ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation

Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo

Abstract—Recent advancements in large language models (LLMs) have demonstrated exceptional success in a wide range of general domain tasks, such as question answering and following instructions. Moreover, LLMs have shown potential in various software engineering applications. In this study, we present a systematic comparison of test suites generated by the ChatGPT LLM and the state-of-the-art SBST tool EvoSuite. Our comparison is based on several critical factors, including correctness, readability, code coverage, and bug detection capability. By highlighting the strengths and weaknesses of LLMs (specifically ChatGPT) in generating unit test cases compared to EvoSuite, this work provides valuable insights into the performance of LLMs in solving software engineering problems. Overall, our findings underscore the potential of LLMs in software engineering and pave the way for further research in this area.

Index Terms—ChatGPT, Search-based Software Testing, Large Language Models

1 Introduction

Unit testing is a widely accepted approach to software testing that aims to validate the functionality of individual units within an application. By using unit tests, developers can detect bugs in the code during the early stages of the software development life cycle and prevent changes to the code from breaking existing functionalities, known as regression [1]. The primary objective of unit testing is to confirm that each unit of the software application performs as intended. This method of testing helps improve the quality and reliability of software by identifying and resolving issues early on.

SBST. The importance of unit testing in software development and the software development life cycle cannot be overstated. To generate unit test cases, search-based software testing (SBST) [2] techniques are widely employed. SBST is a technique that employs search algorithms such as genetic algorithms and simulated annealing to create test cases. The objective of SBST is to utilize these kinds of algorithms to optimize the test suites, resulting in a set of test cases that provide extensive code coverage and effective detection of program defects. Compared to other testing techniques, SBST exhibits promising results in reducing the number of test cases while maintaining the same level of defect detection capability [3], [4]. SBST has emerged as an effective approach to improving the quality and efficiency of software testing, providing a valuable tool for software developers to streamline the testing process.

Large Language Model and ChatGPT. Recently, Large language models (LLMs) have exhibited remarkable proficiency in processing and performing everyday tasks such as machine translation, question answering, summarization, and text generation with impressive accuracy [5], [6], [7]. These models possess a capacity nearly equivalent to that of humans for understanding and generating human-like text. One such example of a real-world LLM application is OpenAI's GPT-3.5 (Generative Pretrained Transformer 3.5), which has been trained on an extensive amount of text data from the internet. Its practical implementation, ChatGPT [1], is widely employed in various daily activities, including text generation, language translation, question answering, and automated customer support. ChatGPT has become an essential tool for many individuals, simplifying various tasks and improving overall efficiency.

Deep-learning based Test Case Generation. Besides accomplishing daily tasks, such as text generation, language translation, and question answering, large language models are also been adopted and used to cope with software engineering (SE) tasks, such as, code generation [8], [9], [10], code summarization [11], [12], [13], document and comments generation [14], [15], and more. These models can be employed to generate unit test cases for programs with the help of a large number of real-world test cases written by developers/testers. This allows for the validation of the intended functionality of individual units within the software application. The integration of LLMs in SE tasks has demonstrated their versatility and potential for improving software development processes.

Motivation. Despite SBST performing well in generating unit tests, there is still a learning cost for test personnel with limited experience. As a result, it can be a barrier to embracing SBST techniques, especially for fresh testers. However, the applications based on large language models can accomplish the same task (i.e., generating test suites) with nearly no learning costs. However, it is still unknown whether the unit tests generated by SBST can be compared with advanced artificial intelligence models and techniques. For example, a complete assessment of the readability,

Y. Tang is with University of Glasgow, United Kingdom
Z. Liu and Z. Zhou are with ShanghaiTech University, China
X. Luo is with the Department of Computing, Hong Kong Polytechnic University, Hong Kong SAR, China
Y. Tang (yutian.tang@glasgow.ac.uk) is the corresponding author.

1. CharGPT: The version used in this study is GPT-3.5 instead of GPT-4

understandability, reliability, and practical usability of the LLM-generated test cases has not yet been conducted. Here, in this paper, we are interested in understanding the strengths and weaknesses of test suites generated by LLM. Specifically, we leverage the state-of-the-art GPT-3.5 [16] model's product ChatGPT [17], [16] as a representative of LLM for comparison. More importantly, this paper intends to gain insights from two aspects: (1) we are keen on the knowledge we can learn from large language models to improve the state-of-the-art SBST techniques, and (2) we are also interested in uncovering the potential limitations of the existing large language models in generating test suites.

The rationale behind focusing our comparison on Search-Based Software Testing (SBST) tools rather than a broader range of unit test case generation tools. In our revised manuscript, we elucidated this decision with the following justifications:

- **Prevalence of SBST:** SBST tools, such as EvoSuite, represent a widely adopted and well-studied class of automated unit test generation tools in both academic research and industry practice. By comparing ChatGPT with a well-established benchmark in the field, we aim to provide a meaningful context for evaluating the performance and potential of LLMs like ChatGPT in unit test generation.
- **Clear Benchmarking:** The systematic and optimization-driven nature of EvoSuite provides clear metrics for comparison, such as code coverage. These metrics allow for a more objective evaluation of the test cases generated by ChatGPT in relation to a known standard.
- **Complementarity in Approach:** The contrast between the heuristic-based approach of EvoSuite and ChatGPT offers a rich ground for comparison. By focusing on SBST, we can more effectively highlight the unique contributions and limitations of ChatGPT in generating unit test suites.

Our Study. To cope with the aforementioned challenges and achieve the goals, in this paper, we intend to answer the following research questions (RQ):

- **RQ1 (Validity):** How Do the Validities of ChatGPT-Generated and Evosuite-Generated Unit Test Suites Compare?
- **RQ2 (Readability):** How Understandable is the Test Suite Provided by ChatGPT Compared to That of Evosuite?
- **RQ3 (Code Coverage):** How does ChatGPT perform with SBST in terms of code coverage?
- **RQ4 (Bug Detection):** How effective are ChatGPT and SBST at generating test suites that detect bugs?
- **RQ5 (Correctness of Assertions):** How do the assertions generated by ChatGPT in unit tests compare in correctness to those produced by SBST?
- **RQ6 (Non-determination of ChatGPT):** How does the non-deterministic output of ChatGPT affect the quality and effectiveness of generated test cases, as measured by code coverage, fault detection?

Contribution. In summary, we make the following contributions in this paper:

- In this paper, we conduct the *first* comparative assessment of LLMs and SBST in terms of generating unit test suites for programs in Java programming language;
- We systematically evaluate the test suites generated by ChatGPT from various aspects, including correctness, readability, code coverage, bug detection capability; and
- Our findings contribute to a better understanding of the potential for LLMs to improve software engineering practices, specifically in the domain of unit test generation.

2 Background

SBST and Evosuite. Search-based software testing (SBST) is a technique that formulates unit test generation as the optimization problem [18]. SBST regards code coverage as the test generation's target (e.g., branch coverage) and describes it as a fitness function to guide genetic algorithms [3], [19], [20]. The genetic algorithms evolve tests by iterating to (1) apply mutation and crossover operators to existing tests (i.e., the current generation) for new offspring tests and (2) form a new generation by selecting those with better fitness scores from the current generation and offspring. In our work, we choose the most mature SBST tool in Java, Evosuite [21].

LLM and ChatGPT. LLM is the type of biggest model in terms of parameter count, trained on enormous amounts of text data (e.g., human-like text, code, and so on) [22], [23], [24], [16], [25], [17]. It is designed to process and understand input natural language text and to generate text consistent with the input, and shows a strong ability in natural language processing (NLP) tasks, such as, machine translation, question answering, text generation, and so on. ChatGPT [17] is now the most widely LLM (i.e., adapt to human expression by using Instruct) [26], [25] implemented atop GPT-3.5. GPT-3.5 [16] is constructed on multi-layer Transformer decoders [22], [27], [28] using few-shot learning (i.e., multiple examples). It shows performance similar to that of state-of-the-art fine-tuned systems in many tasks. One example of using GPT-3.5 is shown in Fig. 1. GPT-3.5 takes in the input prompt and infers the answer based on the task description and examples in the input prompt. To make LLM further align with users (humans), InstructGPT [25] utilizes additional supervised learning and reinforcement learning from human feedback to fine-tune GPT-3.5. ChatGPT [17] uses the same methods as InstructGPT and has the ability to answer follow-up questions.

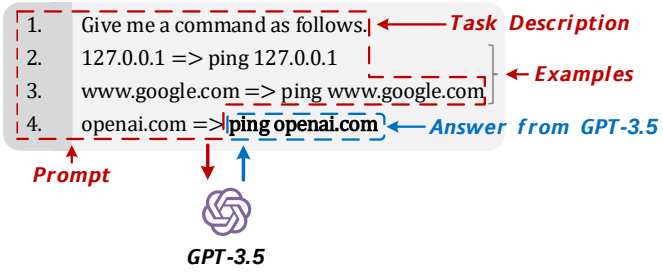

Fig. 1: A Sample Use of GPT-3.5

For generating unit test cases, one can utilize a large language model like GPT-3.5. To generate new test cases given code snippets as input, the model can be fine-tuned on a dataset of code snippets and their accompanying test cases. One can also take advantage of ChatGPT's answering follow-up questions to generate more diverse test suites for given code snippets.

Using of ChatGPT. ChatGPT [17], [16] can be used as follows. The software developer/tester (user) registers an

account for ChatGPT. Then, users send a prompt (a text or a question) to ChatGPT. Then, ChatGPT will respond based on the information it has learned from its training data. Also, ChatGPT can be used in most software-engineering related tasks, such as, generating code, generating comments, and generating test cases. For example, as shown in Fig. 2, ChatGPT offers a basic user interface like a Chatbot, in which a user can ask any question in a natural language. As shown in Fig. 2, we ask ChatGPT how to make an HTTP request in Python, and ChatGPT shows a sample code written in Python with corresponding explanations. If a user is not satisfied with the generated responses, (s)he can ask ChatGPT to regenerate a response by clicking the "Regenerate a response" button at the bottom of the page.

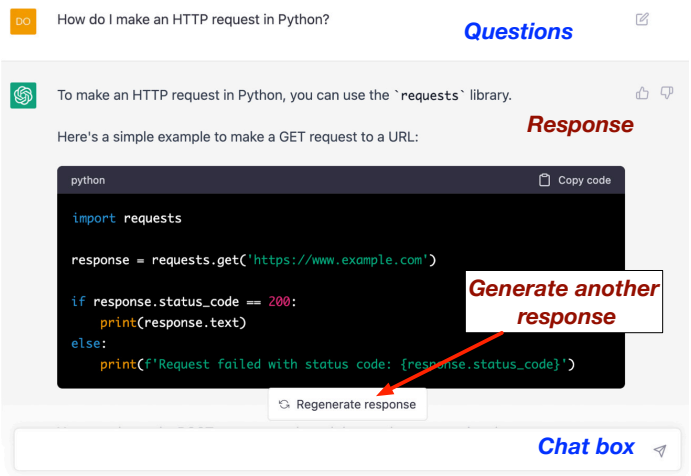

Fig. 2: A Sample Use of ChatGPT in a SE Task

3 Comparative Assessment Setup

3.1 Data Collection

As for **RQ1-3**, to reduce bias in selecting subject code for generating test cases, we reuse the existing benchmark used in the existing study to evaluate the performance of Evosuite. Here, we use the benchmark presented in DynaMOSA (a.k.a Dynamic Many-Objective Sorting Algorithm) [20]. The benchmark contains 346 Java classes from 117 projects. The detailed class information can be founded in [20] and our artifact repository (Sec.8). However, based on facts reported by other works [4], [29], some projects in the SF100 dataset can be obsolete and are no longer maintained. Some projects are not able to build and compile as some classes required in DynaMOSA dataset are missing or not publicly available. As a result, we remove 38 projects and retain 79 projects with 248 Java classes. As for **RQ4**, we use the state-of-the-art defect database for Java-related research, which is Defects4J [30]. It contains 835 bugs from 17 open-source projects.

Potential Data Leakage from ChatGPT (or LLMs) In addressing the concern regarding potential data leakage from using off-the-shelf ChatGPT models, it is important to note the nature of the datasets employed in our study. Defects4J and dataset used by DynaMOSA primarily provide a collection of reproducible bugs from real-world open-source projects, rather than serving as a benchmark for unit test suites. Given this context, the likelihood of ChatGPT having been trained on closely mapped code and unit test case pairs from these sources is exceedingly low. Furthermore, the capability of these models to generate unit test cases is distinct from their training on natural language data. Hence, we contend that the potential for data leakage from ChatGPT's training data impacting the results of our empirical study is considerably low.

3.2 Using ChatGPT to Generate Unit Test Cases

With the help of ChatGPT, we are able to automatically generate unit test cases for programs. Unfortunately, there is no standard or oracle on how to use ChatGPT to automatically generate unit test cases with ChatGPT. Therefore, we adopt the following step to learn a reasonable practice of using ChatGPT to generate unit test cases:

- **Step 1.** Collecting existing tools that leverage LLM (e.g., ChatGPT) to automatically generate unit test cases from various sources, including Google, Google Scholar, GitHub, and technical blogs [31], [32], [33], [34], [35], [36];
- **Step 2.** Analyzing the phrases and descriptions used in these tools to prompt LLM to generate test cases. This part involves analyzing source codes, reading blocks, and learning technical documents; and
- **Step 3.** Verifying the phrases and descriptions collected in Step 2 with ChatGPT to exclude invalid phrases and descriptions;

Through the **Step 1-3**, we obtain the following representative expressions that are able to generate unit test cases for a code segment:

- **Expression 1:** "Write a unit test for ${input}" with the code segment under test as the input;
- **Expression 2:** "Can you create unit tests using JUnit for ${input}?" with the code segment under test as the input;
- **Expression 3:** "Create a full test suite with test cases for the following Java code: ${input}?" with the code segment under test as the input;

Based on the above findings, we summarize our prompt as: "Write a JUnit test case to cover methods in the following code (one test case for each method): ${input}?" with the code segment under test as the input. Note that, to mimic the real-world practice, we do not intend to compare and evaluate the ChatGPT prompts to build a best-performed prompt. Instead, we only intend to **build a reasonable prompt** for ChatGPT to stimulate how developers use ChatGPT in a real-world environment.

In our study, we choose not to extensively compare and evaluate ChatGPT prompts for the following reasons: **(1) Focus on Model Capabilities**: Our primary objective is to assess the capabilities of ChatGPT as a unit test generator, rather than focusing on prompt engineering. We target on understanding its performance "out of the box" using prompts that are straightforward and easy to understand, as this reflects a common usage scenario where developers (users) may not have the expertise to craft complex prompts; and **(2) Avoid Overfitting**: Evaluating multiple prompts and selecting the best-performing one could lead to overfitting the model to the specific prompt and dataset. We intend to avoid this potential bias and assess ChatGPT's generalization abilities.

3.3 Other Setups for the Study

• **Setup for EvoSuite.** EvoSuite provides many parameters (e.g., crossover probability, population size [37]) to run the algorithms. In this paper, to evaluate and compare the performance between Evosuite and ChatGPT, we remain the default settings in Evosuite. As Evosuite leverages genetic algorithms in selecting and generating test cases, to reduce the bias introduced by randomness, we run 30 times for each class. We do not set any time limitation for ChatGPT. It is because the generative nature of ChatGPT means that imposing strict time limits may not be as suitable for its operation compared to tools designed for specific tasks.

• **Long Inputs for ChatGPT.** The maximum input length for ChatGPT is 2,048 tokens, which is roughly equivalent to 340-350 words. If the input submitted is too long, ChatGPT reports an error message and gives no response. In this case, we try to split the entire class by methods and ask ChatGPT to generate unit test cases for methods. However, splitting the entire class by methods to generate test cases cannot be a good practice as some information about the entire class cannot be perceived by ChatGPT. As a result, it hurts the quality of generated test cases. Here, we set the maximum length to be 4,096 tokens. That is, if the length of a class is larger than 4,096 tokens, we discard it. The decision to discard only classes with more than 4096 tokens is based on practical considerations and the limitations of ChatGPT's token limit. When a class exceeds 2,048 tokens, it is already challenging to generate unit tests for it within a single conversation due to the token limit. Therefore, we decided to set the threshold at 4,096 tokens to give ChatGPT a bit more room to work with and avoid discarding classes that might still be feasible to handle. Furthermore, based our experiments, we found that classes with more than 4,096 tokens often have relatively "large" functions, and it's usually not reasonable to split the bodies of these large functions.

• **Environment.** Experiments on EvoSuite are conducted on a machine with Intel(R) Core(TM) i9-10900 CPU @ 2.80GHz and 128 GB RAM.

4 Experiment and Evaluation

4.1 Validity

RQ1: How Do the Validities of ChatGPT-Generated and Evosuite-Generated Unit Test Suites Compare?

Motivation. The first and foremost thing we need to examine is whether ChatGPT can correctly return the test cases for testing the program/code segment given.

Methodology. To test whether the generated test cases are correct. We need to evaluate them from three aspects: (1) whether ChatGPT successfully returns the test case for each input under test; (2) whether these test cases can be compiled and executed; and (3) whether these test cases contain potential bugs. Specifically, for (2), it can be examined with the help of Java Virtual Machine (JVM). We compile and execute the test cases to see whether JVM reports errors. For (3), we rely on the state-of-the-art static analyzer, SpotBugs [38], [39], [40], to scan the test cases generated by ChatGPT to find out whether these test cases contain potential bugs or vulnerabilities. SpotBugs [38] is the successor of FindBugs [39], [40] (an abandoned project) and is an open-source static software analyzer, which can be used to capture bugs in a Java program. It supports more than 400 bug patterns and poor programming practices.

Results.

▷ **Validity of test cases generated by ChatGPT.**

According to the *Long-input setting* in Sec. 3.3, we remove 41 classes and remain 207 Java classes from 75 projects.

We find that ChatGPT can successfully generate unit test cases for all 207 Java classes without reporting any errors. Among these test cases, there are 144 (69.6%) test cases can be successfully compiled and executed without needing extra-human efforts. Next, we ask two undergraduate students who have basic knowledge of Java programming to attempt to repair errors with the help of IntelliJ IDE [41]. For the rest 64 test cases, there are 3 test cases that cannot be directly fixed without the background knowledge of the target program, and 60 test cases can be repaired with the help of IDE. Specifically, the errors in 3 test cases fall into 3 categories: a) fail to implement an interface; b) fail to initiate an abstract class instance; c) try to initiate an instance of an inner class.

TABLE 1: Error Types in 60 Test Cases

Type of Errors	Frequency
Access Private/Protected Field	31
Access Private/protected Methods	20
Invoke undefined methods	11
Fail to initiate an instance for an interface	10
Incorrect parameter type	2
Fail to initiate an instance	2
Access undefined field	1

The errors in other 60 test cases fall into 7 categories as shown in Table. 1. Here, *invoke undefined methods* represents invoking a method, which is not defined in the target class. Table. 2 shows some samples of invoking undefined method errors. The root cause for invoking undefined methods is that ChatGPT is only given the class under test instead of the entire project. As a result, ChatGPT has to predict the name of a callee when needed. This is especially the case when ChatGPT attempts to generate some *assertions*. However, the results in Table. 2 also surprise us that even if the ChatGPT fails to call the correct callees, its prediction also gives a strong clue to find the correct callee names. This is why we can fix these errors without the need of domain knowledge of these target projects. *Fail to initiate an instance for an interface* represents that ChatGPT creates an instance of an interface, but fails to override methods, and *incorrect types* represents that the types of arguments in callsites are incorrect.

TABLE 2: Examples of Invoking Undefined Methods

Project	Classes	ChatGPT's CallSite	Correct CallSite
trove	TFloatDoubleHash	hash.get(val)	hash.index(val)
trove	TFloatDoubleHash	hash.put(3, 4.0f)	hash.insertKeyAt(3, 4.0f)
24_saxpath	XPathLexer	token.getStart()	token.getTokenBegin()
24_saxpath	XPathLexer	token.getType()	token.getTokenType()
73_fim1	UpdateUserPanel	user.setUsername	user.setName()

To wrap up, the compiling errors made by ChatGPT are mainly due to that it fails to have an overview of the entire project. Thus, ChatGPT attempts to predict the callees' names, parameters, parameters' types, and so forth. As a result, compiling errors are introduced.

▷ For (3), we leverage the state-of-the-art static analyzer, SpotBugs, to scan the test cases generated by ChatGPT. As a result, SpotBugs report 403 potential bugs from 204 test cases (3 test cases fail to compile). The overview distribution is shown in Table. 3. On average, each case contains 1.97 bugs.

TABLE 3: Bug Pattern Overview for Test Cased generated by ChatGPT

Num. of Potential Bugs	Num. of Class
Over 20	3 (1.47%)
10 - 20	7 (3.43%)
1 - 9	69 (33.8%)
0	125 (61.2%)

TABLE 4: Bug Patterns' Priority Levels

Priority Level	# Bugs	# Related Test Cases	Average
Scariest	15	8 (3.9%)	1.87
Scary	35	12 (5.8%)	2.91
Troubling	10	7 (3.4%)	1.42
Of Concern	343	70 (34.3%)	4.9

TABLE 5: Bug Patterns

Bug Patterns	# Bugs	# Related Test Cases	Average
Bad Practice	65	20 (9.8%)	3.25
Performance	36	19 (9.4%)	1.89
Correctness	52	20 (9.8%)	2.6
Multi-thread Correctness	1	1 (0.49%	1
Dodgy Code	199	45 (22.2%)	4.42
Internationalization	47	10 (4.9%)	4.7
Experimental	3	2 (0.98%)	1.5

From the **bug priority levels** perspective, SpotBugs rank bugs' priority level into *Scariest*, *Scary*, *Troubling*, and *Of Concern*. *Scariest* level represents bugs that are considered the most severe and potentially harmful to the overall functionality and security of the code. These bugs should be fixed immediately. They can indicate severe security vulnerabilities, data loss, crashes, and so forth. For example, as shown in Listing 1, the code `str.toString()` is annotated as a Sacriest bug for its NullPointerException bug.

Listing 1: Scariest Bug Sample

```
public void method(){
    String str = null;
    str.toString(); // [NullPointerException bug]
}
```

Furthermore, we carefully inspect each *Scariest* test cases generated ChatGPT, we find that all *Scariest* test cases generated ChatGPT fall into one category, which is "comparing incompatible types". For example, Listing 2 shows a *Scariest* bug made by GPT. The statement 'assertEquals(input, Variable.getVariableValue(operand))' compares two different types. The `input` is a String value, whereas the `Variable.getVariableValue(operand)` returns an Integer value.

Listing 2: Scariest Bug in class jipa.Main (ChatGPT)

```
public void testProcessInstruction_in() {
        String operand = "var1";
        String input = "123";
        ...
        assertEquals(input, Variable.getVariableValue(operand));
}
```

Scary level represents bugs are considered significant and could lead to issues if not fixed. These bugs are high priority bugs but as severe as the "scariest" ones. They still indicate issues that should be addressed to maintain a high-quality codebase. For example, as shown in Listing 3, the code is annotated as a Scary bug as there is a potential infinite Loop if `someLargeNumber` is calculated incorrectly.

Listing 3: Scary Bug Sample

```
public void method2(){
    int i = 0;
    while (i < someLargeNumber){
        ...
        i++;
    }
}
```

Troubling level represents bugs are categorized as minor but could still cause issues if left unaddressed. These bugs are medium priority bugs that are concerning but have a more limited impact. Things like unchecked exceptions, unnecessary object creations belong to this category. For example, the code Listing in 4 is an example of Troubling bug. However, there is a potential Troubling bug issue. If `b` is 0, this will result in a `java.lang.ArithmeticException` issue due to division by 0.

Listing 4: Troubling Bug Sample

```
public int divide(int a, int b){
    int result;
    result = a/b;
    return result;
}
```

Last, *Of Concern* level represents bugs are considered informational and generally pose minimal to no risk to the code's functionality or security. These "Of Concern" bugs are typically less severe and might be related to style and best practices, such as, deal code, misuses of static fields. For example, as shown in Listing 5, writing to static field `Main.TOTAL_INSTRUCTION` is an Of Concern bug.

Listing 5: Of Concern Bug Sample

```
public void method3(){
    Main.TOTAL_INSTRUCTION = 5;
}
```

As shown in Table. 4, most bugs (85.11%) are with the *Of Concern* type. There are only 8 test cases (3.9%) that have *Scariest*-level bugs. Furthermore, we manually inspect the 8 test cases that have *Scariest*-level bugs. We find that all *Scariest*-level bugs fall into one issue, which is "comparing incompatibible type for equiality". For example, the code `assertEquals(input, Variable.getVariableValue(operand))` indicates such a problem. The type of `input` is String, while the type of the `Variable.getVariableValue(operand)` is Integer.

From the **bug patterns** perspective, founded bugs fall into 7 categories: (1) Bad Practice; (2) Performance; (3) Correctness; (4)Multi-thread Correctness; (5) Dodgy Code; (6) Internationalization; and (7) Experimental. The detailed descriptions of each bug pattern can be found on the official documentation [42]. As shown in Table. 5, there are 21 test cases involved either in correctness bugs or multi-thread correctness bugs. These types of bugs represent appear coding mistakes, which normally belong to the Scariest or Scary priority level. As for Dodgy code pattern, which holds

the largest proportion, it represents the code is confusing, anomalous, or written in a way that leads itself to errors. Example cases can be dead local stores, switch fall through, and unconfirmed casts. As for *correctness/multi-thread correctness* bugs, it mostly refers to the following 3 cases based on our results: null dereference, out-of-bounds array access, and unused variables.

It is noteworthy that SpotBugs adopts a comprehensive approach to identifying issues, covering not only crash bugs and compilation problems but also delving into style concerns, best practice violations, and potential logical errors. Our evaluation involved a meticulous manual review of SpotBugs' outcomes, with a specific emphasis on pinpointing issues related to crashes and compilation. In summary, our examination revealed a total of 22 bugs, comprising both compilation-related and crash-related issues, across 10 test cases. Specifically, 8 test cases involve the "comparing incompatible type for equality" errors; 1 case involves the "read of unwritten field" bugs; and 1 case involves the "impossible cast exception" bugs. Developers are likely to find it relatively easy to fix the "Comparing Incompatible Type for Equality" errors, especially with the assistance of modern IDEs, which offer real-time error highlighting and auto-correction. However, addressing the "Read of Unwritten Field" bugs and "Impossible Cast Exception" bugs may be more challenging, as these issues often require a deeper understanding of the code logic and may involve manual examination and restructuring. IDEs can still be helpful in identifying problematic areas, but developers will need to rely on their debugging skills and code analysis to address these issues effectively.

In summary, from the bug priority levels and bug patterns, we can conclude that most (61.2%) ChatGPT-generated test cases are bug-free. Only 20 (9.8%) test cases are from the Scariest and Scary levels.

▷ **Validity of test cases generated by EvoSuite.** For fairness, we also run EvoSuite once to generate test cases for 207 Java classes. Note that for fairness, we also apply Long-input setting in Sec. 3.3 to remove 41 classes and remain 207 Java classes from 75 projects. Furthermore, there are 3 test cases cannot be compiled. Therefore, the experiments on EvoSuite are conducted based on the rest 204 unit test cases. EvoSuite can successfully generate test cases for 204 Java classes. All generated test cases can be successfully compiled and executed.

Furthermore, we leverage SpotBugs to scan the test cases generated by EvoSuite. In total, SpotBugs reports 1,371 potential bugs from 204 test cases. The overview distribution is shown in Table. 6. On average, each case contains 6.72 bugs. A notable portion of the classes (51 out of 204, or 25%) are free of potential bugs as identified by Evosuite. This is a positive indicator, showing that a quarter of the analyzed classes do not exhibit detectable bug patterns with the current testing approach. The data also suggests a skewed distribution of bugs across classes, with most classes having few to no bugs and a small number having a large number of bugs.

As shown in Tab. 7, the "Of Concern" category, with 1294 bugs across 146 test cases, suggests a high concentration of lower severity bugs. This indicates that while there are many bugs, most of them might not be critically impacting the software's functionality. The "Scariest" category, despite having only 23 bugs, is significant due to its potential impact. Furthermore, we dive into the *Scariest* bugs introduced by EvoSuite. We find that the *Scariest* bugs mainly fall into 3 categories: incompatible types comparison (18), suspicious calls to generic collection methods (4), and collections should not contain themselves (1). Similar to ChatGPT, the majority of *Scariest* bugs are comparing incompatible types as demonstrated in Listing 6. There are 4 bugs about "suspicious calls to generic collection methods", which usually point to potential issues with how generic collections are used, particularly regarding type safety or unexpected behavior due to incorrect assumptions about types. For example, in Listing 7, the argument of `treeList3.indexOf()` should be an Integer instead of a TreeList. There is one about "collections should not contain themselves" (as depicted in Listing 8), which typically occurs when there is an attempt to include a collection as an element within itself, either directly or indirectly. This kind of operation can lead to unexpected behaviors, such as infinite recursion in methods that traverse the collection.

TABLE 6: Bug Pattern Overview for Test Cases generated by Evosuite

Num. of Potential Bugs	Num. of Class
Over 20	15 (7.36%)
10 - 20	22 (10.78%)
1 - 9	116 (56.86%)
0	51 (25%)

TABLE 7: Bug Patterns' Priority Levels for Test Cases generated by Evosuite

Priority Level	# Bugs	# Related Test Cases	Average
Scariest	23	17 (8.33%)	1.35
Scary	47	10 (4.90%)	4.7
Troubling	7	3 (1.47%)	2.33
Of Concern	1294	146 (71.56%)	8.86

TABLE 8: Bug Patterns for Test Cases generated by EvoSuite

Bug Patterns	# Bugs	# Related Test Cases	Average
Bad Practice	3	3 (1.47%)	1
Performance	115	17 (8.33%)	6.76
Correctness	69	25(12.25%)	2.76
Multi-thread Correctness	1	1 (0.49%)	1
Dodgy Code	1181	143 (70.09%)	8.25
Internationalization	2	2 (0.98%)	1
Experimental	0	0 (0%)	0

Listing 6: Scariest Bug in class Option_ESTest (EvoSuite/Incompatibility)

```
public void test48() throws Throwable {
    Option option0 = new Option(...);
    boolean boolean0 = option0.equals("6pP");
    ...}
```

Listing 7: Scariest Bug in class TreeList_ESTest (EvoSuite/Suspicious calls to generic collection methods)

```
TreeList<Integer> treeList3 = new TreeList<Integer>();
int int0 = treeList3.indexOf(treeList2);
```

Listing 8: Scariest Bug in class DenseInstance (EvoSuite/Collections should not contain themselves)

```
public void test43() throws Throwable {
    DenseInstance denseInstance0 = new DenseInstance(31);
    boolean boolean0 = denseInstance0.containsValue(denseInstance0);
}
```

The fact that these bugs appear in 17 test cases indicates that they might be pervasive or have a high impact where they do occur. As shown in Table. 8, there are 70 test cases involved either in correctness bugs or multi-thread correctness bugs. These types of bugs represent coding mistakes, which normally belong to the Scariest or Scary priority level.

By comparing the experiment results, we find that ChatGPT's generated test cases have a significantly higher proportion with no potential bugs (60.2%) compared to Evosuite (25%), indicating a cleaner output. Despite this, Evosuite exhibits fewer test cases with a high concentration of bugs (over 20). Notably, the average score for 'Of Concern' bugs is lower for ChatGPT (4.9), suggesting that the bugs present are generally of a lower priority compared to those in Evosuite's test cases (8.86). Additionally, Evosuite seems to produce a considerable amount of 'Dodgy Code', with the average severity almost double that of ChatGPT's. This points to ChatGPT's generated tests being less prone to serious bugs and Evosuite's to higher occurrences of non-critical issues.

Answer to RQ1: Validity

- After analyzing the bug priority levels and bug patterns of ChatGPT-generated test cases, it can be inferred that a majority of these cases, specifically 61.2%, are free from any bugs. However, a small proportion of test cases, comprising only 9.8%, have been categorized under the Scariest and Scary levels, indicating the presence of severe issues.
- By comparing the experiment results, we find that ChatGPT's generated test cases have a significantly higher proportion with no potential bugs (60.2%) compared to Evosuite (26.09%), indicating a cleaner output. Despite this, Evosuite exhibits fewer test cases with a high concentration of bugs (over 20). Notably, the average score for 'Of Concern' bugs is lower for ChatGPT (4.9), suggesting that the bugs present are generally of a lower priority compared to those in Evosuite's test cases (8.86). Additionally, Evosuite seems to produce a considerable amount of 'Dodgy Code', with the average severity almost double that of ChatGPT's. This points to ChatGPT's generated tests being less prone to serious bugs and Evosuite's to higher occurrences of non-critical issues.

4.2 Readability

RQ2: How Understandable is the Test Suite Provided by ChatGPT Compared to That of Evosuite?

Motivation. Analyzing the readability of ChatGPT-generated code is to make sure that human developers can easily maintain, comprehend, and modify it. This is crucial when ChatGPT-generated code will be maintained and changed over time by other developers or when it will be merged into already-existing codebases.

Methodology. For this RQ, we set up two sub-tasks: (1) code style checking; and (2) code understandability.

▷ To check code styles of generated test cases, we rely on the state-of-the-art software quality tool which supports Java: Checkstyle [43], which is a development tool to check whether Java code adheres to a coding standard. It automates the process of checking Java code. Here, we leverage two standards (i.e., Sun Code Conventions [44], Google Java Style [45]) with Checkstyle to check whether the ChatGPT generated test suite adheres to these standards.

▷ Dantas et al. [46] proposed cognitive complexity and cyclomatic complexity metrics for measuring the understandability of a code snippet. Cyclomatic complexity measures program complexity by counting independent paths in source code. It indicates code size, structure, and complexity, and helps find error-prone areas. Cognitive complexity is a metric that evaluates code complexity from a human perspective. It considers factors like code structure, naming, and indentation to determine how hard code is to understand. It helps developers gauge maintainability and modification difficulty and identifies complex or confusing code parts. Cyclomatic and cognitive complexity can be measured with the PMD IntelliJ plugin [47]. The details can be found on the project repository (Sec. 8).

Results. According to the *Long-input setting* in Sec. 3.3, we remove 41 classes and remain 207 Java classes from 75 projects. Furthermore, there are 3 test cases cannot be compiled. Therefore, the experiments are conducted based on the rest 204 unit test cases.

▷ **Readability of test cases generated by ChatGPT.**

- **Code Style Checking Results.**

▷ Checkstyle-Google: Fig. 3 shows the boxplot of Google Codestyle violations for each class. It shows that the dataset has several outliers on the higher side, with a median value of approximately 70. The interquartile range (IQR) falls between around 30 to 175, indicating that most of the data lie within this range. However, the data is highly skewed to the right, with a few extreme data points on the higher side, indicating that the distribution does not follow the expected pattern. The minimum value is 4, and the maximum value is 1260, which shows a wide range of values in the dataset.

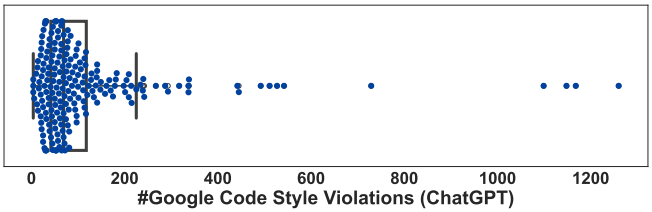

Fig. 3: Boxplot of Google Code Style Violations (ChatGPT)

Next, The radar plot in Fig. 4 breakdowns violation issues by types to display the details. As depicted in Fig. 4, we can conclude that:

- `Indentation` is the most common code style violation, indicating that ChatGPT may need to work on consistently formatting its code to improve readability and maintainability;
- `FileTabCharacter` and `CustomImportOrder` also appear to be frequent violations, which highlights the importance of proper configuration and consistency in code structure; and
- Violations related to code legibility and ease of reading, such as `LineLength` and `AvoidStarImport` should not be ignored to maintain a high standard of code quality.

▷ Checkstyle-SUN: Fig. 5 shows the boxplot. The median value of the data is around 28, with 25% of the data falling

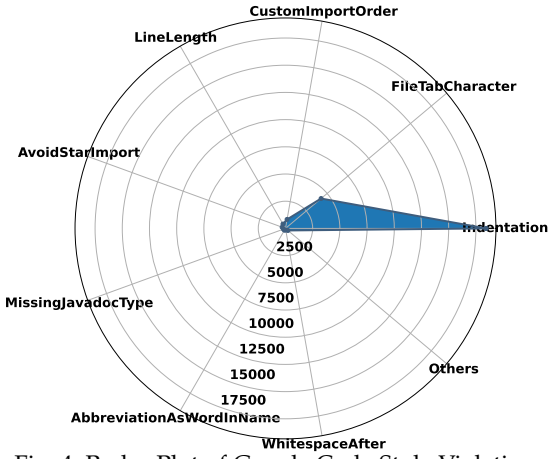

Fig. 4: Radar Plot of Google Code Style Violations

below 15 and 75% falling below 55. There are several values above the upper quartile, indicating potential outliers or extreme values. The minimum value in the data is 3 and the maximum is 297. The IQR for the dataset is 40, indicating that most of the values in the dataset fall within this range.

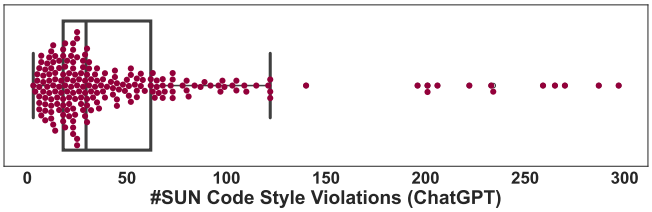

Fig. 5: Boxplot for SUN Code Style Violations

Next, The radar plot in Fig. 6 breakdowns violation issues by types to display the details. As depicted in Fig. 6, it appears that the two most common types of coding issues are `MissingJavadocMethod` and `MagicNumber`, with 2742 and 2498 occurrences respectively. The `MissingJavadocMethod` issue suggests that more documentation and explanations are required for ChatGPT. Furthermore, magic numbers in the test cases generated by ChatGPT are mainly used in the assertions. Additionally, the figure shows that `FinalParameters`, `RegexpSingleline`, and `AvoidStarImport` also occur frequently, indicating that attention should be paid to these areas as well. Some of the less frequent issues, such as `HiddenField` and `UnusedImports`, may be less urgent but still worth addressing to improve overall code quality for ChatGPT.

In summary, as an AI language model, ChatGPT may not have a specific code style that it adheres to when generating test cases. However, the code style of the test cases can be influenced by the parameters and rules set for the generation process or the input that is given to the model. It also suggests that programmers should pay attention to the code style when using test cases generated by ChatGPT.

• **Code Complexity.** The default cyclomatic and cognitive complexity thresholds in PMD are 10 and 15, which means if the cyclomatic and cognitive complexities of a class/method are lower than these values, the system does not report the issue. Thus, we build a series of customized rules to measure complexity. The rule sets can be downloaded from our online repository. Note that, the complexity is measured on a method basis.

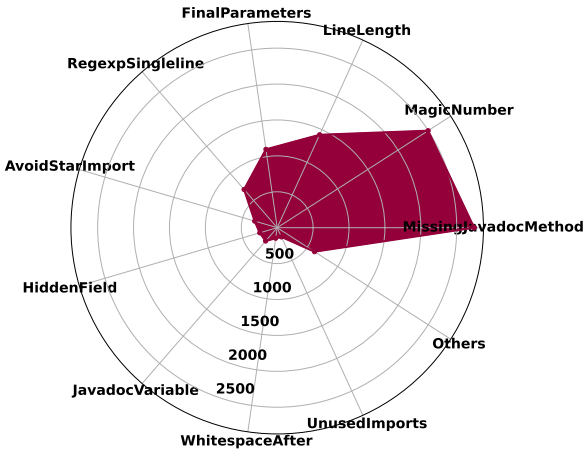

Fig. 6: Radar Plot of SUN Code Style Violations

▷ Cognitive Complexity: Based on the technical report from SonarSource [48], Cognitive Complexity can be categorized into four categories: low (<5 cognitive complexity), moderate (6-10), high (11-20), and very high complexity (21+). As the results are shown in Table. 9, all methods are with low complexity.

TABLE 9: Cognitive Complexity Results Overview

Cognitive Complexity Level	Num. of Class	Num. of Methods
Low complexity (<5)	204	3302
Moderate complexity (6-10)	0	0
High complexity (11-20)	0	0
Very High complexity (21+)	0	0

TABLE 10: Cyclomatic Complexity Results Overview

Cyclomatic Complexity Level	Num. of Class	Num. of Methods
Low complexity (1-4)	204	3300
Moderate complexity (5-7)	2	2
High complexity (8-10)	0	0
Very High complexity (11+)	0	0

▷ Cyclomatic Complexity: Based on the official documentation from PMD [47], Cyclomatic Complexity can be categorized into four categories: low (1-4 cyclomatic complexity), moderate (5-7), high (8-10), and very high complexity (11+). As the results are shown in Table. 10, there are 3300 methods from 204 classes with low complexity and 2 methods from 2 classes with moderate complexity.

User Study on ChatGPT-generated Test Cases. Furthermore, we invite 5 Java programmers with at least 3 years of Java programming experience to assess the readability of generated unit test cases. Developers are required to read the code, and then answer the following questions:

• On a scale of 1-5 (1 (very poor) to 5 (excellent)), how readable do you find this code?

• What aspects of the code made it easy or difficult to understand?

We collect developers' feedback on the test cases and the average readability score for each test case. The Pareto Chart depicted in Fig. 7 illustrates that the average readability score for ChatGPT-generated test cases is 3.92. Out of the

total 1020 collected readability scores (calculated as 5*204), the distribution is as follows: 159 scores at 5, 632 scores at 4, 218 scores at 3, and 11 scores at 2. The majority of test cases fall into the high readability category (score 4), suggesting that ChatGPT tends to generate test code that is relatively easy to understand.

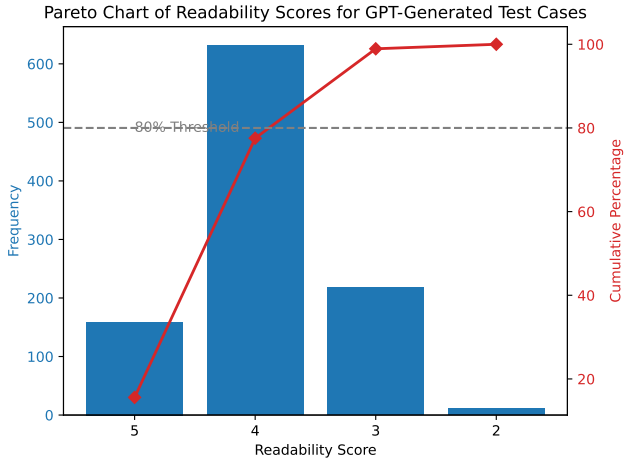

Fig. 7: Pareto Chat of Readability Score for GPT-generated Test Cases

Nevertheless, the results highlight that ChatGPT occasionally produces low-readability test cases, as observed in 11 (with a score of 2) out of 1020 instances. We conducted a more in-depth examination of the 11 cases that received a rating of 2, revealing several prevalent issues that adversely affect the readability of the generated code:

• **Complex Syntax and execution logic.** ChatGPT may generate test cases with unnecessarily complex syntax or convoluted logic, making it difficult for developers to follow and maintain the code. For example, for the `charting.CoordSystemUtitlites`, developers comment that the code contains a complex setup in the setUp method with multiple anonymous inner classes and objects that implement various interfaces. This setup makes developers hard to follow the code's logic.

• **Lack of Full Context.** ChatGPT may not fully understand the specific context or requirements of the software being tested, leading to test cases that are irrelevant or confusing.

• **Lack of Proper Organization and Structure.** For example, for the `agents.GroupAgent`, one developer complains that the code lacks proper organization and structure. It's challenging to discern the purpose and relationships between different parts of the code; and

• **Lack of Comments and Explanation.** ChatGPT may generate comments that are inaccurate or misleading, which can lead to confusion or incorrect assumptions about the test cases.

Therefore, based on all the aforementioned results, we can conclude that the ChatGPT-generated test cases are overwhelmingly easy to follow and in low complexity.

▷ **Readability of test cases generated by EvoSuite.**

• **Code Style Checking Results.**

▷ Checkstyle-Google: Fig. 8 shows the boxplot of Google Codestyle violations for each class generated with EvoSuite. The dataset has several outliers on the higher side, with a median value of approximately 234. The interquartile range (IQR) falls between around 101 to 445.75, indicating that most of the data lie within this range. The minimum value is 14, and the maximum value is 11,002, which shows a wide range of values in the dataset.

Fig. 8: Boxplot of Google Code Style Violations (EvoSuite)

▷ Checkstyle-SUN: Fig. 9 shows the boxplot of SUN Codestyle violations for each class generated with EvoSuite. It reveals a median value of approximately 153.5, indicating that half of the values are below this point. The 25th percentile (Q1) is around 65.75, and the 75th percentile (Q3) is about 321.75, suggesting that the middle 50% of the data is spread over a range of 256 (the Interquartile Range, IQR). The dataset has a minimum value of 7 and a maximum of 4546, with several values, especially the maximum, significantly higher than the upper quartile, pointing to potential outliers or extreme values.

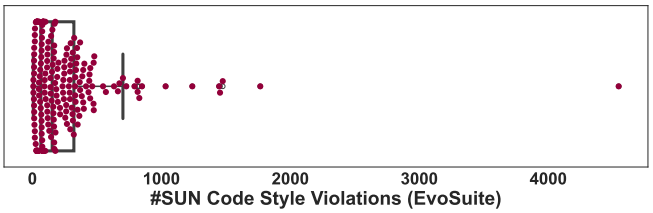

Fig. 9: Boxplot for SUN Code Style Violations (EvoSuite)

▷ Cognitive Complexity: As the results are shown in Table. 11, all methods are with low complexity.

TABLE 11: Cognitive Complexity Results Overview

Cognitive Complexity Level	Num. of Class	Num. of Methods
Low complexity (<5)	204	8115
Moderate complexity (6-10)	0	0
High complexity (11-20)	0	0
Very High complexity (21+)	0	0

TABLE 12: Cyclomatic Complexity Results Overview

Cyclomatic Complexity Level	Num. of Class	Num. of Methods
Low complexity (1-4)	204	8115
Moderate complexity (5-7)	0	0
High complexity (8-10)	0	0
Very High complexity (11+)	0	0

▷ Cyclomatic Complexity: As the results are shown in Table. 12, there are 8,115 methods from 204 classes with low complexity.

User Study on Evosuite-generated Test Cases. The same user study (i.e., same study setting) is conducted on Evosuite-generated test cases with the same users. We collect developers' feedback on the test cases and the average readability score for each test case. The Pareto Chart depicted in Fig. 10 illustrates that the average readability

score for ChatGPT-generated test cases is 3.89. Out of the total 1020 collected readability scores (calculated as 5*204), the distribution is as follows: 130 scores at 5, 652 scores at 4, and 238 scores at 3. The majority of test cases fall into the high readability category (score 4), suggesting that EvoSuite tends to generate test code that is relatively easy to understand.

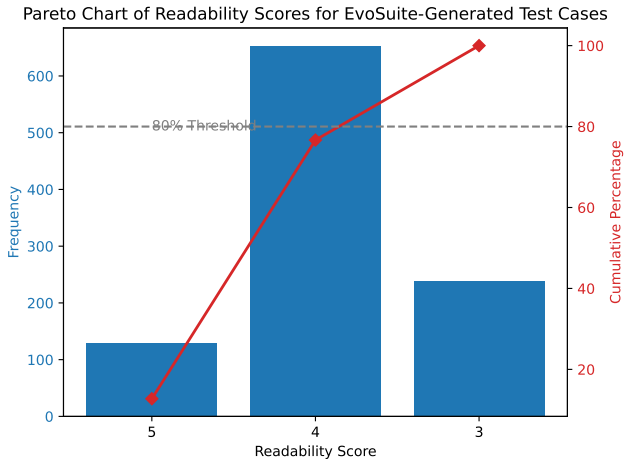

Fig. 10: Pareto Chat of Readability Score for EvoSuite-generated Test Cases

• **Readability Comparison between Unit Test Suite generated by ChatGPT and EvoSuite.** In summary, the readability comparison between unit test suites generated by ChatGPT and EvoSuite reveals that both tools are capable of producing highly readable and understandable tests, albeit through different approaches.

Answer to RQ2: Readability

• **Code Style-Google Rule** The median value of approximately 70 (violations). The interquartile range (IQR) falls between around 30 to 175, indicating that most of the data lie within this range. Furthermore, `Indentation` is the most common code style violation;
• **Code Style-SUN Rule** The median value of the data is around 28 (violations), with 25% of the data falling below 15 and 75% falling below 55. The two most common types of coding issues are `MissingJavadocMethod` and `MagicNumber`, with 2742 and 2498 occurrences respectively; and
• **Code Complexity.** From the cognitive complexity perspective, all methods are in low complexity. From the cyclomatic complexity perspective, almost all (3300 out of 3302) methods are in low complexity and the other 2 methods are in moderate complexity. Thus, the ChatGPT-generated test cases are overwhelmingly easy to follow and with low complexity.
• The readability comparison between unit test suites generated by ChatGPT and EvoSuite reveals that both tools are capable of producing highly readable and understandable tests, albeit through different approaches.

4.3 Code Coverage

RQ3: How does ChatGPT perform with SBST in terms of code coverage?

Motivation. While low coverage implies that certain portions of the code have not been checked, high coverage shows that the produced tests have thoroughly evaluated the code. Comparing the code coverage between the test suite generated by ChatGPT and SBST allow us to evaluate and assess the ChatGPT-generated test suite.

Methodology. JaCoCo [49] measures instruction and branch coverage. The instruction coverage relates to Java bytecode instructions and is thus analogous to statement coverage on source code. We use just instruction coverage (i.e., statement coverage (SC)) to evaluate code coverage as JaCoCo's definition of branch coverage counts only branching of conditional statements, nor edges in the control flow graph.

Results. According to the *Long-input setting* in Sec. 3.3, we remove 41 classes and remain 207 Java classes from 75 projects.

TABLE 13: Statement Code Coverage for Project (I)

Projects	(A)Max	(A)Min	(A)SDEV.	(A)Avg.	(A)ChatGPT
1_tullibee	100%	100%	0.00	100%	93%
100_jgaap	95.0%	95.0%	0.00	95.0%	83%
105_freemind	71.5%	64.5%	1.56	69.1%	52%
107_weka	87.0%	79.0%	2.95	83.0%	37%
11_imsmart	100%	100%	0.00	100%	100%
12_dsachat	35.5%	35.5%	0.00	35.5%	34%
14_omjstate	67.0%	67.0%	0.00	67.0%	55%
15_beanbin	80.5%	80.5%	0.00	80.5%	46%
17_inspirento	94.0%	92.5%	0.33	94.0%	87.5%
2_a4j	50.5%	44.0%	1.54	48.5%	31%
21_geo-google	54.0%	54.0%	0.00	54.0%	67%
24_saxpath	97.0%	96.0%	0.34	96.5%	95%
26_jipa	88.0%	73.5%	3.63	83.50%	97%
29_apbsmem	98.0%	98.0%	0.00	98.0%	80%
31_xisemele	71.0%	71.0%	0.00	71.0%	75%
33_javaviewcontrol	82.0%	62.5%	6.13	76.0%	46%
35_corina	85.0%	75.0%	3.69	78.0%	65%
36_schemaspy	100.0%	100.0%	0.00	100.0%	67%
39_diffi	99.0%	93.0%	3.02	95.5%	69.5%
4_rif	100.0%	100.0%	0.00	100.00%	96%
40_glengineer	97.0%	86.5%	3.37	95.0%	73%
41_follow	92.5%	71.0%	5.73	82.0%	38%
43_lilith	100.0%	100.0%	0.00	100.0%	95%
45_lotus	70.5%	70.5%	0.00	70.5%	75%
47_dvd-homevideo	13.3%	13.3%	0.00	13.3%	0.7%
51_jiprof	96.5%	78.0%	3.76	93.0%	44.5%
52_lagoon	19.5%	14.0%	1.15	18.0%	27%
55_lavalamp	100.0%	100.0%	0.00	100.0%	100%
60_sugar	96.0%	87.5%	2.47	90.0%	79%
61_noen	82.5%	81.5%	0.18	81.5%	60%
63_objectexplorer	51.5%	51.5%	0.00	51.5%	47%
64_jtailgui	76.5%	17.0%	16.41	70.0%	0%
68_biblestudy	81.5%	81.5%	0.00	81.5%	57%
69_lhamacaw	43.5%	43.5%	0.00	43.5%	6%
7_sfmis	100.0%	100.0%	0.00	100.0%	87%
72_battlecry	1.0%	1.0%	0.00	1.0%	57%
73_fim1	24.0%	24.0%	0.00	24.0%	44.5%
74_fixsuite	67.5%	50.0%	6.43	54.5%	40%
77_io-project	100.0%	100.0%	0.00	100.0%	71%
78_caloriecount	92.7%	88.3%	1.24	89.7%	46.7%
79_twfbplayer	97.5%	95.5%	0.53	96.5%	69.5%
8_gfarcegestionfa	68.0%	62.5%	1.31	65.0%	55%
80_wheelwebtool	84.3%	83.0%	0.31	83.3%	36%
82_ipcalculator	91.5%	81.0%	4.07	85.0%	73%
83_xbus	34.0%	19.0%	6.75	23.00%	33%
84_ifx-framework	55.0%	55.0%	0.00	55.0%	32%
85_shop	71.5%	55.8%	4.22	63.8%	24.8%
86_at-robots2-j	86.0%	48.0%	15.02	58.0%	45%
87_jaw-br	32.0%	31.0%	0.18	32.0%	17.5%
88_jopenchart	99.5%	72.0%	10.87	78.5%	52%
89_jiggler	91.0%	81.7%	2.10	89.7%	30.3%
90_dcparseargs	100.0%	94.0%	1.22	99.0%	75%
91_classviewer	93.0%	91.0%	0.29	92.5%	73%
92_jcvi-javacommon	100.0%	100.0%	0.00	100.0%	74%
94_jclo	82.0%	68.0%	4.31	74.0%	11%
95_celwars2009	47.0%	47.0%	0.00	47.0%	46%
97_feudalismgame	25.0%	19.5%	2.06	21.1%	15%
98_trans-locator	50.0%	47.0%	0.57	50.0%	15%
99_newzgrabber	20.7%	17.7%	0.76	20.3%	10.7%

Note: The colored cells in Tables 8 and 9 indicate cases where ChatGPT outperforms EvoSuite in terms of statement code coverage. However, there is no cell colored in Table 9 because EvoSuite outperforms ChatGPT for all projects in that table.

▷**Statement Coverage (SC) Comparison.** As we run 30

TABLE 14: Statement Code Coverage for Project (II)

Projects	(A)Max	(A)Min	(A)SDEV.	(A)Avg.	(A)ChatGPT
checkstyle	87.5%	79.3%	3.28	84.7%	65.2%
commons-cli	98.5%	95.0%	1.09	98.1%	69%
commons-collections	94.3%	89.3%	0.91	94.1%	68%
commons-lang	94.0%	86.0%	2.36	90.1%	73.1%
commons-math	72.7%	64.1%	3.17	69.0%	45.6%
compiler	67.7%	36.9%	9.48	53.9%	6.29%
guava	75.0%	70.1%	1.28	72.9%	63.1%
javaml	97.1%	87.3%	2.46	96.4%	76.1%
javex	94.0%	67.0%	12.59	81.2%	63%
jdom	80.7%	80.5%	0.06	80.7%	31.3%
joda	94.9%	92.4%	0.64	93.9%	71.6%
jsci	97.0%	86.0%	2.62	92.4%	50%
scribe	95.3%	95.3%	0.00	95.3%	91.2%
trove	81.0%	76.7%	1.26	79.3%	45.3%
twitter4j	92.2%	89.7%	0.67	91.3%	70.7%
xmlenc	97.0%	94.0%	0.61	95.1%	54%
Overall Avg. (Project)	77.4%	70.6%	-	74.5%	55.4%

times for EvoSuite, we compute the maximum, minimum, average, and average standard deviation. Recall the result in RQ1, for the 3 ChatGPT-generated test cases, which failed to be fixed without the background knowledge, we regard their code coverage as 0 [2].

As shown in Table 13 and 14, for Evosuite, on average, the maximum SC can reach 77.4% for all projects; the minimum SC can reach 70.6% for all projects; and the average SC can reach 74.2% for all projects. In contrast, for ChatGPT, on average, the average SC can reach 55.4% for all projects. In general, Evosuite outperforms ChatGPT 19.1% in regards to SC. Additionally, ChatGPT outperforms Evosuite in 10 out of 75 (13.33%) projects, which are highlighted in Table. 13 and 14. From the class perspective, ChatGPT outperforms EvoSuite in 23 (11.11%) out of 207 classes.

Furthermore, by investing 23 cases that ChatGPT outperforms EvoSuite, we find that ChatGPT is well performed in generating test cases for the following reasons:

1. ChatGPT can generate different String objects/integer/double values to use (e.g., comparison) with high diversity compared to Evosuite (Ref: `guava::Objects`, `math::SimplexTableu`). Number of cases: 3;
2. ChatGPT can generate an instance of Font for FontChooser, which is not applicable for Evosuite (Ref: `71_film2::FontChooserDialog`) Number of cases: 1;
3. ChatGPT can generate more reasonable and useable UI operations (i.e., ActionEvents) for testing UIs compared to Evosuite (Ref: `72_bcry::battlecryGUI`) Number of cases: 3;
4. ChatGPT can generate test cases or instances based on the existing information from the classes under tests (Ref: `45_lotus::Phase`). Fig. 11 shows a code segment from `45-lotus::Phase.java`. This code segment also suggests some instances (e.g, `UpkeepPhase()`, `DrawPhase()`, `Main1Phase()`) are compatible with the type of `Game.currentPhase`. Such information can be correctly captured by ChatGPT and be used to generate diverse `Phase` instances. As a result, it can reach a high coverage than EvoSuite Number of cases: 1;
5. ChatGPT can generate more complex call chains for testing based on the semantics information collected from the classes under test compared to EvoSuite (Ref `guava::Monitor`). For example, the code segment in Fig. 12, ChatGPT can generate a more complex call chain rather than invoking a single method once. More importantly, its call chain is logically correct. That is, the method `enter` must be invoked before `leave`. This can benefit from that the LLM can precept semantic context from the code or identifiers. Number of cases: 7 and;

2. Different from 204 test cases in other RQs, we have 207 test cases considered in this RQ.

```
if(Game.currentPhase instanceof UntapPhase) changePhase(new UpkeepPhase());
else if(Game.currentPhase instanceof UpkeepPhase) changePhase(new DrawPhase());
else if(Game.currentPhase instanceof DrawPhase) changePhase(new Main1Phase());
else if(Game.currentPhase instanceof Main1Phase) changePhase(new
CombatBeginningPhase());
else if(Game.currentPhase instanceof CombatBeginningPhase) changePhase(new
DeclareAttackersPhase());
else if(Game.currentPhase instanceof DeclareAttackersPhase) changePhase(new
DeclareBlockersPhase());
else if(Game.currentPhase instanceof DeclareBlockersPhase) changePhase(new
CombatDamagePhase());
else if(Game.currentPhase instanceof CombatDamagePhase) changePhase(new
CombatEndPhase());
else if(Game.currentPhase instanceof CombatEndPhase) changePhase(new
Main2Phase());
else if(Game.currentPhase instanceof Main2Phase) changePhase(new
EndOfTurnPhase());
else if(Game.currentPhase instanceof EndOfTurnPhase) changePhase(new
CleanupPhase());
else if(Game.currentPhase instanceof CleanupPhase) changePhase(new
PlayerChangePhase());
else if(Game.currentPhase instanceof PlayerChangePhase)
```

Fig. 11: The Code Segment from 45-lotus::Phase

```
@Test
public void testEnterWhen() throws InterruptedException {
    Guard guard = new Guard(monitor) {
        @Override
        public boolean isSatisfied() {
            return true;
        }
    };
    monitor.enterWhen(guard);
    assertTrue(monitor.lock.isLocked());
    monitor.leave();
    assertFalse(monitor.lock.isLocked());
}
```

Fig. 12: The Code Segment from guava::MonitorTest

6. ChatGPT can generate test data that is suitable for the target regarding the semantic context. For example, the input parameter for invoking the method `setCountry` (Ref: `21_geo-google::GeoStatusCode`) can be any String. However, a real country name (e.g., United States) can be more suitable for testing the method `setCountry` compared to a random String Number of cases: 8;

From the breakdown of the cases into the different reasons, we find that ChatGPT performs well in generating complex test inputs with contexts and complex execution logical chains in terms of generate test cases (inputs). Moreover, as the code complexity increases, so does the search space for identifying appropriate test cases, leading to longer execution times and greater computational expenses for SBST techniques. Consequently, this can pose a significant challenge in uncovering effective test cases that can ensure optimal code coverage and expose any defects.

Following the previous research works [4], [50], [51], we adopt Vargha-Delaney $\hat{A}_{ab}$ to evaluate whether a particular approach (a) outperforms another (b). According to Vargha and Delaney $\hat{A}_{ab}$ [51], negligible, small, medium, and large differences are indicated by A12 over 0.56, 0.64, 0.71, and 0.8, respectively.

▷**All Classes Comparison.** As shown in Table. 15, there 193 test cases fall into *large* and 14 test cases fall into *negligible* group. This indicates EvoSuite is overwhelmingly better than ChatGPT in reaching higher code coverage for most cases. The overall Vargha-Delaney measure for all classes is 0.71 (medium).

TABLE 15: Vargha-Delaney Measures for Evosuite vs. ChatGPT

	Large	Medium	Small	Negligible
Num. of Classes	193	0	0	14
Overall V.D.	0.71 (Medium)			

▷**Small/Big Classes Comparison.** Here, small classes are defined as classes with less than 50 branches. Classes with more than 50 branches are considered as big classes.

TABLE 16: Vargha-Delaney Measures for Big Classes

	Large	Medium	Small	Negligible
Num. of Big Classes	121	0	0	5
Overall V.D.	0.764 (Large)			

TABLE 17: Vargha-Delaney Measures for Small Classes

	Large	Medium	Small	Negligible
Num. of Small Classes	70	0	0	11
Overall V.D.	0.63 (Small)			

▷**Big Classes Comparison.** Table. 16 shows the comparison for big classes. There 121 test cases fall into *large* and 5 test cases fall into *negligible* group. This indicates EvoSuite is overwhelmingly better than ChatGPT in reaching higher code coverage for big class cases. The overall Vargha-Delaney measure for all classes is 0.764 (large).

▷**Small Classes Comparison.** Table. 17 shows the comparison for small classes. There 70 test cases fall into *small* and 11 test cases fall into *negligible* group. This indicates EvoSuite is overwhelmingly better than ChatGPT in reaching higher code coverage for small class cases. The overall Vargha-Delaney measure for all classes is 0.63 (small).

Unfortunately, we fail to see ChatGPT outperforms EvoSuite for even big classes. It indicates no matter the big or small classes, developers are suggested to turn to EvoSuite in order to obtain a higher code coverage. The potential causes may be diverse and varied. Some possible reasons can be: (1) **incomplete specifications:** ChatGPT is only given the classes under test instead of the entire project. Thus, without the information from the entire project, it can be hard for ChatGPT to generate more valuable test cases; (2) **lack of feedback mechanisms:** Unlike Evosuit, which can learn from feedback (i.e., cover data), ChatGPT relies solely on the training data. It makes it challenging for ChatGPT to comprehend the feedback from test results through an iterative process leading to low test coverage.

However, the results also suggest two insights:

⋆**Insight 1:** Our study indicates a promising trend wherein ChatGPT demonstrates a notable ability to grasp the semantics and context of the code under test. This initial observation suggests that integrating an AI model, like ChatGPT, with Search-Based Software Testing (SBST) tools could potentially improve their understanding of complex code structures. Such an integration might enable SBST tools to generate test cases that are more precisely aligned with the intricacies of the code semantics. However, it's important to note that this is a preliminary insight based on our current dataset and observations. Further empirical research is needed to explore this potential integration in depth; and

⋆**Insight 2:** Even though it cannot compare with EvoSuite, ChatGPT can still reach a relatively high code coverage (55.4%). Thus, ChatGPT can still serve as an entry-level tool for testing newcomers or as a backup option.

Answer to RQ3: Code Coverage

- For Evosuite, on average, the maximum SC can reach 77.4% for all projects; the minimum SC can reach 70.6%; and the average SC can reach 74.2%. In contrast, for ChatGPT, on average, the average SC can reach 55.4%;
- After examining 23 cases in which ChatGPT outperformed EvoSuite (in code coverage), our analysis suggests six potential scenarios where ChatGPT may be better suited. These findings contribute to a growing body of research exploring the efficacy of automated testing tools;
- The experimental results indicate EvoSuite is overwhelmingly better than ChatGPT in reaching higher code coverage for both big class cases and small class cases; and
- Two potential reasons for low code coverage can be: incomplete specifications; and lack of feedback mechanisms.

4.4 Bug Detection

RQ4: How effective are ChatGPT and SBST at generating test suites that detect bugs?

Motivation. The main use of generated test suites is to find buggy code in a program. Therefore, in this RQ, we evaluate the effectiveness of generated test suite in detecting bugs.

Methodology. To evaluate the effectiveness of generated test suite in terms of detecting bugs, we first generate unit test suites for the target classes and examine whether the test suite can successfully capture the bug in the Defects4J benchmark. Note that, in this RQ, for fairness, we only run EvoSuite once to generate test cases. It is also worth mentioning that, in this RQ, our primary goal is to evaluate the bug detection ability of the unit test cases generated by ChatGPT, rather than treating ChatGPT as a dedicated bug detection or fuzzing tool.

```
@Test
public void testConstructorWithStartAndEndInstant() {
    Instant start = new Instant(0);
    Instant end = new Instant(1000);
    Period p = new Period(start.getMillis(), end.getMillis());
    assertEquals(1000, p.getMillis());
}
```

Fig. 13: The Test Cases for Time Project

Results. Some bugs in the Defects4J are logical bugs, which are triggered with `assertions`. Unfortunately, we find that sometimes the `assertions` generated by ChatGPT are not reliable. For example, Fig. 13 illustrates a test case for `Period` in Time project. The assertion statement `assertEquals(1000, p.getMillis());` is incorrect. However, the code segment under test is not buggy and the expected value should be 0 instead of 1000. ChatGPT makes an incorrect assertion for this case. It means we cannot fully rely on the `assertions` in ChatGPT-generated test cases to determine whether the bugs are successfully triggered. However, manually checking the assertions in ChatGPT-generated test cases can be effort-consuming and error-prone [52], [53], [54]. Therefore, in this RQ, we focus on bugs that associate

with Java Exceptions, such as `NullPointerException`, `UnsupportedOperationException`.

TABLE 18: Bug Detection Comparison for ChatGPT and Evosuite

		ChatGPT		Evosuite	
Project	# All/ # Exce. Bugs	Detected	Coverage	Detected	Coverage
Chart	26 / 8	4 (50%)	62%	3 (38%)	85%
Cli	39 / 8	1 (13%)	70%	2 (25%)	88%
Closure	174 / 9	1 (11%)	14%	0 (0%)	4%
Codec	18 / 7	0 (0%)	60%	2 (29%)	94%
Collections	4 / 2	0 (0%)	87%	0 (0%)	67%
Compress	47 / 19	6 (32%)	42%	3 (16%)	57%
Csv	16 / 7	2 (29%)	80%	5 (71%)	90%
Gson	18 /12	2 (17%)	59%	6 (50%)	55%
JacksonCore	26 / 8	2 (25%)	38%	2 (25%)	64%
JacksonDatabind	112 / 53	9 (17%)	30%	4 (8%)	56%
JacksonXml	6 / 1	0 (0%)	29%	0 (0%)	49%
Jsoup	93 / 22	4 (18%)	63%	10 (45%)	86%
JxPath	22 / 1	1 (100%)	40%	1 (100%)	88%
Lang	64 / 20	6 (30%)	68%	3 (15%)	55%
Math	106 / 28	5 (18%)	64%	12 (43%)	84%
Time	26 / 7	1 (14%)	56%	2 (29%)	88%
Total	796 / 212	44 (21%)	50%	55 (26%)	67%

Table. 18 shows the experimental results. In the table, for each project, the higher values (e.g., higher code coverage) are highlighted in comparison between the two approaches. Furthermore, out of 212 bugs, 44 were successfully detected by test cases generated by ChatGPT, with an average statement code coverage of 50%. In contrast, test cases generated by EvoSuite successfully detected 55 bugs, with an average statement code coverage of 67%. From the comparison, we can also see that in some cases, EvoSuite detected more bugs than ChatGPT, while in other cases, ChatGPT detected more bugs than EvoSuite. For example, in the Chart project, EvoSuite had a higher coverage rate for bug detection than ChatGPT, but ChatGPT detected more bugs than EvoSuite in some cases. It is worth noting that the coverage rates for both tools varied greatly across different projects, indicating that the effectiveness of each tool may depend on the specific characteristics of the project being tested. It is interesting to note that ChatGPT was able to detect bugs in some cases where EvoSuite was not, indicating that the two tools may complement each other and could be used together to improve bug detection.

By comparing the test cases generated by ChatGPT and EvoSuite, we find several possible reasons that LLM (e.g., ChatGPT) may not outperform Evosuite:

• As ChatGPT can only take text as input, instead of the binary representation of the entire project (e.g., a Jar file), it can be challenging for ChatGPT to generate complex instances without a complete understanding of the project. Similarly, it may struggle to generate corner cases for exploring bugs.

• As a large language model, ChatGPT generates/predicts content takes a prompt or starting text as input, and uses its learned understanding of language to predict what words or phrases should come next. This prediction is based on the probability that a certain sequence of words would appear in the dataset. It is highly possible that a commonly used case (i.e., test case/data in our context) holds a higher probability compared to an edge case; and

• By adopting the genetic algorithm to explore potential test suites capable of achieving higher code coverage, Evosuite may theoretically possess a greater probability of discovering bugs. Notably, such a feedback mechanism is presently absent in LLMs, such as ChatGPT, underscoring the potential benefits of combining SBST techniques with LLMs for program testing and bug detection.

It is also worth mentioning that the results presented do not reflect the capability of ChatGPT in finding or locating bugs. It only implicates the bug detection capability of ChatGPT-generated test cases.

Answer to RQ4: Defects and Bug Detection

• The test cases generated by ChatGPT can be misleading in finding logical-related bugs, as the `assertions` generated can be incorrect and unreliable;

• Out of 212 bugs, 44 were successfully detected by test cases generated by ChatGPT, with an average statement code coverage of 50%. In contrast, test cases generated by EvoSuite successfully detected 55 bugs, with an average statement code coverage of 67%;

• Evosuite integrates a genetic algorithm to find test cases that can provide better code coverage and increase the chances of finding bugs. LLM tools like ChatGPT do not have this feedback mechanism. Thus, combining the SBST technique and LLM can improve software testing accuracy and bug detection.

4.5 Correctness of Assertions

RQ5: How do the assertions generated by ChatGPT in unit tests compare in correctness to those produced by SBST?

Methodology. To evaluate the correctness of assertions generated by ChatGPT in unit tests compare in correctness to those produced by SBST, we leverage the IntelliJ IDE's sophisticated debugging tools. With the IDE, we can meticulously monitor the runtime behavior of each test case. This process enables a precise assessment of whether the assertions accurately reflect the intended functionality of the code.

Results.

▷ **Assertions in test cases generated by ChatGPT.** For the corrections of assertions generated by ChatGPT, we only considered the 144 test cases (according to RQ1) that can successfully compiled and executed. For the test cases that cannot be executed, it can be infeasible to evaluate the correctness of assertions.

In general, there are 1,776 methods in these test cases with 2,553 assertions. Among 2,553 assertions, there are 2,001 (78.38%) correct assertions. It suggests that a significant majority of the assertions generated by ChatGPT (78.38%) were fully correct, accurately reflecting the expected behavior of the code. However, a notable fraction of the assertions (21.62%) are not correct, either due to misinterpretation of the code's functionality or limitations in the AI's contextual understanding.

We further investigate the possible reasons that lead to incorrect assertions. Specifically, we find the following main reasons:

• Lack of context (274/548 (50%)): ChatGPT, or any AI model, may not always have the full context of the application or the intricate details of the business logic. This lack of context can lead to assertions that don't accurately capture the intended behavior of the code under test. For instance, the model might not understand the wider implications of a function within the overall application, leading to assertions that are technically correct in isolation but incorrect within the broader context of the application. For the example,

the following code represents an example of this type. The assertion is incorrect due to lack of the context that it requires a case-sensitive representation.

Listing 9: Lack of context

```
// Assume a function that returns true if a user has admin privileges
public boolean isAdmin(User user) {
    return user.hasRole("admin");
}

// Generated test without context might not account for user roles being case-sensitive
public void testIsAdmin() {
    User user = new User();
    user.setRole("Admin"); // Incorrect role due to case sensitivity
    assertFalse(isAdmin(user));
}
```

• Errors with initialization (119/548 (21.7%)): Initialization errors occur when the test setup is incorrect or incomplete. This can lead to assertions that are based on improperly initialized objects or variables, resulting in false positives or negatives. For example, the following code segment presents this type of error.

Listing 10: Errors with initialization

```
public void testCalculateArea() {
    Rectangle rect = null; // Should be initialized properly
    double area = rect.calculateArea();
    assertEquals(20.0, area, 0.01); // Incorrect assertion due to initialization error
}
```

• Misunderstanding of method behavior/lack of implementation details (53/548 (9.7%)) This happens when ChatGPT misunderstands the purpose or the expected behavior of a method. Without a deep understanding of the method's implementation details or intended use, the generated assertions might not accurately test the method's functionality. For example, the following code segment presents this type of error.

Listing 11: Misunderstanding of method behavior/lack of implementation details

```
public void testGetUserFullName() {
    User user = new User("John", "Doe");
    String fullName = user.getFullName();
    assertEquals("Doe, John", fullName);
    // Incorrect assertion, expected "John Doe"
}
```

• Math Errors (26/548 (4.7%)): Math errors in assertions can arise from incorrect calculations or logic related to numerical operations. This might happen if the AI misunderstands the mathematical logic or operations within the code, leading to assertions that expect incorrect values. For example, the following code segment presents this type of error.

Listing 12: Math Error

```
public void testDivide() {
    Calculator calc = new Calculator();
    double result = calc.divide(10, 2);
    assertEquals(6, result); // Incorrect assertion, result should be 5
}
```

• Initialization with NULL (24/548 (4.3%)): Similar to initialization errors, this specific error involves initializing objects or variables with NULL (or null in some languages like Java). This can result in NullPointerExceptions or other unintended behaviors that make the assertions invalid or cause them to fail for the wrong reasons. For example, the following code segment presents this type of error.

Listing 13: Initialization with NULL

```
public void testProcessOrder() {
    Order order = null; // Should not be null
    boolean result = processOrder(order);
    assertTrue(result);
    // Incorrect assertion, will throw NullPointerException
}
```

• Overlooking Common Java Methods (16/548 (2.9%)): This error indicates that ChatGPT might have overlooked or misunderstood common Java methods and their usage. This could lead to assertions that don't utilize standard library methods correctly or fail to account for their behavior, leading to inaccurate tests. For example, the following code segment presents this type of error.

Listing 14: Overlooking Common Java Methods

```
public void testStringConcatenation() {
    String result = "Hello, " + "World!";
    assertEquals("Hello, World!", result.trim());
    // Incorrect usage of trim() method
}
```

• Formatting error/unmatched arguments/unsupported operations(12/548 (2.1%)): These errors can occur due to syntactical mistakes, incorrect formatting, or the use of unsupported arguments in assertions. They can also happen when there's a mismatch between the expected and actual arguments in the assertion method calls. For example, the following code represents an example of this type. The assertion will throw exception due to unmatched arguments.

Listing 15: Formatting error/unmatched arguments/unsupported operations

```
public void testFormatString() {
    String result = String.format("%s is %d years old", "John", "thirty");
    assertEquals("John is thirty years old", result);
}
```

• Method overriding with Null (10/548 (1.8%)): This error involves overriding methods with null values, leading to incorrect behavior in the tests. For instance, if a method that is supposed to return a value is overridden to return null in the test setup, the assertions may fail or pass incorrectly. For example, the following code represents an example of this type.

Listing 16: Method overriding with Null

```
public void testGetUserEmail() {
    User user = new User() {
        public String getEmail() {
            return null; // Overridden to return null
        }
    };
    assertNotNull(user.getEmail()); // Incorrect assertion
}
```

▷ **Assertions in test cases generated by EvoSuite.** Regarding the correctness of assertions in test cases generated by EvoSuite, we also consider the 144 test cases to compare with ChatGPT. Note that, in this RQ, for fairness, we only run EvoSuite once to generate test cases. EvoSuite generate 5,764 test methods in these test cases with 21,818 assertions. Among 21,836 assertions, there are 21,832 (99.9%) correct assertions. It suggests that assertions generated by EvoSuite are highly accurate, demonstrating a success rate of nearly 100%. This level of precision indicates that EvoSuite is a reliable tool for automated test case generation, capable of

producing assertions with a very high probability of being correct.

Answer to RQ5: Correctness of Assertions

In conclusion, when assessing the accuracy of test assertions, EvoSuite shows exceptional performance, underlining its reliability in automated test case generation. In contrast, ChatGPT's test cases exhibit a 78.38% correctness rate in its assertions, reflecting a substantial majority of accurate assertions but also highlighting a gap where 21.62% were incorrect. These inaccuracies may stem from misunderstandings of the code's functionality or limitations in the AI's contextual comprehension. The analysis suggests that while both tools are effective in generating test assertions, EvoSuite demonstrates a higher level of precision compared to ChatGPT.

4.6 Non-determination of ChatGPT

RQ6: How does the non-deterministic output of ChatGPT affect the quality and effectiveness of generated test cases, as measured by code coverage, fault detection?

Methodology. We acknowledge the inherent non-deterministic nature of models like ChatGPT, which can yield different outputs for identical inputs due to factors such as randomness in the sampling methods used to generate text. Therefore, we randomly selected 10 Java classes from the Defects4J and generated unit test cases with ChatGPT 10 times. The generated unit test cases from these repeated trials were then compared using several criteria, including code coverage, and the ability to detect faults.

Results. We randomly select the following 10 subjects (bug.id) from Defects4J to conduct our experiment.

TABLE 19: Randomly Selected Sample for RQ6

Project	BugID
Codec	5,15
Collections	28
Compress	18,35
Csv	2,4
Lang	44
MATH	1
Cli	14

Table. 20 shows the experimental results for 10 trials, in which the symbol '✓' represents the bug is captured by the test case, and the symbol '✗' represents the bug is not captured.

In terms of code coverage and bug detection efficiency, we find that the following findings:

• **Inconsistent Code Coverage Across Trials**: The data shows fluctuating code coverage percentages for the same project and BugId across different trials. For instance, in the Codec project for BugId 5, coverage ranges from 68% to 87%;

• **Varying Bug Detection Efficiency**: The table indicates that the same bug is sometimes detected and sometimes not across different trials (e.g., BugId 15 in Codec detected in Trials 2 and 8 but not in others). This inconsistency in bug detection, despite similar conditions, suggests that ChatGPT's approach to identifying bugs is non-deterministic, leading to different outcomes in different runs;

• **Distinct Outcomes in Trials with Identical Coverage:** Instances where different trials yield identical code coverage but differing results in bug detection underscore the unpredictable nature of the generated test cases. For instance, in the Csv project, BugId 4 showed bug detection in Trials 2, 4, and 5 with varying coverage percentages, including 23% and 54%, while other trials with similar coverage did not detect the bug.

Answer to RQ6: Correctness of Assertions

• The degree of non-determinism in ChatGPT's unit test case generation appears to be relatively high, as evidenced by the variability in code coverage and inconsistency in bug detection across trials for the same BugId.

• This non-determinism is likely influenced by the inherent variability in AI decision-making processes, the complexity of the code and bugs, and the nuances in ChatGPT's understanding of the task context.

4.7 Time Efficiency

A direct comparison of time efficiency between ChatGPT and EvoSuite is not justifiable due to their differing nature, objectives, and operational paradigms. ChatGPT, as a generative language model, is not inherently designed with a primary focus on test efficiency but rather on leveraging natural language understanding and code semantics to generate coherent and contextually relevant test cases. On the other hand, SBST tools like EvoSuite are explicitly engineered to optimize specific criteria such as code coverage or fault detection within a given computational budget, making efficiency a core aspect of their design.

5 Limitations, and Threats to Validity

5.1 Limitations

The results and experiments of this study are limited in two parts: (1) Given the need to manually query ChatGPT, our study is limited to only the queries made for the study. As ChatGPT is closed-source and we cannot map our results to the details or characteristics of ChatGPT's internal model. We also do not know ChatGPT's exact training data, which means we cannot determine if the exact response to our queries are members of the training data; and (2) It also worth noting that ChatGPT's capability is continuously refined and improved by their developers over time according the OpenAI's release note [55]. The responses of ChatGPT can only reflect the performance of ChatGPT at the time we conducted our work (i.e., ChatGPT Jan 30 (2023) Version).

5.2 Threats to Validity

To reduce bias by manually selecting subject programs for testing, we reuse the benchmarks (i.e., Defects4J, DynaMOSA Dataset), which have been used and studied in the existing researches. Furthermore, we also reuse the metrics presented in existing research works to calculate the code coverage, code readability and so forth. Another threat to internal validity comes from the randomness of the genetic algorithms. To reduce the risk, we repeat EvoSuite for 30 times for every class. As for external validity, due to size of the benchmarks, we do not attempt to generalize our results and conclusions.

TABLE 20: Code Statement Coverage and Bug Detection (✓/✗) for 10 Trials

Project	BugId	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10
Codec	5	71%(✗)	78%(✗)	78%(✗)	71%(✗)	68%(✗)	75%(✗)	75%(✗)	87%(✗)	78%(✗)	81%(✗)
Codec	15	63%(✗)	82%(✓)	68%(✗)	72%(✗)	74%(✗)	70%(✗)	82%(✗)	93%(✓)	93%(✗)	97%(✗)
Collections	28	100%(✗)	100%(✗)	100%(✗)	100%(✗)	100%(✗)	100%(✗)	100%(✗)	50%(✗)	100%(✗)	100%(✗)
Compress	18	35%(✗)	0%(✗)	28%(✗)	38%(✗)	19%(✗)	39%(✗)	35%(✗)	35%(✗)	9%(✗)	35%(✗)
Compress	35	7%(✗)	51%(✗)	0%(✗)	18%(✗)	0%(✗)	18%(✗)	17%(✗)	17%(✗)	0%(✗)	25%(✗)
Csv	2	95%(✗)	85%(✗)	90%(✗)	85%(✗)	95%(✗)	80%(✗)	80%(✗)	65%(✗)	85%(✗)	85%(✗)
Csv	4	53%(✓)	23%(✓)	0%(✗)	20%(✓)	54%(✓)	54%(✓)	23%(✗)	17%(✓)	73%(✗)	49%(✗)
Lang	44	39%(✗)	28%(✓)	53%(✗)	43%(✗)	25%(✓)	57%(✗)	2%(✗)	44%(✓)	47%(✗)	22%(✗)
MATH	1	54%(✗)	43%(✗)	43%(✗)	46%(✗)	44%(✗)	53%(✗)	40%(✗)	49%(✗)	37%(✗)	56%(✗)
Cli	14	42% (✗)	35% (✗)	35% (✗)	35% (✗)	33% (✗)	54% (✗)	64% (✗)	53% (✗)	42% (✗)	35% (✗)

6 Related Work

Language Models. Language models are used in NLP for many tasks, such as, machine translation, question answering, summarization, text generation and so on [5], [7], [56], [57], [58], [59], [60], [61], [62], [63]. To better understand language, models with massive parameters are trained on an extremely large corpus (i.e., LLM). Transformer [22] is constructed on stacked encoders and decoders. It leverages self-attention mechanism to weigh the importance of words in the input text, capturing long-range dependencies and relationships between words in the input. It is the base for many LLMs. ELMo [64] utilizes multi-layer bidirectional LSTM and provides high-quality word representations. GPT [28] and BERT [23] are built on the decoders (unidirectional) and encoders (bidirectional) of Transformer, respectively, using pre-training and fine-tuning techniques. GPT-2 [27] and GPT-3 [16] are the descendants of GPT. GPT-2 has a larger model size than GPT, and GPT-3 is larger than GPT-2. Moreover, with larger corpus, GPT-2 and GPT-3 introduce zero-shot and few-shot learning to make models adapt to Multitask. Codex [62] is obtained by training GPT-3 using Github code data. It is the model that powers GitHub Copilot [65], a tool generating computer code automatically. InstructGPT [25] utilizes additional supervised learning and reinforcement learning from human feedback to fine-tune GPT-3, aligning LLM with users. ChatGPT [17] uses the same methods as InstructGPT and has the ability to answer follow-up questions.

Search-based Software Testing. SBST approaches test case generation as an optimization problem. The first SBST method to produce test data for functions with float-type inputs was put out by Miller et al. [66]. Many software testing methods [67], [68], [69] have made extensive use of SBST approaches. Most studies concentrate on (1) Search algorithms: Tonella [18] suggested iterating to generate one test case for each branch. A test suite for all branches was suggested by Fraser et al. [3]. Many-objective optimization techniques were presented by Panichella et al. [19], [20]. To lower the expenses of computing, Grano et al. [70] developed a variation of DynaMOSA; (2) Enhancing fitness gradients: Arcuri et al. introduced testability transformations into API tests [71] For programs with complicated inputs. Lin et al. [72] suggested an approach to deal with the interprocedural flag issue. A test seed synthesis method was suggested by Lin et al. to produce complicated test inputs [29]. Braione et al. [73] coupled symbolic execution and SBST; (3) Design of the fitness function: Xu et al. [74] suggested an adaptive fitness function for enhancing SBST; Rojas et al. [75] suggested combining multiple coverage criteria for fulfilling more requirements from developers. Gregory Gay experimented with various criterion combinations [76] to compare the usefulness of multi-criteria suites for spotting practical flaws. Zhou et al. [4] proposed a method to select coverage goals from multiple criteria instead of combining all goals; (4) Readability of created tests: Daka et al. [77] suggested naming tests by stating covered goals. Deep learning techniques were presented by Roy et al. [78]; (5) Applying SBST to more software fields such as Machine Learning libraries [79], Android applications [80], Web APIs [81], and Deep Neural Networks [82]. Tufano et. al[31] train a model using the transformer architecture to generate unit tests by understanding both the structure of code and the natural language descriptions associated with it. This process entails pre-training on a large corpus of Java code and related documents, followed by fine-tuning on a specifically curated dataset aimed at unit test generation. This approach allows the model to learn the context and intricacies of both programming and natural languages, enabling it to produce more accurate and relevant unit tests. AthenaTest emphasizes leveraging transformers for understanding and generating unit tests from code, focusing on real-world code and tests to create more readable and understandable test cases. Meanwhile, the LLM versus EvoSuite comparison explores the potential of cutting-edge language models in creating effective test cases, contrasting them with EvoSuite's evolutionary algorithm approach.

Leverage LLMs in Software Testing Recent advancements in Large Language Models (LLMs) have opened new avenues in software testing. Specifically, Deng et al proposed TitanFuzz [83], which leverages Large Language Models (LLMs) to generate input programs for fuzzing deep learning libraries. Schäfer et al. [84] proposed TESTPILOT, which is an adaptive LLM-based test generation tool for JavaScript that automatically generates unit tests for the methods in a given project's API. Lemieux et al. leveraged LLMs to help SBST's exploration in taming the Plateaus problem [85]. It addresses the challenges of SBST hitting coverage plateaus by using Codex to provide example test cases for under-covered functions, aiding SBST to explore more effectively. But, CODAMOSA is susceptible to coverage bias. It relies on existing test cases and example inputs, which may not fully explore all program paths. Consequently, it might miss certain edge cases or unanticipated behaviors. CODAMOSA's integration of LLMs enhances test case generation by providing diverse examples, adaptive exploration, and breaking through coverage plateaus. However, it's es-

sential to consider the trade-offs and limitations associated with this approach. In this paper, we focus on evaluating the performance of LLM-generated test cases instead of using LLMs to improve the code coverage. Yuan et al. [86] proposed ChatTESTER to improve the quality of test cases generated by LLMs.

7 Conclusion

In this article, we present a systematic assessment of unit test suites generated by two state-of-the-art techniques: ChatGPT and SBST. We comprehensively evaluate test suites generated by ChatGPT from multiple critical perspectives, including correctness, readability, code coverage, and bug detection capability. Our experimental results demonstrate that (1) 69.6% of the ChatGPT-generated test cases can be successfully compiled and executed; (2) We also observed that the most common violations in the generated code style were `Indentation` (for Google Style) and `MissingJavadocMethod` (for SUN Style), while the majority of the test cases exhibited low complexity; (3) Moreover, our evaluation revealed that EvoSuite outperforms ChatGPT in terms of code coverage by 18.8%; and (4) EvoSuite outperforms ChatGPT in terms of bug detection by 5%. In this paper, our primary objective is to explore the potential benefits and limitations of leveraging Large Language Models (LLMs) like ChatGPT in the generation of unit test suites, particularly in comparison with traditional Search-Based Software Testing (SBST) tools like EvoSuite. We plan to compare the performance of various Large Language Models (LLMs) in generating unit test suites in future work. Such a comparison would not only provide insights into the capabilities and limitations of different LLMs but also guide practitioners and researchers in selecting the most suitable models for their specific needs.

8 Data Availability

The experimental results and raw data are available at: https://sites.google.com/view/chatgpt-sbst

9 Acknowledgment

This work is partially supported by the Hong Kong RGC Project (No. PolyU15224121), HKPolyU Grant (No. ZGGG), and the National Natural Science Foundation of China (No. 62202306).

References

[1] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, p. 366–427, 1997.
[2] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, pp. 1–61, 2012.
[3] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.
[4] Z. Zhou, Y. Zhou, C. Fang, Z. Chen, and Y. Tang, "Selectively combining multiple coverage goals in search-based unit test generation," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.
[5] N. Carlini, F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. B. Brown, D. Song, U. Erlingsson *et al.*, "Extracting training data from large language models." in *USENIX Security Symposium*, vol. 6, 2021.
[6] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean, "Large language models in machine translation," 2007.
[7] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 1, 2022.
[8] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proc. of ESEC/FSE*, 2020, p. 1433–1443.
[9] U. Alon, R. Sadaka, O. Levy, and E. Yahav, "Structural language models for any-code generation," 2019.
[10] G. Poesia, A. Polozov, V. Le, A. Tiwari, G. Soares, C. Meek, and S. Gulwani, "Synchromesh: Reliable code generation from pre-trained language models," in *Proc. of ICLR*, 2022.
[11] P. W. McBurney and C. McMillan, "Automatic source code summarization of context for java methods," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 103–119, 2016.
[12] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proc. of ICSE*, 2010, p. 223–226.
[13] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *Proc. of ICSE*, 2020, p. 1385–1397.
[14] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proc. of ICPC*, 2014, p. 279–290.
[15] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proc. of ICPC*, 2018, p. 200–210.
[16] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 1877–1901.
[17] OpenAI, "Chatgpt: Optimizing language models for dialogue," 2023, https://openai.com/blog/chatgpt/.
[18] P. Tonella, "Evolutionary testing of classes," in *Proc. of ISSTA*, 2004, p. 119–128.
[19] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," in *Proc. of ICST*, 2015, pp. 1–10.
[20] ——, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2018.
[21] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proc. of ESEC/FSE*, 2011, p. 416–419.
[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.
[23] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.
[24] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.
[25] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *arXiv preprint arXiv:2203.02155*, 2022.
[26] M. Artetxe, J. Du, N. Goyal, L. Zettlemoyer, and V. Stoyanov, "On the role of bidirectionality in language model pre-training," *arXiv preprint arXiv:2205.11726*, 2022.
[27] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.
[28] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.
[29] Y. Lin, Y. S. Ong, J. Sun, G. Fraser, and J. S. Dong, "Graph-based seed object synthesis for search-based unit testing," in *Proc. of ESEC/FSE*, 2021, p. 1068–1080.

[30] Defects4J, "Defects4j: A database of real faults and an experimental infrastructure to enable controlled experiments in software engineering research," 2023, https://github.com/rjust/defects4j.

[31] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," *arXiv preprint arXiv:2009.05617*, 2020.

[32] P. Nie, R. Banerjee, J. J. Li, R. J. Mooney, and M. Gligoric, "Learning deep semantics for test completion," *arXiv preprint arXiv:2302.10166*, 2023.

[33] "Writing unit test cases with chatgpt," 2023, https://gist.github.com/kostysh/dbd1dfb2181b96563754222903bf67e7.

[34] "Chatgpt guide: Use these prompt strategies to maximize your results," 2023, https://the-decoder.com/chatgpt-guide-prompt-strategies/.

[35] "Replacing myself: Writing unit tests with chatgpt," 2023, https://bignerdranch.com/blog/replacing-myself-writing-unit-tests-with-chatgpt/.

[36] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1398–1409.

[37] Evosuite, "Evosuite: Automatic test suite generation for java," 2023, https://www.evosuite.org/.

[38] SpotBugs, "Spotbugs," 2023, https://spotbugs.github.io/index.html.

[39] B. Pugh and D. Hovemeyer, "Findbugs," 2023, https://findbugs.sourceforge.net/.

[40] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008.

[41] JetBrain, "Intellij idea – the leading java and kotlin ide," 2023, https://www.jetbrains.com/idea/.

[42] Spotbugs, "Spotbug bug descriptions," 2023, https://spotbugs.readthedocs.io/en/stable/bugDescriptions.html.

[43] CheckStyle, "Checkstyle," 2023, https://checkstyle.sourceforge.io/.

[44] Oracle, "Code conventions for the java programming language," 1999, https://www.oracle.com/java/technologies/javase/codeconventions-contents.html.

[45] Google, "Google java style guide," 2023, https://google.github.io/styleguide/javaguide.html.

[46] C. E. C. Dantas and M. A. Maia, "Readability and understandability scores for snippet assessment: an exploratory study," *arXiv preprint arXiv:2108.09181*, 2021.

[47] P. S. C. Analyzer, "Pmd," 2023, https://pmd.github.io/.

[48] sonarsource, "Cognitive computing: A new way of measuring understandability," 2021, https://www.sonarsource.com/docs/CognitiveComplexity.pdf.

[49] M. G. . C. KG, "Jacoco java code coverage library," 2023, https://www.jacoco.org/jacoco/.

[50] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[51] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, "Combining multiple coverage criteria in search-based unit test generation," in *Search-Based Software Engineering: 7th International Symposium*, 2015, pp. 93–108.

[52] K. Shrestha and M. J. Rutherford, "An empirical evaluation of assertions as oracles," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, 2011, pp. 110–119.

[53] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "An empirical validation of oracle improvement," *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1708–1728, 2021.

[54] V. Terragni, G. Jahangirova, P. Tonella, and M. Pezzè, "Gassert: A fully automated tool to improve assertion oracles," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2021, pp. 85–88.

[55] OpenAI, "Chatgpt release note," 2023, https://help.openai.com/en/articles/6825453-chatgpt-release-notes.

[56] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," *arXiv preprint arXiv:1909.11942*, 2019.

[57] Y. Zhang, S. Sun, M. Galley, Y.-C. Chen, C. Brockett, X. Gao, J. Gao, J. Liu, and B. Dolan, "DIALOGPT : Large-scale generative pre-training for conversational response generation," in *Proc. of ACL*, 2020.

[58] J. Pilault, R. Li, S. Subramanian, and C. Pal, "On extractive and abstractive neural document summarization with transformer language models," in *Proc. of EMNLP*, 2020, pp. 9308–9319.

[59] X. Cai, S. Liu, J. Han, L. Yang, Z. Liu, and T. Liu, "Chestxraybert: A pretrained language model for chest radiology report summarization," *IEEE Transactions on Multimedia*, pp. 845 – 855, 2021.

[60] D. Khashabi, S. Min, T. Khot, A. Sabharwal, O. Tafjord, P. Clark, and H. Hajishirzi, "Unifiedqa: Crossing format boundaries with a single qa system," *arXiv preprint arXiv:2005.00700*, 2020.

[61] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[62] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[63] N. D. Bui, Y. Yu, and L. Jiang, "Infercode: Self-supervised learning of code representations by predicting subtrees," in *Proc. of ICSE*. IEEE, 2021, pp. 1186–1197.

[64] M. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations. arxiv 2018," *arXiv preprint arXiv:1802.05365*, vol. 12, 2018.

[65] G. Copilot, "Your ai pair programmer," 2023, https://github.com/features/copilot/.

[66] W. Miller and D. L. Spooner, "Automatic generation of floating-point test data," *IEEE Transactions on Software Engineering*, no. 3, pp. 223–226, 1976.

[67] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[68] R. A. Silva, S. d. R. S. de Souza, and P. S. L. de Souza, "A systematic review on search based mutation testing," *Information and Software Technology*, vol. 81, pp. 19–35, 2017.

[69] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Time-aware test suite prioritization," in *Proc. of ISSTA*, 2006, pp. 1–12.

[70] G. Grano, C. Laaber, A. Panichella, and S. Panichella, "Testing with fewer resources: An adaptive approach to performance-aware test case generation," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2332–2347, 2019.

[71] A. Arcuri and J. P. Galeotti, "Enhancing search-based testing with testability transformations for existing apis," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 1, pp. 1–34, 2021.

[72] Y. Lin, J. Sun, G. Fraser, Z. Xiu, T. Liu, and J. S. Dong, "Recovering fitness gradients for interprocedural boolean flags in search-based testing," in *Proc. of ISSTA*, 2020, pp. 440–451.

[73] P. Braione, G. Denaro, A. Mattavelli, and M. Pezzè, "Combining symbolic execution and search-based testing for programs with complex heap inputs," in *Proc. of ISSTA*, 2017, pp. 90–101.

[74] X. Xu, Z. Zhu, and L. Jiao, "An adaptive fitness function based on branch hardness for search based testing," in *Proc. of GECCO*, 2017, pp. 1335–1342.

[75] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, "Combining multiple coverage criteria in search-based unit test generation," in *Search-Based Software Engineering*, M. Barros and Y. Labiche, Eds., 2015, pp. 93–108.

[76] G. Gay, "Generating effective test suites by combining coverage criteria," in *Search Based Software Engineering*, 2017, pp. 65–82.

[77] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?" in *Proc. of ISSTA*, 2017, pp. 57–67.

[78] D. Roy, Z. Zhang, M. Ma, V. Arnaoudova, A. Panichella, S. Panichella, D. Gonzalez, and M. Mirakhorli, "Deeptc-enhancer: Improving the readability of automatically generated tests," in *Proc. of ASE*, 2020, pp. 287–298.

[79] S. Wang, N. Shrestha, A. K. Subburaman, J. Wang, M. Wei, and N. Nagappan, "Automatic unit test generation for machine learning libraries: How far are we?" in *Proc. of ICSE*, 2021, pp. 1548–1560.

[80] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *Proc. of ICSE*, 2020, pp. 481–492.

[81] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "Restest: automated black-box testing of restful web apis," in *Proc. of ISSTA*, 2021, pp. 682–685.

[82] F. U. Haq, D. Shin, L. C. Briand, T. Stifter, and J. Wang, "Automatic test suite generation for key-points detection dnns using many-

objective search (experience paper),” in *Proc. of ISSTA*, 2021, pp. 91–102.
[83] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, “Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models,” in *Proceedings of ISSTA*, 2023, pp. 423–435.
[84] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, “Adaptive test generation using a large language model,” *arXiv preprint arXiv:2302.06527*, 2023.
[85] 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 *Proceedings of ICSE*, 2023, pp. 919–931.
[86] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng, “No more manual tests? evaluating and improving chatgpt for unit test generation,” *arXiv preprint arXiv:2305.04207*, 2023.

Yutian Tang is an assistant professor at the School of Computing Science, University of Glasgow. He received the PhD degree from The Hong Kong Polytechnic University. His current research interests include mobile security and privacy, software product lines, empirical software engineering, and testing. He also served as a program committee of several international conferences, including ICSE, FSE, ASE, etc. More information is available at https://www.chrisyttang.org/.

Zhijie Liu is currently a graduate student at the Systems and Security Center, ShanghaiTech University, Shanghai, China. In addition, he is interning at Tencent Keen Security Lab. His present research interests include binary code similarity analysis, malware detection, large language model, program analysis, fuzzing, and so on.

Zhichao Zhou is currently pursuing a Master's degree at the School of Information Science and Technology, ShanghaiTech University, China.

Xiapu Luo is a professor at the Department of Computing and the director of Research Centre for Blockchain Technology of the Hong Kong Polytechnic University. His research focuses on Blockchain and Smart Contracts Security, Mobile and IoT Security, Network Security and Privacy, and Software Engineering with papers published in top-tier security, software engineering, and networking conferences and journals. His research led to ten best/distinguished paper awards, including ACM SIGSOFT Distinguished Paper Awards in ISSTA'22 and ICSE'21, Best Paper Award in INFOCOM'18, Best Research Paper Award in ISSRE'16, etc. and several awards from the industry. His research uncovered many severe vulnerabilities in critical infrastructures and applications, such as blockchain systems and smart contracts, mobile platforms and apps, IoT devices and vehicles. He regularly serves in the program committees of top security and software engineering conferences and received Top Reviewer Award from CCS'22 and Distinguished TPC member Award from INFOCOM'23. He also served as a program committee/general chair of several international conferences, including RAID, SECURECOM, ICICS, etc. He is an associate editor for IEEE/ACM Transactions on Networking (ToN) and IEEE Transactions on Dependable and Secure Computing (TDSC).