein Deljouyi, Roham Koohestani, Maliheh Izadi, Andy Zaidman
Delft University of Technology
Delft, The Netherlands
a.deljouyi@tudelft.nl, r.koohestani@student.tudelft.nl, {m.izadi, a.e.zaidman}@tudelft.nl

***Abstract*—Automated unit test generators, particularly search-based software testing tools like EvoSuite, are capable of generating tests with high coverage. Although these generators alleviate the burden of writing unit tests, they often pose challenges for software engineers in terms of understanding the generated tests. To address this, we introduce *UTGen*, which combines search-based software testing and large language models to enhance the understandability of automatically generated test cases. We achieve this enhancement through contextualizing test data, improving identifier naming, and adding descriptive comments. Through a controlled experiment with 32 participants from both academia and industry, we investigate how the understandability of unit tests affects a software engineer's ability to perform bug-fixing tasks. We selected bug-fixing to simulate a real-world scenario that emphasizes the importance of understandable test cases. We observe that participants working on assignments with UTGen test cases fix up to 33% more bugs and use up to 20% less time when compared to baseline test cases. From the post-test questionnaire, we gathered that participants found that enhanced test names, test data, and variable names improved their bug-fixing process.**

***Index Terms*—Automated Test Generation, Large Language Models, Unit Testing, Readability, Understandability**

I. Introduction

In today's software-dominated world, software reliability and correctness are very important [1]. Consequently, automated testing in the form of unit tests has become a crucial element for software engineers in ensuring high-quality software [2]–[4]. Despite the widely acknowledged importance of testing, writing tests is tedious and time-consuming [5]–[8]. To alleviate this burden on developers and testers, the research community has devoted considerable effort on investigating automatic test generation approaches [9]–[14]. Among the notable test generators are Randoop [15] and EvoSuite [11]. EvoSuite, for example, is a search-based test generator that employs genetic algorithms to construct a test suite [16] and has demonstrated good results in terms of coverage [17], [18].

However, based on insights obtained through industrial case studies, there are limitations in terms of the quality of the generated test cases [19]–[25]. One critical limitation revolves around the understandability of generated test cases, which involves various aspects such as meaningful test data, proper assertions, well-defined mock objects, descriptive identifiers and test names, as well as informative comments. Additionally, the difficulty in following the scenario depicted in the test case and the ambiguity surrounding test data significantly hamper clarity [25], [26].

Listing 1 provides an example of an EvoSuite-generated test case. This test case checks the `equals` method with two objects of `weaponGameData` with different minimum damage values. Here, we see several comprehension challenges: 1) the purpose and functionality of a test method named with five arguments and "callsEquals3" is obscure, 2) the rationale behind the chosen test data remains unclear, 3) the identifiers are not providing any additional information, and 4) the absence of comments leaves the test case without essential explanatory context.

To address these issues, we aim to enhance automatically generated test cases by focusing on contextual test data, clear test method and identifier names, and adding descriptive comments. In this study, we investigate the synergy of Search-Based Software Testing (SBST) and Large Language Models (LLMs). While Natural Language Processing (NLP) techniques have shown promise in text generation and optimization [27], [28], and LLMs have advanced text-based capabilities [29]–[33], their impact in generating high-coverage test cases for complex systems remains limited [34], [35]. Conversely, SBST, while effective in coverage, often falls short in test case understandability.

Our approach, *UTGen*, integrates an LLM into the SBST test generation process. We hypothesize that this combined approach can leverage the strengths of both techniques to generate effective and understandable test cases. Our study is steered by three Research Questions (RQs) that consider the effectiveness of the UTGen approach, and the understandability of the generated test cases.

RQ$_1$ *Does UTGen have the capability to generate effective unit tests by utilizing a combination of LLMs and SBST?*

The investigation into the effectiveness of the approach seeks to establish whether the non-determinism of both the SBST and LLM components impact the ability to generate compilable and high-coverage unit tests.

RQ$_2$ *What is the impact of LLM-improved unit tests' understandability on the efficiency of bug fixing by developers?*

When it comes to the understandability of generated test cases, we intend to measure understandability through the ease by which software engineers can perform bug-fixing tasks involving failing test cases, a setup previously used by Panichella et al. [36].

```
@Test
public void
↪   testCreatesWeaponGameDataTaking6ArgumentsAndCallsEquals3() {
  WeaponGameData jSWeaponData_WeaponGameData0 = new
    WeaponGameData(35, 17, 35, "N&zMn$@6gffi<");
  WeaponGameData jSWeaponData_WeaponGameData1 = new
    WeaponGameData(35, 35, 35, "N&zMn$@6gffi<");
  boolean boolean0 = jSWeaponData_WeaponGameData0.
    equals(jSWeaponData_WeaponGameData1);
  assertFalse(boolean0);
}
```

Listing 1: Motivating Example

RQ$_3$ *Which elements of UTGen affect the understandability of the generated unit tests?*

We frame RQ$_3$ to obtain a deeper understanding about which elements of the UTGen approach determine the understandability of the generated test cases.

The key contributions of our paper are outlined as follows:

- UTGen, our novel approach that integrates an LLM into the SBST process to enhance the understandability of generated unit tests.
- The application of UTGen on 346 classes to examine the effectiveness of the generated unit tests.
- A controlled experiment and a post-test questionnaire with 32 participants from industry and academia were meant to evaluate the impact of LLM-improved test cases in terms of understandability in a bug-fixing scenario.
- We release a replication package that is publicly available with our implementation, as well as detailed data and results from our evaluation [37].

II. Background

A. Search-Based Software Testing

Automated test generation approaches have been developed in order to reduce testing effort. Tools like EvoSuite [11] and Randoop [15] generate a test suite starting from Java source code using a search-based or random approach [17], [18]. Several studies have uncovered challenges involving automatically generated tests [20]–[25], an important one being that generated tests are typically less readable than their human-written counterparts [38]. In this context, Almasi et al. [25] have observed that developers 1) find the test case scenario difficult to follow, 2) find the test data unclear, and 3) have difficulties with the meaningfulness of generated assertions.

B. Large Language Models

Large Language Models are a subset of AI systems predominantly based upon the transformer architecture [39]. These LLMs are trained on vast amounts of data, through which, they learn the underlying patterns inherent in texts, code, dialogue, etc., and are therefore capable of generating a (somewhat) relevant response given a prompt by the user [40]. LLMs operate based on predicting the subsequent tokens in a sequence and reusing the extended sequence by running it through the model once again to predict the tokens to follow (referred to as autoregression). This process is continued up until a point in which either the maximum amount of required tokens is reached or a termination character is generated.

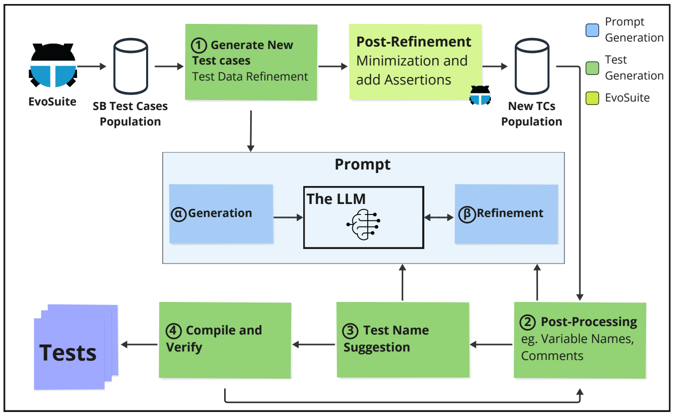


Fig. 1. Overview of the UTGen approach

Since their emergence, software engineers have utilized LLMs to enrich and simplify the development process [41]–[43]. In alignment with this, various open-source models, e.g., Code Llama [44] and StarCoder [45], and closed-source models, e.g., Codex [46], and GPT4 [40], have been trained and fine-tuned for this very purpose.

The research community has recently investigated incorporating LLMs into the test-generation process. In particular, attempts have been made to evaluate the efficacy of utilizing existing LLMs for unit test generation [29], [35], and training specialized LLMs for test generation [47]. However, the understandability and usability of these (hybrid) tests remain unclear. Additionally, methods such as CodaMosa [48] and TestPilot [29], respectively, propose the addition of LLMs to combat stalls in the search-based process and the full automation of the test generation process, which have proven to be useful. With all these additions, however, problems arise regarding the reliability, correctness, and complexity of incorporation given the non-deterministic nature of the results [35], [49].

Recent studies have pointed out that engineering good prompts is crucial for obtaining high-quality results [49]. For instance, the usage of the Chain of Thought reasoning (CoT) method has been shown to provide major improvements in the zero-shot performance of models [50], [51]. Furthermore, recent guidelines have been proposed that point toward the fact that including further information about the goal, context, and even the persona of the model can impact the quality of the results obtained [52]. While these guidelines provide a good start to the process of constructing a quality prompt, in most cases the task remains an empirical process at heart [53].

III. The UTGen Approach

Figure 1 provides an overview of our approach, named UTGen. The core of our framework is a search-based approach in which we integrated an LLM in various stages of the test generation process (highlighted in green). We use EvoSuite [11] as the search-based test generation framework of choice, and we have developed additional functionalities that facilitate the integration of EvoSuite with LLMs (highlighted in blue).

The aim of our approach, UTGen, is to enhance the understandability of test cases by improving four key elements of generated tests: 1) providing context-rich test data, 2) incorporating informative comments, 3) using descriptive variable names, and 4) picking meaningful test names. These goals define the stages in our approach.

As a first step, after the genetic algorithm has ended and the test cases mature in the search-based process, our approach focuses on refining test data (① in Figure 1). UTGen uses an LLM to generate contextually relevant test data, unlike traditional search-based methods that often rely on random values. Following this refinement, the search process ends, and we transition to post-processing tasks. Here, EvoSuite minimizes the number of test cases in the test suite, shortens the length of individual tests, and adds assertions.

Once the test cases are fully formed, at stage ②, UTGen leverages an LLM to add descriptive comments and enhance variable names. In stage ③, UTGen uses an LLM to suggest suitable names for the tests, reflecting the assertions and logic within. Finally, to ensure that test cases are compilable and stable after these enhancements, UTGen compiles them (stage ④), and in case of compilation issues, the process iteratively revisits stage ② for adjustments.

We first explain the prompt engineering component and then describe our test generation process per stage.

A. Prompt Generation

The prompt component of UTGen uses the `code-llama:7b-instruct` model from Meta [44] as provided by Ollama[1]. We have designed UTGen in such a way, that the Code-llama can easily be exchanged for another LLM. There are three stages within the UTGen approach uses the prompt component: 1) the refinement of test data, 2) the post-processing of tests, and 3) the naming of tests. The general prompt component contains two distinct parts, namely (α) which is responsible for generating the prompts provided to the LLM, and (β) which manages the request and ensures the correctness of the returned response.

For each stage, we devised specialized prompts following guidelines from recent prompt engineering research [50]–[52]. As shown in Listing 2, these guidelines emphasize the following: writing clear instructions with action words (as in ②), adopting a persona for the model (as in ①), allowing sufficient processing time through techniques like Chain of Thought (CoT) (as in ③), standardizing input and output formats, and framing requests in a positive manner (as in ④). The starting point for each prompt resembles the one presented in Listing 2. As each model has its complexities, pitfalls, and preferred input format, no one-size-fits-all solution exists to prompt engineering, however, the guidelines set out above have guided us. We have followed an iterative prompt engineering process in which each adjustment of the prompt was deliberated upon, before being accepted or rejected by the authors based on potential improvements in the results. An emerging pattern that we initially observed is that LLMs are incapable of always adhering to the output format described for them. Therefore, we put guidelines in place to deal with such mismatches; as an example, we had to deal with cases where plain text was placed inside the code blocks, or when the intended delineation was not used by the LLM. Our replication package contains the final versions of the prompts that we engineered, in addition to other measures that were taken [37].

```
   <<SYS>>
①{   You are a (ADJECTIVE) developer focusing on (TASK AT HAND)
   <</SYS>>
   [INST]
②  Your task is to (TASK)
   To achieve this, you should follow these structured steps:
     ** Detailed steps for analysis or improvement, emphasizing **
     1. Careful reading and understanding of the provided code.
     2. Identification of key functionalities or aspects requiring
     ↪  attention.
③    3. Formulation or modification of specific elements (e.g.,
     ↪  test names, data, comments) to enhance clarity,
     ↪  descriptiveness, or functionality.
     4. Adherence to coding standards and best practices, such as
     naming conventions or comment clarity.
   The code section requiring your attention is delineated by
   ↪  [CODE] and [/CODE] tags.
④  Your response, (whether it be a name, modified code, or
   comments), should be placed between the (OPENING TAG) and
   ↪  (CLOSING TAG).
   [CODE]
     The code segment to be analyzed or improved, dynamically
     ↪  inserted during execution
   [/CODE]
```

Listing 2: Prompt template for UTGen

B. Stage 1: Test Data Refinement

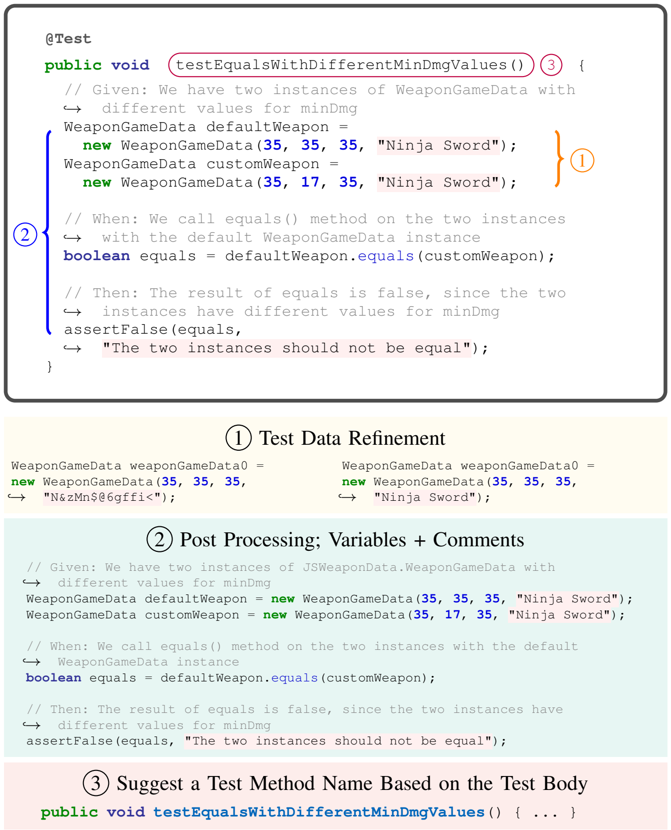
In this stage, we focus on requesting contextualized test data from the LLM to increase the domain relevance of test data for a test scenario. We designed a parser that converts the LLM's responses into the structured format required by EvoSuite.

The test data refinement stage should be considered as another iteration in the search process in which both new and original test cases coexist in the test population. The refined test cases are capable of changing the logic of the original test, and they cover different parts of the method under test.

An example of the refinement stage can be seen in stage ① of Figure 2, with the original and enhanced test data shown side by side. Based on the context, the LLM changes the fourth argument of the `WeaponGameData` constructor call from `"N&zMn$@6gffi<"` into `"Ninja Sword"`, which is more meaningful in the context of `WeaponGameData`.

However, it is important to acknowledge certain limitations in the LLM's responses. Occasionally, the LLM may hallucinate [54], e.g., generate lines that deviate from the original test case, or alter the number of parameters in method invocations. To mitigate these inconsistencies, we designed our parser to substitute the erroneous line with the corresponding line from the original test case if a corresponding line exists for it in the original test case. In the absence of a corresponding line in the original test case, the parser will skip parsing these erroneous

[1] Ollama: https://ollama.com/



```
@Test
public void testEqualsWithDifferentMinDmgValues() {
  // Given: We have two instances of WeaponGameData with
  ↪  different values for minDmg
  WeaponGameData defaultWeapon =
    new WeaponGameData(35, 35, 35, "Ninja Sword");
  WeaponGameData customWeapon =
    new WeaponGameData(35, 17, 35, "Ninja Sword");

  // When: We call equals() method on the two instances
  ↪  with the default WeaponGameData instance
  boolean equals = defaultWeapon.equals(customWeapon);

  // Then: The result of equals is false, since the two
  ↪  instances have different values for minDmg
  assertFalse(equals,
  ↪  "The two instances should not be equal");
}
```

① Test Data Refinement

```
WeaponGameData weaponGameData0 =          WeaponGameData weaponGameData0 =
new WeaponGameData(35, 35, 35,            new WeaponGameData(35, 35, 35,
↪  "N&zMn$@6gffi<");                      ↪  "Ninja Sword");
```

② Post Processing; Variables + Comments

```
// Given: We have two instances of JSWeaponData.WeaponGameData with
↪  different values for minDmg
WeaponGameData defaultWeapon = new WeaponGameData(35, 35, 35, "Ninja Sword");
WeaponGameData customWeapon = new WeaponGameData(35, 17, 35, "Ninja Sword");

// When: We call equals() method on the two instances with the default
↪  WeaponGameData instance
boolean equals = defaultWeapon.equals(customWeapon);

// Then: The result of equals is false, since the two instances have
↪  different values for minDmg
assertFalse(equals, "The two instances should not be equal");
```

③ Suggest a Test Method Name Based on the Test Body

```
public void testEqualsWithDifferentMinDmgValues() { ... }
```

Fig. 2. A simplified example of a test case enhanced by UTGen per step

lines and continue parsing the remaining portions of the LLM-generated test cases. This increases the chance that even test cases with omissions are valid for compilation. For instance, if the LLM's response adds a non-existent statement like `weaponGameData0.increaseDmg(10)`, the parser skips this line and continues processing. Similarly, if the LLM alters a method's parameter count, like changing `weaponGameData0.getDmgBonus()` to `weaponGameData0.getDmgBonus(10)`, the parser uses the original method call with zero parameters. These strategies ensure the parser extracts the maximum number of statements from the LLM responses, minimizing the need for re-prompting.

In post-refinement, EvoSuite optimizes the test case population and adds assertions to them. The optimization includes shortening test cases, and eliminating duplicated test cases from the population. The selection of which duplicate test case to keep and which to eliminate is directed by a secondary objective, which prioritizes selecting the test case that minimizes the total length of all test cases within the set of duplicates.

C. Stage 2: Post-Processing

In this stage, we make the final chosen test as understandable as possible by making various aspects of the code more understandable. UTGen achieves this by adding descriptive comments, and making variable names more clear.

After the post-refinement has finished, assertions are added to the test cases, and the test cases have reached maturity in terms of coverage, they are given to the LLM for improvement. The LLM is instructed to add comments (using the Given, When, Then convention — seen as more understandable [55]) and to exclusively change the naming of the variables but to let the data and logic untouched given that this could impact the intended behavior of a certain test.

To ensure maximal logical similarity between original and enhanced test cases, we use the CodeBLEU metric which effectively assesses syntactic and semantic similarities between two sequences [56]. We choose to control for similarity to increase the cohesion between generated and improved test cases as well as minimize the impact of LLM hallucinations. A CodeBLEU score below 0.5 triggers a re-prompting process.

We cap re-prompting at three iterations, as our findings suggest that this limit preserves logical coherence and still facilitates the improvement of tests. If the LLM does not meet the threshold after three attempts, the prompt simplifies, removing comment structure constraints, and thus allowing deviation from the initial format. Should the LLM's response still not reach a satisfactory level after a total of six attempts, the original test case is retained. The value of three attempts per prompting strategy is also chosen to balance effectiveness and execution cost.

Additionally, as previous literature has pointed out, results from LLMs can be non-deterministic given a non-modified temperature of the model being used, this can in turn lead to results diverging from the original tests, or tests that do not have correct syntax. To ensure consistency and reliability in the returned results, we employ a set of heuristic safeguards to facilitate the process of controlling for such anomalies. With each response from the LLM, we 1) try to identify and remove common mistakes made by the LLM in the code, e.g., comments placed inside the code as plain text and not as comments, 2) attempt to correct any missing closing brackets in a piece of code, 3) validate code using CodeBLEU as previously described, and 4) check the syntactic correctness of the returned results with the parser generator tool ANTLR[2].

Furthermore, we re-prompt the LLM in the case when any of the previously described safeguards fail to improve the response, fail to achieve syntactic correctness, or have lower-than-threshold values for CodeBLEU. We limit the amount of recursive calls that are made to not have a single improvement request stall the entire process. All the processes explained above relate to the component marked with β in Figure 1.

An example of this step is shown in ② in Figure 2: the comments in Given-when-then format are added, and the variable names are changed from `weaponGameData0` and `weaponGameData1` to the `defaultWeapon` and `customWeapon`, matching the logic of the test case. Also, the assertion message is added from the LLM response.

D. Stage 3: Test Method Name Suggestion

In this stage, UTGen gives the LLM the completed method body of the test, and it is asked to deduce a descriptive name.

[2] ANTLR: https://www.antlr.org/

TABLE I
DEMOGRAPHICS OF PARTICIPANTS

Attendance	Academia	Industry	Σ
In Person	17	4	21
Remote	3	8	11
Σ	20	12	32

Experience	Academia		Industry	
	In Java	In Testing	In Java	In Testing
0-2 years	5	9	0	2
3-6 years	11	9	8	5
7-10 years	2	1	3	2
$\geq$ 10 years	2	1	1	3

Affiliation	Academia		Industry	
	Role	Number	Role	Number
	PhD Student	9 (45%)	Developer	6 (50%)
	MSc Student	8 (40%)	Senior Researcher	3 (25%)
	BSc Student	1 (5%)	Scientific Dev.	1 (8%)
	Post Doc	1 (5%)	Team Lead	2 (17%)
	Scientific Dev.	1 (5%)		

We chose to put this stage after the post-processing of the test method body because then the test case includes comments that increase the context for the LLM to generate a descriptive test method name. If another test case already has a similar test name, we re-prompt until it has a unique name.

For instance, in ③ in Figure 2, the LLM suggests `testEqualsWithDifferentMinDmgValues()`. This name reflects the test's functionality of examining the `equals` method across varying minimum damage values. In comparison, EvoSuite named this test `testCreatesWeaponGameDataTaking6ArgumentsAnd-CallsEquals3`.

E. Stage 4: Compile and Verify

After successfully navigating through the safeguards, it is still possible for a test case to fail to compile. Therefore, we compile all test cases, and a non-compiling test case undergoes a repeated cycle of post-processing and test method name suggestion, with a default post-processing budget of 2 iterations.

Compiling test cases are then assessed for their stability. A test case is considered unstable if it fails due to an exception unrelated to a JUnit assertion. All test cases that are both compilable and stable are saved.

IV. Experiment Setup

In this section, we describe the methodology of evaluation of our approach. We investigate the following RQs:

RQ$_1$ *Does UTGen have the capability to generate effective unit tests by utilizing a combination of LLMs and SBST?*

RQ$_2$ *What is the impact of LLM-improved unit tests' understandability on the efficiency of bug fixing by developers?*

RQ$_3$ *Which elements of UTGen affect the understandability of the generated unit tests?*

We now discuss the evaluation strategies for RQ1 to RQ3.

A. Effectiveness Evaluation Setup (RQ1)

We explore the effectiveness of UTGen on two axes: the compilability rate of LLM-improved test cases, and a comparison in coverage of baseline and UTGen test cases.

TABLE II
JAVA CLASSES USED FOR THE CONTROLLED EXPERIMENT

Project	Class	LOC	Methods	Branches
caloriecount	Budget	152	21	16
twfbplayer	JSWeaponData	177	19	44

1) Dataset: We utilize the DynaMOSA dataset composed of 346 non-trivial Java classes from 117 open-source projects for RQ1 [18]. The classes are selected from four different benchmarks, with the primary source being the 204 non-trivial classes of SF110 [57].

2) Evaluation: We evaluated UTGen using the EvoSuite framework as a baseline. We applied UTGen on a dataset and generated two types of test cases: original EvoSuite test cases and LLM-improved test cases. We then compare these two types of test cases by measuring 1) the number of LLM-improved test cases that compiled successfully, 2) branch and instruction test coverage, and 3) pass/fail rates.

3) Parameter Configuration: We decided to use the default configuration parameters for EvoSuite, which have been empirically shown to provide good results [58]. We did increase the test budget (max_time) from 60 to 200 seconds, to ensure that the search algorithm has enough time to generate a test population that achieves reasonable coverage levels.

B. Controlled Experiment (RQ2)

We conducted a controlled experiment to assess the understandability of test cases in a real-world scenario, namely bug fixing [59]. This extends the work of Panichella et al., who investigated the impact of generating documentation for automatically generated tests in the context of bug fixing [36].

The experiment involved 32 participants. The experimental group worked with UTGen test cases, while the control group was given EvoSuite test cases. We configured EvoSuite with *coverage-based test naming*, which generates more readable test names than the default setting [60].

We examined two dependent variables in the experiment: 1) the number of fixed bugs, and 2) time efficiency, measured as the time taken to fix the bugs.

1) Participants: We recruited participants with academic and industrial backgrounds. Table I presents their demographics. To engage academic participants, the experiment was advertised via the university's communication channels. Additionally, developers from an industrial partner were enlisted. Furthermore, all authors reached out to their professional networks of software engineers. We made sure to extend the invitation to individuals with experience in Java and testing.

2) Objects: To design the bug-fixing assignments and compare experimental and control groups, it was essential to choose two projects that would offer a solid foundation for understanding the context of bug fixing. To do so, we analyzed all classes within the SF110 dataset, gathered insights into the distribution of Lines of Code (LOC), which serves as an indicator of complexity [61]. Using this data, we calculated the mean (μ) and standard deviation (σ) for each distribution. We then identified all classes falling within the range of $\mu \pm 0.1\sigma$

across all specified metrics. This process yielded a total of 15 classes. Upon manual inspection of these classes, we selected two for consideration: 1) `Budget`, which includes methods for calculating calories over intervals, and 2) `JSWeaponData`, featuring methods related to Weapon Objects in a Java game. Table II provides details on the two classes. We inject four faults in each class, with each fault located in a different method under test. The injected bugs included replacement of arithmetic operations (2 bugs), statement deletion (1 bug), boolean relation replacement (2 bugs), and variable replacement (3 bugs). While the types of faults were similar across both classes, fixing the faults in the `Budget` class can be more challenging due to its detailed time calculations.

3) Experimental Design: Our experiment utilized a 2 × 2 factorial crossover design; it featured two periods and included a two-level blocking variable based on the object. In each period, subjects applied a different technique (treatment) to a different object (assignment). We preferred the crossover design over a between-subject design due to the latter requiring a larger number of participants to achieve sufficient statistical power. The design of the experiment is detailed in Table III, which outlines the four sequences used. We followed the experimental design guidelines provided by Vegas et al. [62].

To minimize learning effects, participants were given tasks involving different objects in each period. Additionally, to avoid any potential bias from optimal sequencing, we balanced the participants over the sequences in terms of the number of participants, and academic versus industry background. For participants from academia, each sequence was executed 5 times, while for industrial participants it was executed 3 times.

4) Experimental Procedure: The participants were able to execute the controlled experiment either in-person or remotely through videoconferencing. Before the actual experiment, we asked participants to fill in a pre-test questionnaire to gauge their experience. One day before the experiment session we sent them 1) a statement of consent, 2) instructions and materials for performing the experiment, including the two assignments, 3) a number indicating the sequence (see Table III), and 4) a link to the online survey platform. This advance preparation was necessary, because during the pilot evaluation we observed that receiving the projects just before the experiment led to additional preparation time, increasing the threat of tiredness. It could also lead to stress among participants if they encountered difficulties.

During the experiment, an examiner was continuously present to explain expectations and control any external factors that could affect the experiment, e.g., ensuring that participants did not use external sources to fix bugs.

In the experiment, we asked the participants to carry out two tasks; each task consisted of fixing four bugs in 30 minutes. We assume extending the time or having an unlimited window box could intensify the learning effect and introduce threats of tiredness/boredom. If the participant indicated to have fixed all 4 bugs within the 30-minute time frame, the examiner double-checked this, and the participant could proceed to the next step. Each participant received two tasks: 1) a task consisting of one Java class with a corresponding test class generated with UTGen, and 2) a Java class with a corresponding test class generated by the baseline approach, i.e., EvoSuite.

TABLE III
EXPERIMENTAL DESIGN

#Seq	Order	Period 1		Period 2	
		Object	Technique	Object	Technique
I	U-E	Budget	UTGen	JSWeaponData	EvoSuite
II	E-U	Budget	Evosuite	JSWeaponData	UTGen
III	U-E	JSWeaponData	UTGen	Budget	Evosuite
IV	E-U	JSWeaponData	Evosuite	Budget	UTGen

5) Pilot: We engaged 4 participants (not part of the 32 participants) to pilot our experiment. After the pilot run, we changed the tasks from fixing 5 bugs in 20 minutes, to fixing 4 bugs in 30 minutes, and clarified the expected behaviours through Javadoc documentation. We also narrowed the scope of the code, segregating it into *definitely good* and *possibly faulty* code sections. Thus ensuring that the assignments were feasible within the 30-minute time frame. Finally, we improved the task descriptions, sending detailed instructions and an overview of the experiment to participants beforehand.

6) Analysis Method: We conducted statistical tests to determine whether there was a significant difference between the number of bugs found and the time taken to fix bugs in LLM-improved test cases compared to baseline test cases. Due to our crossover design, we accounted for potential carryover effects, which required treating the data as dependent. Therefore, nonparametric hypothesis tests for independent samples like the Wilcoxon Rank Sum test were not suitable [62].

Instead, we employed mixed models for our analysis. Specifically, for each of our dependent variables:

1) *the number of fixed bugs*: this variable is discrete and bounded between 0–4, and we treated it as an ordinal variable. Consequently, we used Cumulative Link Mixed Models [63], which are appropriate for this type of data.
2) *time efficiency*: this variable represents the time taken to fix bugs, and we used Generalized Linear Mixed Models with a Gamma distribution, which is suitable for time-related data [64].

We considered Technique, Object, Technique:Object, Order (confounded with carryover), and Period as fixed effects, and participants (#id) as a random effect. The sequence effect is embedded within the variables Order and Technique:Object. We set the significance level at 0.05 for both models.

Additionally, we examined whether factors such as participants' background, programming experience in Java and Testing, as well as whether the sessions were attended in-person or remotely, interacted with the technique on the number of fixed bugs. In these cases, we extended the mixed model by adding these factors to assess their interaction with the technique.

We also used Cohen's d to measure the effect size ranging from very small ($d < 0.2$) to small ($0.2 \leq d < 0.49$), medium ($0.5 \leq d < 0.79$), and large ($d \geq 0.8$) [65].

TABLE IV
QUESTIONNAIRE OVERVIEW

#	Title	Type of Question	Aspect
Q1	In your opinion, what factors make finding bugs easier for you?	Open	1
Q2	Do you think the understandability of the test cases affects your bug fixing?	Likert	1
Q3	Prioritize the elements in helping understandability	Ranking	2
Q4	How important are the following elements in the understandability of the test case	Likert	2
Q5	How do you judge the understandability of the provided test case (Task 1 and 2)	Likert, Open	3
Q6	Evaluate how good you think the first task is in each item	Matrix Table, Open	3
Q7	Evaluate how good you think the second task is in each item	Matrix Table, Open	3

C. Post-Test Questionnaire (RQ3)

We used the post-test questionnaire to obtain feedback from the participants of the controlled experiment on which aspects of UTGen affect the understandability of test cases (see Table IV). We focused on gauging three aspects: 1) participants' views on how the understandability of test cases impacts their bug-fixing effectiveness, 2) their opinion on what factors in test code contribute to the understandability of generated test cases, and 3) their ratings of the quality of these factors in test cases with and without the LLM-improved enhancements.

1) Questionnaire: In Q1, we ask participants to identify factors they believe to affect bug fixing effectiveness. Importantly, at this stage, the participants are unaware that the experiment focuses on the understandability of generated test cases, ensuring that their responses genuinely reflect their initial thoughts on bug fixing. In Q2, we query whether the participants think the clarity of generated test cases influences bug fixing. Q3 and Q4 gauge which factors impact understandability most. In Q5, we ask the participants to rate the understandability of the two tasks, using a Likert scale along with open-ended feedback. Finally, in Q6 and Q7, we ask participants to rate specific elements such as comments, test data, test names, and variable naming in the test cases of both tasks in terms of *completeness*, *conciseness*, *clarity*, and *naturalness*, thus aiming for a detailed evaluation of different aspects of test case quality [28], [36], [66].

2) Analysis Method: For the open-ended questions, we sorted the data into categories using a card-sorting method and calculated the frequency of each category. Two authors independently reviewed the card-sorting process and achieved an 84% agreement. For the Likert-scale questions, we determined the mean value and percentage of each answer. For Q6 and Q7, we used the Wilcoxon Rank Sum test with an α of 0.05 because it did not follow a normal distribution (as determined by the Shapiro-Wilk test with a p-value $<< 0.01$). We used Cohen's d effect size to determine the extent of the difference.

V. RESULTS

In the following we discuss the results per research question.

TABLE V
EFFICACY RESULTS OF UTGEN -GENERATED TESTS

#	1. Pass/Failed	Test Count	Pass Rate
1.	EvoSuite	8315	79.01%
2.	UTGen	8430	73.27%
#	2. Improved Tests	Test Count	Percentage
1.	Improved Tests	6110	72.48%
2.	Reverted Tests	992	11.77%
3.	Enhancment Stagnation	1328	15.75%
#	3. Coverage	Instruction Coverage	Branch Coverage
1.	EvoSuite	25.03%	18.68%
2.	UTGen	24.43%	17.87%

A. RQ1: Effectiveness of Integrating LLMs and Search-Based Methods for Generating Unit Tests

We define effectiveness as the capability of UTGen to generate unit tests that are compilable and execute reliably, along with their ability to cover the classes under test. The success rate, defined as the proportion of generated tests that pass upon execution, reflects functional correctness. It is important to note that while all generated tests compile, the success rate pertains solely to their execution outcome.

UTGen successfully generates a total of 8430 tests, with a pass rate of 73.27%, while EvoSuite produces 8315 tests at a slightly higher pass rate of 79.01%. The heuristic safeguard described in Section III-C ensures the syntactic correctness and compilability of test cases, but also leads to 27.52% of the tests were categorized as "*enhancement stagnation*", i.e., the LLM could not improve the test case, or "*reverted*", i.e., we went back to the EvoSuite base test case, as the test case failed to compile. As such, these 27.52% of test cases compile, but are not meaningfully affected by UTGen.

The origin of certain test cases not being meaningfully affected by UTGen lies in the non-deterministic nature of LLMs. As we have no guarantee that tests given to the LLM will compile upon improvement due to the possible hallucinations by the LLM, we employ several safeguards. While the safeguards explained in Section III-C do manage to catch a great portion of the tests that would not compile, some do fall through. Therefore, we perform a compilation check (④ in Figure 1). If any (improved) test fails to compile, we revert back to an EvoSuite-generated test case.

Out of the total 8430 tests generated by UTGen, 11.77% are non-compiling and are thus reverted to the initial test case generated by EvoSuite. The remaining 15.75% of tests are due to the stagnation of the enhancement process and the inability of the LLM to make a significant contribution.

Finally, from Table V we observe that EvoSuite reaches slightly higher coverage compared to UTGen: instruction coverage is 25.03% compared to 24.43%, while branch coverage is 18.68% compared to 17.87%. In a further investigation into the reason for this delta in coverage, we find that small changes in the post-processing step, e.g., changes in values of parameters, affect the overall coverage achieved.

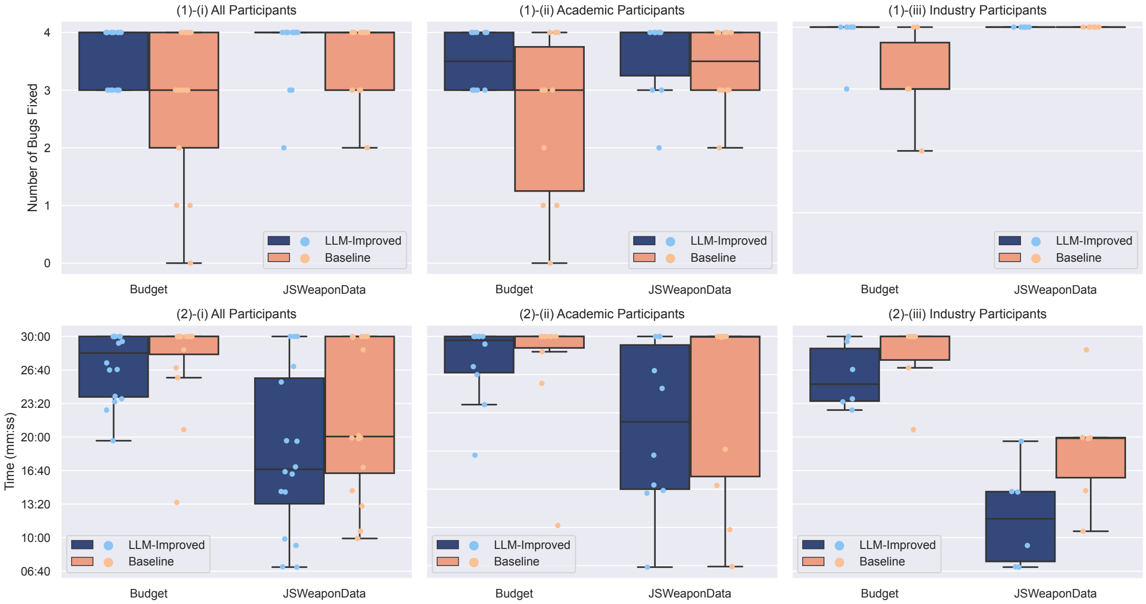


Fig. 3. (1) Number of bugs and (2) Time taken for each different group: (i) All Participants (ii) Academic Participants (iii) Industry Participants

> **RQ1** *A total of 8430 tests are generated by UTGen, of which 72.48% are improved, and 27.52% are not due to reversion or stagnation. The coverage results are marginally comparable to the baseline.*

B. RQ2: The Impact on Bug Fixing

Figure 3 presents the results of the controlled experiment in terms of two dependent variables: 1) the number of bugs fixed, and 2) time efficiency, measured by the duration required to complete the tasks. The results are reported for respectively the entire population, the academic participants, and the industry participants.

For both objects, participants fixed more bugs in the task with the LLM-improved test cases compared to the baseline test cases. For the `Budget` class the difference is more pronounced as the participants fixed a median of 4 bugs with LLM-improved test cases, compared to a median of 3 bugs fixed for baseline test cases. For the `JSWeaponData` class, the difference is marginal, as participants fixed a median of 4 bugs with either of the test cases. According to the tests of fixed effects presented in Table VI, we observe in the fixed bugs column that both technique ($p = 0.024$) and object ($p = 0.025$) significantly influence the number of fixed bugs. This implies that using the LLM-Improved test cases significantly increases the likelihood of fixing more bugs. Similarly, when the object is JSWeaponData, the probability of fixing more bugs is also significantly higher. The result of Cohen's d effect size for the treatment is medium at 0.59.

Regarding time efficiency, participants using LLM-improved test cases generally took less time to fix all bugs for both classes.

However, the differences in timing are not statistically significant for the technique ($p = 0.063$), with significance observed only for the object ($p = 0.031$). The difference is more apparent in the `JSWeaponData` class, where the average time to fix all bugs was 18:22 for LLM-improved versus 22:06 for baseline test cases (20% less time). For the `Budget` class, the averages are closer: 27:06 for LLM-improved and 27:51 for baseline test cases. This is mainly due to a 30-minute cutoff, which limited the observable difference.

Additionally, a post hoc analysis of Estimated Marginal Means involving a pairwise comparison of different technique levels for each specific object level indicates that in the `Budget` class, the treatment (LLM-Improved test cases) is significant ($p = 0.024$), whereas it is not significant in the `JSWeaponData` class ($p = 0.319$). The Cohen's d effect size is large of 0.92 for the treatment in the `Budget` class. We hypothesize that the statistically significant improvement in the number of bugs fixed in the `Budget` assignment, compared to `JSWeaponData`, is due to the greater complexity of scenarios and bugs in the `Budget` class. This complexity likely increases the demand for clearer and more understandable test cases.

Furthermore, neither the period ($p = 0.176$ and $p = 0.068$) nor the order ($p = 0.138$ and $p = 0.517$) significantly impact the number of fixed bugs and time efficiency. This indicates that there is no carryover effect between treatments. The interaction between technique and object is not significant, suggesting that the effect of the technique on the number of bugs fixed and time efficiency does not depend on the object. Additionally, our analysis found no significant interaction between the technique and co-factors such as participants' backgrounds, experience in Java and testing, or whether they attended sessions remotely or in person ($p >> 0.05$).

Finally, in terms of the influence of the background of our participants, we observe that both population groups show better performance when using LLM-improved test cases compared to baseline test cases in terms of both number of bugs fixed and time taken to fix bugs. We observe that academic

TABLE VI
Tests of Fixed Effects

Source	Fixed Bugs		Time Efficiency	
	Estimate	$Pr(>\|z\|)$	Estimate	$Pr(>\|z\|)$
Technique	2.997	0.024	-0.116	0.063
Object	2.903	0.025	0.133	0.031
Technique: Object	0.951	0.401	0.088	0.408
Order	1.588	0.138	-0.069	0.517
Period	0.951	0.176	-0.114	0.068

TABLE VII
Participants' responses to the questionnaire

Q2. The effect of understandability on bug fixing	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
	6.2%	9.3%	6.2%	34.4%	**43.7%**
Q3. Prioritize the elements in helping understandability		Rank 4	Rank 3	Rank 2	Rank 1
1. Comment		9.3%	25%	31.2%	**34.3%**
2. Test Name		34.3%	9.3%	15.6%	**40.6%**
3. Variable Naming		21%	**34.3%**	**34.3%**	9.3%
4. Test Data		**34.3%**	31.2%	18.7%	15.6%
Q4. How important are the elements in the understandability	Not important	Slightly important	Moderately important	Very important	Extremely important
1. Comment	6.2%	15.6%	21.8%	**34.3%**	21.8%
2. Test Name	6.2%	21.8%	21.8%	12.5%	**34.3%**
3. Variable Naming	3.1%	12.5%	**34.3%**	31.2%	18.7%
4. Test Data	0%	12.5%	28.1%	**43.7%**	15.6%
Q5. The quality of test cases	Very low	Low	Moderate	High	Very high
LLM-improved	3.13%	6.25%	25.0%	50.0%	15.63%
Baseline	6.25%	18.75%	25.0%	43.75%	6.25%

participants seem to benefit more from the LLM-improved test cases in aiding bug fixing. For industrial participants on the other hand, the time-saving gain is more pronounced. Figure 3 provides a more detailed overview.

> **RQ2** *In our experiment, using LLM-Improved tests significantly increases the likelihood of fixing more bugs.*

C. RQ3: The effects of different elements of UTGen on understandability

The results of the post-test questionnaire show three aspects: 1) the participants' views on how the understandability of test cases impacts their bug-fixing effectiveness, 2) their opinion on what factors in test code contribute to the understandability and 3) their ratings of the quality of elements in test cases with and without the LLM-improved enhancements.

Aspect 1: How understandability of test cases impacts bug-fixing: We answer the first aspect through the responses to Questions 1 and 2 in the survey. We have observed that participants find a well-written test suite important for bug fixing: they frequently highlighted (14 mentions) the importance of descriptive and clear test names, appropriate use of assertions, and well-chosen test data in test suites. This aspect was prioritized over other factors like high-quality production code (10 mentions). We also take note of the overall (strong) agreement that test case understandability is important in in the context of bug fixing, as indicated by a median score of 4 out 5 (Q2 in Table VII).

Aspect 2: What factors in test code contribute to understandability: We have analyzed the participants' responses to Questions 3 and 4, where they ranked and scored the importance of elements. From Table VII we observe that participants give more importance to comments and test names, than to variable naming and test data. Specifically, 34.3% of the participants ranked comments as most important, while 40.6% gave priority to test names in Question 3.

Aspect 3: The quality of factors in test cases with and without LLM enhancements: In Q5 of Table VII, we see that participants rate the understandability of LLM-improved tests somewhat better when compared to the baseline test cases.

We asked participants in Q6 and Q7 to evaluate an LLM-improved and baseline test case of the assignments on different criteria and specifically per test element. These criteria comprised *completeness*, *conciseness*, *clarity*, and *naturalness*.

Figure 4 shows the results of Q6 and Q7. The results indicate that LLM-improved test cases are consistently rated higher compared to baseline test cases for each of the criteria (first row). The Wilcoxon test confirms this statistically significant difference (p-value < 0.05) for all criteria. The effect size for conciseness was small, while it was large for completeness, naturalness, and clarity. Notably, in the open-ended responses, some participants mentioned that some comments in LLM-improved test cases were too general and added little value. The respondents did appreciate the Given-When-Then-structured comments.

When we zoom into the test elements, we see improvements in all areas for LLM-improved test cases: comments, test data, test name, and variable naming. The Wilcoxon test for all of these elements is statistically significant with a p-value $<< 0.05$. The effect size for comments, test data, and test names is medium, while very large for variable naming ($d > 1.2$).

Through the analysis of the open-ended responses to Questions 5–7, we found that the complexity of a test case has an impact on the necessity of comments. For simpler test cases, using a Given-When-Then (Arrange/Act/Assert) structure is often sufficient. However, for more complex cases, more detailed comments are needed to ensure optimal comprehension. Overall, participants mentioned this point 18 times, with one participant stating "The test code lines are straightforward, so comments are unnecessary." Similarly, for simpler test cases, the quality of variable naming was less of a concern: participants mentioned this factor only 4 times when rating a short baseline test case.

> **RQ3** *Comments, test names, variable names, and test data are improved compared to the baseline. Specifically, participants highlighted improved conciseness, clarity, and naturalness in these test elements.*

VI. Discussion

In this section, we discuss our results, their implications, and threats to the validity of our study.

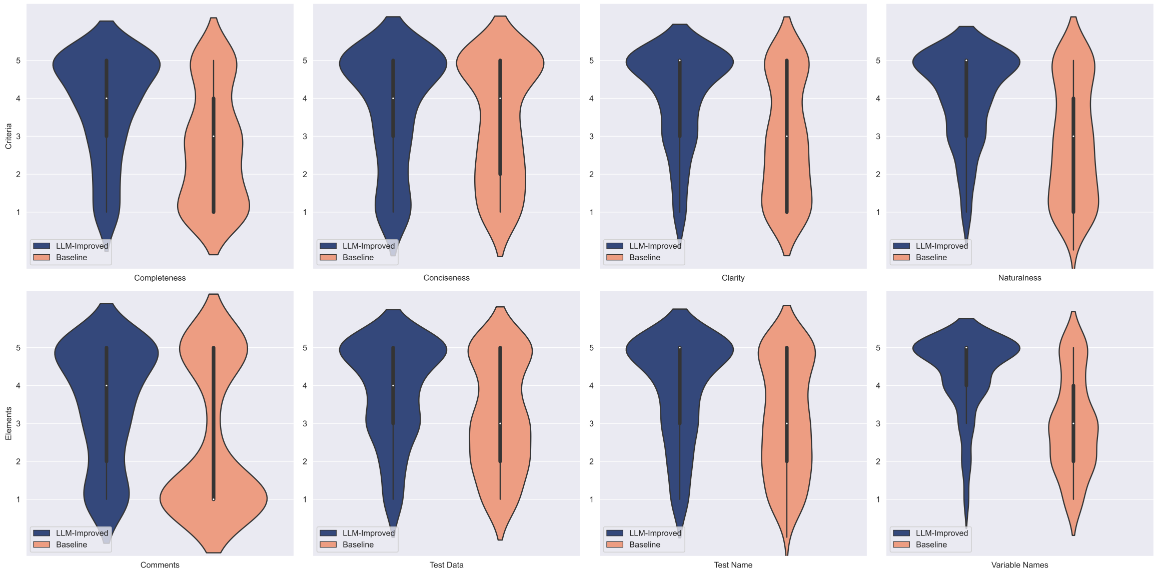


Fig. 4. The results of Q6 and Q7 which test cases were rated in terms of criteria (first row) and test elements (second row)

A. *Revisiting the Research Questions*

RQ1: *Does UTGen have the capability to generate effective unit tests by utilizing a combination of LLMs and SBST?* When we compare the effectiveness of our LLM-inspired UTGen approach and EvoSuite, we observe that UTGen generates test cases that have relatively similar structural coverage. However, we also noticed a phenomenon that we term *enhancement stagnation*, which occurs when the LLM is not able to improve the test case, even when re-prompting multiple times. We analyzed this situation and found indications that this stagnation is correlated with high complexity. In this context, we define complexity at the level of the class under test to be: 1) methods having a high number of parameters, and 2) methods being tightly coupled, i.e., many method calls between objects or within an object. While generally adding more relevant context can help an LLM, highly complex projects can overwhelm LLMs due to lengthy input codes and insufficient contextual information, thus hindering the enhancement process during post-processing. To overcome this, we propose to incorporate Retrieval Augmented Generation (RAG) techniques. We hypothesize that these enhancements can reduce occurrences of Enhancement Stagnation as it has resolved similar stagnation issues in other domains [67], [68]. RAG involves enhancing LLMs by dynamically integrating knowledge from databases, knowledge graphs, or the internet in real time into the generation process to provide contextually richer and more accurate responses.

RQ2: *What is the impact of LLM-improved unit tests' understandability on the efficiency of bug fixing by developers?* From the results of the controlled experiment, we see indications that the LLM-based enhancements brought to the generated unit tests improve their understandability in the bug-fixing scenario. Specifically, the experimental group outperformed the control group by fixing up to 33% more bugs and completing tasks up to 20% faster. Our experiment consisted of two assignments involving respectively the `Budget` and `JSWeaponData` classes. While we observed statistically significant improvements for the `Budget` assignment, the other assignment did not reach statistical significance. Since the `Budget` class is comprised of more complex scenarios and bugs, we hypothesize that the complexity of a test scenario increases the need for understandable test cases. This hypothesis was anecdotally confirmed by participants in the post-test questionnaire.

RQ3: *Which elements of UTGen affect the understandability of the generated unit tests?* Through the post-test questionnaire, we captured that participants think that LLM-improved test cases are showing improvements in terms of comments, test names, test data, and variable names when compared to baseline test cases. At a higher level, participants also rated completeness, conciseness, clarity, and naturalness as better. However, feedback from open-ended questions highlights that comments should be more precise and informative. Similarly, some participants also highlighted that simple test methods might not require (extensive) comments. Upon reflecting on this feedback, we hypothesize that generically trained LLMs, while generally robust, might lack task-specific data to effectively assist in creating comments.

B. *Implications*

Our study's results have an important implication for researchers and tool builders. In particular, our study indicates that a generally trained LLM can already instigate a considerable improvement in the understandability of search-based generated test cases. However, our results also show that test case comments should be more detailed in some cases, while seeming superfluous in other situations. Therefore, we see potential in creating specifically-trained LLMs for particular software engineering tasks, but equally in customizing LLM responses to individual software engineers.

C. *Threats to Validity*

Construct Validity. Threats to construct validity relate to the setup of our study. We conducted the study either in person or remotely, with an examiner present. To control for factors other than the codebase, we ensured a consistent setup for all participants and limited the choice of IDE to IntelliJ, providing uniform capabilities. However, this approach may disadvantage participants having experience with other IDEs, potentially affecting their performance.

Internal Validity. To mitigate threats to internal validity, we did not reveal the tool names in our experiment and questionnaire. To prevent bias in selecting classes for the assignments, we followed a systematic selection process to strengthen the methodological integrity. To compensate for a learning effect, we created four different sequences of the experimental design. Using mixed models, we found that period and carryover effects were not statistically significant, indicating they do not pose major threats to the study's validity.

External Validity. The classes that we use to determine the efficacy of test generation in RQ1 are a potential threat to the generalization of our results. To address this, we used a dataset of 346 classes from 117 open-source Java projects that form a representative sample and were previously used in software testing studies [18], [69]. We limited the controlled experiment in RQ2 to two Java classes. To ensure their representativeness, we carefully selected them from the SF110 dataset containing real-world classes, and taking the average LOC of that entire dataset into consideration to select "average classes". Future work will explore more complex classes. In order to mitigate potential imbalance between the experimental and control groups, we carefully balanced participants over both groups in terms of experience and background.

VII. Related Work

A. *Improving the Understandability of Test Cases*

Panichella et al. [36] introduced TestDescriber, which generates test case summaries that describe the intent of a generated unit test; they established that these summaries enable software engineers to resolve bugs more quickly. Similarly, Roy et al. [28] developed DeepTC-Enhancer, leveraging deep learning to produce method-level summaries for test cases. Both efforts highlight the value of summarizing test cases. In contrast, UTGen generates detailed comments within the test cases themselves and provides a narrative of the test scenario.

Zhang et al. [27] introduced an NLP technique for automatically generating descriptive unit test names. Daka et al. [60] applied coverage criteria for naming automatically generated unit tests, while Roy et al. [28] created DeepTC-Enhancer by employing deep learning to rename identifiers in test cases to improve readability. Unlike these methods that rely on traditional NLP techniques, UTGen utilizes LLMs to suggest identifiers that fit the test scenario's context.

Afshan et al. [70] enhanced the readability of inputs by combining natural language models with search-based test generation. Deljouyi et al. [66] proposed an approach that generates understandable test cases with meaningful data through end-to-end test scenario carving. Baudry et al. [71] developed a test data generator using LLMs to produce realistic, domain-specific constraints. Our method is similar to Baudry et al.'s, but we focus on search-based unit test generation.

B. *Generating Test Cases by LLM*

Despite the progress in LLM-based test generation, to the best of our knowledge, no study has focused on enhancing unit test case understandability through the integration of search-based methods and LLMs. Research in this field shows considerable variability in methods and outcomes. Siddiq et al. generated tests using LLMs and reported 2% coverage on the SF110 dataset [35]. In contrast, Schäfer et al.'s [29] TestPilot for JavaScript achieved 70% statement-level coverage on relatively small systems. Alshahwan et al. aimed to improve human-written tests by LLMs and submit them for human review [72]. Meanwhile, Lemieux et al. explored overcoming coverage stalls in SBST with LLMs [48], and Moradi et al. investigated mutation testing with LLMs [73]. Steenhoek et al. improved test generation by minimizing test smells through reinforcement learning [74]. Unlike the aforementioned studies, UTGen focuses on enhancing understandability through integrating LLMs in the SBST process. Notably, UTGen achieved 17.87% branch coverage, surpassing the pure LLM approach by Siddiq et al. [35].

VIII. Conclusion

Recent research has suggested that the understandability of test cases is a key factor to optimize in the context of automated test generation [25]. Therefore, in this paper, we introduce the UTGen approach that incorporates a Large Language Model (LLM) into the Search-Based Software Testing (SBST) process. In doing so, UTGen aims to improve the understandability by providing context-rich test data, informative comments, descriptive variables, and meaningful test names.

We first evaluated UTGen's test generation effectiveness on 346 non-trivial Java classes, observing that UTGen successfully enhanced 72.48% of the test cases, and slightly decreased coverage compared to EvoSuite-generated tests (RQ1). We then performed a controlled experiment with 32 participants from industry and academia; we observed that test cases generated by UTGen facilitated easier bug-fixing with participants fixing up to 33% more bugs and doing so up to 20% faster (RQ2). Feedback from participants in the post-test questionnaire indicated a significant improvement in test case completeness, conciseness, clarity, and naturalness (RQ3).

In future work, we aim to explore optimization strategies, such as Retrieval Augmented Generation (RAG), to enhance prompt efficiency and minimize the need for re-prompting. Furthermore, we plan to refine our approach by creating customized fine-tuned LLMs specifically for test generation. These customized LLMs would replace the publicly-available pre-trained LLM that we currently use.

Acknowledgement

This research was partially funded by by the Dutch science foundation NWO through the Vici "TestShift" grant (No. VI.C.182.032).

References

[1] A. J. Ko, B. Dosono, and N. Duriseti, "Thirty years of software problems in the news," in *Proc. Int'l Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. ACM, 2014, pp. 32–39.

[2] K. L. Beck, *Test-Driven Development - By Example*, ser. The Addison-Wesley signature series. Addison-Wesley, 2003.

[3] A. Khatami and A. Zaidman, "State-of-the-practice in quality assurance in Java-based open source software development," *Software: Practice and Experience*, vol. 54, no. 8, pp. 1408–1446, 2024.

[4] ——, "Quality assurance awareness in open source software projects on github," in *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2023, pp. 174–185.

[5] M. Beller, G. Gousios, A. Panichella, S. Proksch *et al.*, "Developer testing in the IDE: patterns, beliefs, and behavior," *IEEE Trans. Software Eng.*, vol. 45, no. 3, pp. 261–284, 2019.

[6] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their IDEs," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 179–190.

[7] M. Beller, G. Gousios, and A. Zaidman, "How (much) do developers test?" in *37th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2015, pp. 559–562.

[8] M. F. Aniche, C. Treude, and A. Zaidman, "How developers engineer test cases: An observational study," *IEEE Trans. Software Eng.*, vol. 48, no. 12, pp. 4925–4946, 2022.

[9] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *IEEE Trans. Software Eng.*, vol. 36, no. 6, pp. 742–762, 2010.

[10] L. Baresi and M. Miraz, "Testful: automatic unit-test generation for java classes," in *32nd IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 2010, pp. 281–284.

[11] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *Proc. Joint Meeting Symp. Foundations of Software Engineering and the European Softw. Eng. Conf. (ESEC/FSE)*. ACM, 2011, pp. 416–419.

[12] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated unit test generation really help software testers? A controlled empirical study," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 4, pp. 23:1–23:49, 2015.

[13] P. Derakhshanfar, X. Devroey, A. Panichella, A. Zaidman, and A. van Deursen, "Generating class-level integration tests using call site information," *IEEE Trans. Software Eng.*, vol. 49, no. 4, pp. 2069–2087, 2023.

[14] C. E. Brandt, A. Khatami, M. Wessel, and A. Zaidman, "Shaken, not stirred: How developers like their amplified tests," *IEEE Trans. Software Eng.*, vol. 50, no. 5, pp. 1264–1280, 2024.

[15] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Conf. on Object-Oriented Programming Systems and Applications (OOPSLA-Companion)*. ACM, 2007, pp. 815–816.

[16] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *Empirical Software Engineering*, vol. 20, no. 3, pp. 783–812, 2015.

[17] ——, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[18] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Trans. Software Eng.*, vol. 44, pp. 122–158, 2018.

[19] A. Arcuri, "An experience report on applying software testing academic results in industry: we need usable automated test generation," *Empirical Software Engineering*, vol. 23, no. 4, pp. 1959–1981, 2018.

[20] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "Automatic test case generation: What if test code quality matters?" in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2016, pp. 130–141.

[21] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia, "On the diffusion of test smells in automatically generated test code: An empirical study," in *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*, 2016, pp. 5–14.

[22] G. Grano, F. Palomba, D. Di Nucci, A. De Lucia, and H. C. Gall, "Scented since the beginning: On the diffuseness of test smells in automatically generated test code," *Journal of Systems and Software*, vol. 156, pp. 312–327, 2019.

[23] G. Fraser and A. Arcuri, "EvoSuite: On the challenges of test case generation in the real world," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2013, pp. 362–369.

[24] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser *et al.*, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges," in *International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 201—-211.

[25] 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 *Proc. Int'l Conf. on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017.

[26] C. E. Brandt and A. Zaidman, "Developer-centric test amplification," *Empir. Softw. Eng.*, vol. 27, no. 4, p. 96, 2022.

[27] B. Zhang, E. Hill, and J. Clause, "Towards automatically generating descriptive names for unit tests," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 2016, pp. 625–636.

[28] D. Roy, Z. Zhang, M. Ma, V. Arnaoudova *et al.*, "Deeptc-enhancer: Improving the readability of automatically generated tests," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 287–298.

[29] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 85–105, 2024.

[30] S. Yu, C. Fang, Y. Ling, C. Wu, and Z. Chen, "LLM for test script generation and migration: Challenges, capabilities, and opportunities," *arXiv*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2309.13574

[31] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio *et al.*, "Studying the usage of text-to-text transfer transformer to support code-related tasks," *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021.

[32] V. Liventsev, A. Grishina, A. Härmä, and L. Moonen, "Fully autonomous programming with large language models," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 2023, pp. 1146–1155.

[33] J. Wang, Y. Huang, C. Chen, Z. Liu *et al.*, "Software testing with large language model: Survey, landscape, and vision," *arXiv*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2307.07221

[34] K. El Haji, C. Brandt, and A. Zaidman, "Using github copilot for test generation in python: An empirical study," in *Proceedings of the International Conference on Automation of Software Test (AST)*. ACM, 2024.

[35] M. L. Siddiq, J. C. S. Santos, R. H. Tanvir, N. Ulfat *et al.*, "Using large language models to generate junit tests: An empirical study," in *International Conference on Evaluation and Assessment in Software Engineering (EASE)*. ACM, 2024.

[36] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, "The impact of test case summaries on bug fixing performance: An empirical investigation," in *Proc. Int'l Conference on Software Engineering (ICSE)*, 2016, pp. 547–558.

[37] "Replication package of UTGen," 2024. [Online]. Available: https://doi.org/10.5281/zenodo.13329464

[38] G. Grano, S. Scalabrino, H. C. Gall, and R. Oliveto, "An empirical investigation on the readability of manual and generated test cases," in *International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 348–351.

[39] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit *et al.*, "Attention is all you need," *arXiv*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.1706.03762

[40] OpenAI:, J. Achiam, S. Adler, S. Agarwal *et al.*, "Gpt-4 technical report," *arXiv*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.08774

[41] M. Izadi, R. Gismondi, and G. Gousios, "Codefill: Multi-token code completion by jointly learning from structure and naming sequences," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 401–412.

[42] M. Izadi, J. Katzy, T. Van Dam, M. Otten *et al.*, "Language models for code completion: A practical evaluation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[43] A. Al-Kaswan, T. Ahmed, M. Izadi, A. A. Sawant *et al.*, "Extending source code pre-trained language models to summarise decompiled

binaries," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 260–271.

[44] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla *et al.*, "Code llama: Open foundation models for code," *arXiv*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.12950

[45] R. Li, L. B. Allal, Y. Zi, N. Muennighoff *et al.*, "Starcoder: may the source be with you!" *arXiv*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.06161

[46] M. Chen, J. Tworek, H. Jun, Q. Yuan *et al.*, "Evaluating large language models trained on code," *Arxiv*, 2021. [Online]. Available: https://doi.org/10.48550/arXiv.2107.03374

[47] N. Rao, K. Jain, U. Alon, C. Le Goues, and V. J. Hellendoorn, "Cat-lm training language models on aligned code and tests," in *38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 409–420.

[48] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 919–931.

[49] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "LLM is like a box of chocolates: the non-determinism of ChatGPT in code generation," *arXiv*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.02828

[50] J. Wei, X. Wang, D. Schuurmans, M. Bosma *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *arXiv*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2201.11903

[51] J. Li, G. Li, Y. Li, and Z. Jin, "Structured chain-of-thought prompting for code generation," *arXiv*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.06599

[52] G. Marvin, N. Hellen, D. Jjingo, and J. Nakatumba-Nabende, "Prompt engineering in large language models," in *International Conference on Data Intelligence and Cognitive Informatics*. Springer, 2023, pp. 387–402.

[53] J. Zamfirescu-Pereira, R. Y. Wong, B. Hartmann, and Q. Yang, "Why Johnny can't prompt: how non-AI experts try (and fail) to design LLM prompts," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–21.

[54] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy *et al.*, "Large language models for software engineering: Survey and open problems," *arXiv*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2310.03533

[55] V. Khorikov, *Unit Testing Principles, Practices, and Patterns*. Manning, 2019.

[56] S. Ren, D. Guo, S. Lu, L. Zhou *et al.*, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv*, 2020. [Online]. Available: https://doi.org/10.48550/arXiv.2009.10297

[57] G. Fraser and A. Arcuri, "A large scale evaluation of automated unit test generation using evosuite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, p. 8, 2014.

[58] A. Arcuri and G. Fraser, "Parameter tuning or default values? an empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, pp. 594–623, 2013.

[59] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., 2005.

[60] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?" in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2017, pp. 57–67.

[61] J. Graylin, J. E. Hale, R. K. Smith, H. David *et al.*, "Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship," *Journal of Software Engineering and Applications*, vol. 2, no. 03, p. 137, 2009.

[62] S. Vegas, C. Apa, and N. Juristo, "Crossover designs in software engineering experiments: Benefits and perils," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 120–135, 2016.

[63] R. H. B. Christensen, "Cumulative link models for ordinal regression with the r package ordinal," *Submitted in J. Stat. Software*, vol. 35, 2018.

[64] S. Lo and S. Andrews, "To transform or not to transform: using generalized linear mixed models to analyse reaction time data," *Frontiers in Psychology*, vol. 6, 2015.

[65] G. M. Sullivan and R. Feinn, "Using effect size—or why the p value is not enough," *Journal of graduate medical education*, vol. 4, no. 3, pp. 279–282, 2012.

[66] A. Deljouyi and A. Zaidman, "Generating understandable unit tests through end-to-end test scenario carving," in *Proceedings of the 23rd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2023, pp. 107–118.

[67] M. R. Parvez, W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Retrieval augmented code generation and summarization," *arXiv*, 2021. [Online]. Available: https://doi.org/10.48550/arXiv.2108.11601

[68] S. Liu, Y. Chen, X. Xie, J. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid gnn," 2021.

[69] J. Campos, Y. Ge, N. Albunian, G. Fraser *et al.*, "An empirical evaluation of evolutionary algorithms for unit test suite generation," *Information and Software Technology*, vol. 104, pp. 207–235, 2018.

[70] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2013, pp. 352–361.

[71] B. Baudry, K. Etemadi, S. Fang, Y. Gamage *et al.*, "Generative ai to generate test data generators," *arXiv*, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2401.17626

[72] N. Alshahwan, J. Chheda, A. Finegenova, B. Gokkaya *et al.*, "Automated unit test improvement using large language models at Meta," *arXiv*, 2024. [Online]. Available: https://doi.org/10.48550/ArXiv.2402.09171

[73] 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," 2023.

[74] B. Steenhoek, M. Tufano, N. Sundaresan, and A. Svyatkovskiy, "Reinforcement learning from automatic feedback for high-quality unit test generation," *arXiv*, 2023. [Online]. Available: https://doi.org/10.48550/ArXiv.2310.02368