Evaluating and Improving ChatGPT for Unit Test Generation

ZHIQIANG YUAN, Fudan University, China
MINGWEI LIU, Fudan University, China
SHIJI DING, Fudan University, China
KAIXIN WANG, Fudan University, China
YIXUAN CHEN, Fudan University, China
XIN PENG, Fudan University, China
YILING LOU, Fudan University, China

Unit testing plays an essential role in detecting bugs in functionally-discrete program units (*e.g.,* methods). Manually writing high-quality unit tests is time-consuming and laborious. Although the traditional techniques are able to generate tests with reasonable coverage, they are shown to exhibit low readability and still cannot be directly adopted by developers in practice. Recent work has shown the large potential of large language models (LLMs) in unit test generation. By being pre-trained on a massive developer-written code corpus, the models are capable of generating more human-like and meaningful test code.

In this work, we perform the first empirical study to evaluate the capability of ChatGPT (*i.e.,* one of the most representative LLMs with outstanding performance in code generation and comprehension) in unit test generation. In particular, we conduct both a quantitative analysis and a user study to systematically investigate the quality of its generated tests in terms of correctness, sufficiency, readability, and usability. We find that the tests generated by ChatGPT still suffer from correctness issues, including diverse compilation errors and execution failures (mostly caused by incorrect assertions); but the passing tests generated by ChatGPT almost resemble manually-written tests by achieving comparable coverage, readability, and even sometimes developers' preference. Our findings indicate that generating unit tests with ChatGPT could be very promising if the correctness of its generated tests could be further improved.

Inspired by our findings above, we further propose CHATTESTER, a novel ChatGPT-based unit test generation approach, which leverages ChatGPT itself to improve the quality of its generated tests. CHATTESTER incorporates an initial test generator and an iterative test refiner. Our evaluation demonstrates the effectiveness of CHATTESTER by generating 34.3% more compilable tests and 18.7% more tests with correct assertions than the default ChatGPT. In addition to ChatGPT, we further investigate the generalization capabilities of CHATTESTER by applying it to two recent open-source LLMs (*i.e.,* CodeLlama-Instruct and CodeFuse) and our results show that CHATTESTER can also improve the quality of tests generated by these LLMs.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**;

Additional Key Words and Phrases: Unit testing, Test generation, Large language model

---

*Corresponding author (yilinglou@fudan.edu.cn)

Authors' addresses: Zhiqiang Yuan, Fudan University, Department of Computer Science, Shanghai, China, zhiqiangyuan23@m.fudan.edu.cn; Mingwei Liu, Fudan University, Department of Computer Science, Shanghai, China, liumingwei@fudan.edu.cn; Shiji Ding, Fudan University, Department of Computer Science, Shanghai, China, sjding22@m.fudan.edu.cn; Kaixin Wang, Fudan University, Department of Computer Science, Shanghai, China, kxwang23@m.fudan.edu.cn; Yixuan Chen, Fudan University, Department of Computer Science, Shanghai, China, yixuanchen23@m.fudan.edu.cn; Xin Peng, Fudan University, Department of Computer Science, Shanghai, China, pengxin@fudan.edu.cn; Yiling Lou, Fudan University, Department of Computer Science, Shanghai, China , yilinglou@fudan.edu.cn.

---

1 INTRODUCTION

Unit testing [56, 67, 89] validates whether a functionally-discrete program unit (*e.g.,* a method) under test behaves correctly. As the primary stage in the software development procedure, unit testing plays an essential role in detecting and diagnosing bugs in a nascent stage and prevents their further propagation in the development cycle. Therefore, writing high-quality unit tests is crucial for ensuring software quality. For a method under test (*i.e.,* often called as the focal method), its corresponding unit test consists of a *test prefix* and a *test oracle* [8]. In particular, the test prefix is typically a series of method invocation statements or assignment statements, which aim at driving the focal method to a testable state; and then the test oracle serves as the specification to check whether the current behavior of the focal method satisfies the expected one.

Manually writing and maintaining high-quality unit tests can be very time-consuming and laborious [19, 44]. To alleviate manual efforts in writing unit tests, researchers have proposed various techniques to facilitate automated test generation [9, 45, 49, 50, 68]. Traditional unit test generation techniques leverage search-based [10, 20, 37], constraint-based [16, 28, 83], or random-based strategies [59, 87] to generate a suite of unit tests with the main goal of maximizing the coverage in the software under test. Although achieving reasonable coverage, these automatically-generated tests exhibit a large gap to manual-written ones in terms of readability and meaningfulness, and thus developers are mostly unwilling to directly adopt them in practice [6].

To address these issues, recent work [24, 51, 54, 73, 79] has leveraged advanced deep learning (DL) techniques, especially large language models (LLMs), to generate unit tests. These techniques mostly formulate unit test generation as a neural machine translation problem by translating a given focal method into the corresponding test prefix and the test assertion. In particular, they incorporate the power of LLMs by fine-tuning these pre-trained models on the test generation task. Owing to being extensively pre-trained on a massive developer-written code corpus and then being specifically fine-tuned on the test generation task, these models are capable of generating more human-like and meaningful test code, showing a large potential of LLMs in unit test generation.

Study. In this work, we perform the first empirical study to evaluate the capability of ChatGPT [57] (*i.e.,* one of the most representative LLMs) in unit test generation. Different from the relatively-smaller pre-trained models (*e.g.,* BART [14], BERT [23], and T5 [64]) used in existing learning-based test generation techniques [17, 23, 63–65, 84], ChatGPT incorporates instruction tuning and RLHF [7] (Reinforcement Learning from Human Feedback) on a significantly-larger model, which exhibits better generalization and higher alignment with human intention in various domains. In particular, ChatGPT has demonstrated outstanding capability of solving various tasks in code generation and comprehension [26, 27, 62, 76]. To enable a comprehensive evaluation, we first construct a dataset of 1,000 Java focal methods, each along with a complete and executable project environment. We incorporate ChatGPT to generate unit tests for each focal method and analyze the quality of the generated tests to answer the following four research questions.

- **RQ1 (Correctness): How is the correctness of the unit tests generated by ChatGPT?** We first measure the syntactic correctness, compilation correctness, and execution correctness of the generated tests; and then further build a breakdown of the error types in the incorrect tests.
- **RQ2 (Sufficiency): How is the sufficiency of the unit tests generated by ChatGPT?** We investigate the coverage and assertions of the tests generated by ChatGPT.

- **RQ3 (Readability): How is the readability of the unit tests generated by ChatGPT?** For those correct tests generated by ChatGPT, we perform a user study to assess their readability along with the manually-written tests as reference.
- **RQ4 (Usability): How can the tests generated by ChatGPT be used by developers?** For those correct tests generated by ChatGPT, we perform a user study to investigate whether developers are willing to adopt them.

Based on our results, we have the following main findings. On the bad side, we find that only a portion (24.8%) of tests generated by ChatGPT can pass the execution and the remaining tests suffer from diverse correctness issues. In particular, 57.9% of ChatGPT-generated tests encounter compilation errors, such as using undefined symbols, violating type constraints, or accessing private fields; and 17.3% of the tests are compilable but fail during execution, which mostly result from the incorrect assertions generated by ChatGPT. On the good side, we find the passing tests generated by ChatGPT actually resemble manually-written ones by achieving comparable coverage, readability, and sometimes even developers' preference compared to manually-written ones. Overall, our results indicate that ChatGPT-based test generation could be very promising if the correctness issues in its generated tests could be further addressed. To this end, we further distill two potential guidelines, *i.e.,* providing ChatGPT with deep knowledge about the code and helping ChatGPT better understand the intention of the focal method, so as to reduce the compilation errors and assertion errors in its generated tests, respectively.

Technique. Inspired by our findings above, we further propose CHATTESTER, a novel ChatGPT-based unit test generation approach, which leverages ChatGPT itself to improve the correctness of its generated tests. CHATTESTER includes an initial test generator and an iterative test refiner. The initial test generator decomposes the test generation task into two sub-tasks by (i) first leveraging ChatGPT to understand the focal method via the *intention prompt* and (ii) then leveraging ChatGPT to generate a test for the focal method along with the generated intention via the *generation prompt*. The iterative test refiner then iteratively fixes the compilation errors in the tests generated by the initial test generator, which follows a validate-and-fix paradigm to prompt ChatGPT based on the compilation error messages and additional code context.

To evaluate the effectiveness of CHATTESTER, we further apply CHATTESTER on an evaluation dataset of 100 additional focal methods (to avoid using the same dataset that has been extensively analyzed in our study part) and three projects (to analyze the project-level effectiveness), and compare the tests generated by CHATTESTER with the default ChatGPT. Furthermore, since the idea of ChatGPT is general and not limited to the specific model (*e.g.,* ChatGPT), we further investigate the generalization capabilities of CHATTESTER by applying it to two recent open-source LLMs (*i.e.,* CodeLlama-Instruct and CodeFuse). We answer the following research questions in experiments.

- **RQ5 (Improvement): How effective is CHATTESTER in generating correct tests compared to ChatGPT? How effective is each component in CHATTESTER?** We compare the number of compilation errors and execution failures between the tests generated by CHATTESTER and the default ChatGPT. Moreover, we investigate the contribution of each component in CHATTESTER.
- **RQ6 (Generalization): How effective is CHATTESTER in improving the quality of generated tests when applied to other LLMs?** We replace ChatGPT with other LLMs and study the generalization capability of CHATTESTER on helping other LLMs generate high-quality tests.
- **RQ7 (Project-level Effectiveness): How effective is CHATTESTER when applied to the entire projects?** We evaluate the coverage, readability, and usability of CHATTESTER-generated tests when applied to entire projects.

Our results show that CHATTESTER substantially improves the correctness of the ChatGPT-generated tests with 34.3% and 18.7% improvement in terms of the compilable rate and execution

passing rate. In addition, our results further confirm the contribution of both components in CHATTESTER, *i.e.,* the initial test generator is capable to generate more tests with correct assertions while the iterative test refiner is capable to fix the compilation errors iteratively. Furthermore, our results show that CHATTESTER can be generalized to the two studied open-source LLMs (*i.e.,* CodeLlama-Instruct and CodeFuse) by helping both of them generate more correct tests.

In summary, this paper makes the following contributions:

- **The first study** that extensively investigates the correctness, sufficiency, readability, and usability of ChatGPT-generated tests via both quantitative analysis and user study;
- **Findings and practical implications** that point out the limitations and prospects of ChatGPT-based unit test generation;
- **The first technique CHATTESTER** includes a novel initial test generator and iterative test refiner, which leverages ChatGPT itself to improve the correctness of its generated tests;
- **An extensive evaluation** that demonstrates the effectiveness and generalization capability of CHATTESTER by substantially reducing the compilation errors and incorrect assertions in tests generated by ChatGPT and two open-source LLMs.

The data and code can be found on our website [12].

2 BACKGROUND

2.1 Large Language Models

Large language models (LLMs) are large-scale models pre-trained on a massive textual corpus [52, 58, 64, 74]. In order to fully utilize the massive unlabeled training data, LLMs are often pre-trained with self-supervised pre-training objectives [5, 11, 33], such as Masked Language Modeling [29], Masked Span Prediction [78], and Causal Language Modeling [55]. Most LLMs are designed on a Transformer [74], which contains an encoder for input representation and a decoder for output generation. Existing LLMs can be grouped into three categories, including encoder-only (*e.g.,* CodeBERT [29]), decoder-only (*e.g.,* CodeGen [55]), and encoder-decoder models (*e.g.,* CodeT5 [78]).

To enhance the generalization ability and the human-intention-alignment of LLMs on unseen downstream tasks, more recent work leverages instruction tuning and reinforcement learning to further improve the model performance [57, 58]. For example, ChatGPT [57], one of the most representative LLMs developed by OpenAI based on the generative pre-trained transformer (GPT) architecture, first tunes the GPT model with instruction tuning and then updates the model with reinforcement learning from human feedback. In addition to commercial LLMs, there are an emerging number of open-source instructed LLMs (*e.g.,* CodeLlama-Instruct [4] and CodeFuse [15]) that also show promising performance in various tasks.

2.2 Unit Test Generation

For a method under test (*i.e.,* often called as the focal method), unit test generation techniques automatically generate its corresponding unit test, which often consists of a *test prefix* and a *test oracle* [8]. The test prefix is sequential method invocation statements or assignment statements to drive the focal method to a testable state, and the test oracle (*e.g.,* assertions) checks whether the focal method behaves consistently with the specification.

Traditional techniques use search-based [31], random-based [59], or constraint-based strategies [16, 28] to generate unit tests automatically. For example, Evosuite [31], one of the most representative search-based techniques, uses evolutionary algorithms to generate test suites for given Java classes with the goal of maximizing coverage. Although achieving reasonable coverage, the tests generated by traditional techniques have low readability and meaningfulness

compared to manually-written ones, which cannot be directly adopted by developers in practice [6, 18, 35, 36, 60, 61].

Recent work [24, 51, 54, 73, 79] leverages advanced deep learning techniques, especially LLMs, to generate unit tests. Some learning-based techniques regard test generation as a neural machine translation problem, which fine-tunes the model to translate the focal method into the corresponding test prefix or assertion. More recently, researchers propose different prompt strategies [43, 46, 48, 53] to leverage instructed LLMs to generate test inputs and assertions.

3 STUDY SETUP

3.1 Benchmark

Existing benchmarks on unit test generation [73, 79] only include a limited code context (*e.g.,* the focal method alone) rather than a complete and executable project, and it is hard to directly compile and execute the generated tests with the existing datasets. Therefore, to comprehensively evaluate the quality of ChatGPT-generated tests, we construct a new benchmark of not only focal methods but also complete and executable projects. We construct our benchmark as follows.
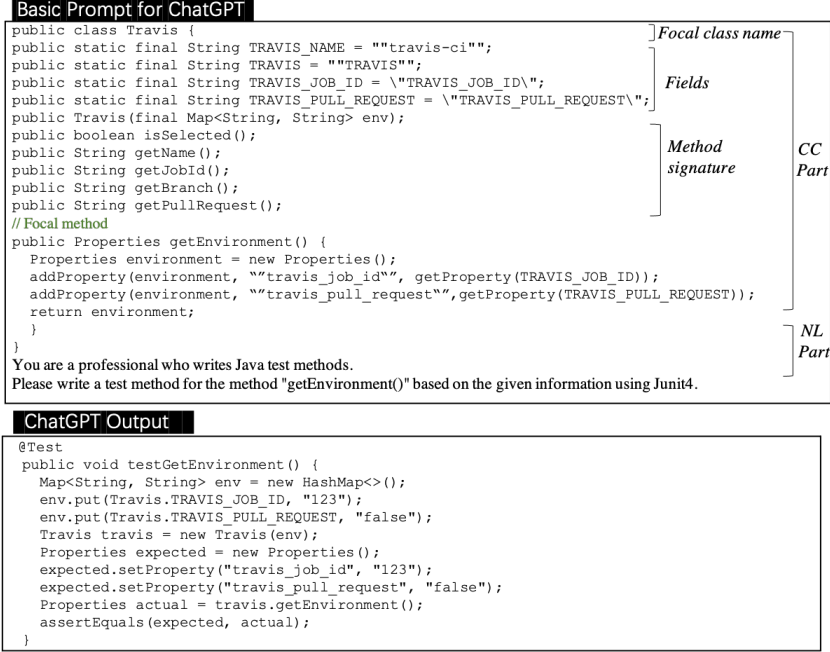**Project Collection.** We use the 4,685 Java projects in the popular benchmark CodeSearchNet [39] as the initial project list. For each project, we clone it from GitHub (as of March 25, 2023) and collect its relevant information (*e.g.,* its creating time and its last commit time). To keep high-quality projects, we filter the 4,685 Java projects according to the following criteria: (i) the project is under continuous maintenance (*i.e.,* the project should have been updated as of January 1, 2023); (ii) the project has at least 100 stars; (iii) the project is built with Maven framework (for the ease of test executions) and it could be successfully compiled in our local environment. In this way, we obtain 185 Java projects, containing 502/118 code/test files and 244,876/20,934 lines of code/test code.
Data Pair Collection. We then extract data pairs from the 185 Java projects. Each data pair refers to the pair of the focal method information and its corresponding test method. In particular, in addition to the focal method itself, the focal method information also includes the focal class declaration, all the fields, and all the method signatures (*i.e.,* the class constructor and the instance methods). For each Java project, we extract data pairs in the following steps. (i) Given a Java project, we first find all the test classes in the project. If a class contains at least one method annotated with @Test, we regard this class as a test class and collect all the test methods in this test class. (ii) We then find the corresponding focal method for each test method based on the file path and the class name matching. For example, for a test method *"testFunction()"* located in the path *"src/test/java/FooTest.java"*, we consider the method *"Function()"* located in the path *"src/main/java/Foo.java"* as its focal method. For the cases when there are multiple focal methods with the same name in the same class, we further filter them by the number and types of parameters to find the unique matching one. For fair comparison, we currently keep the test methods comprising one test case.

With such strict mapping criteria, we extract 1,748 data pairs from 185 Java projects. Considering the costs of using ChatGPT API and the manual efforts, we further sample 1,000 data pairs as our final benchmark for the empirical study, which includes test methods and focal methods of diverse scales and structures. The detailed statistics of our benchmark can be found at our website [12].

3.2 Basic Prompt Design

To avoid using too simple prompts that might lead to underestimation of ChatGPT's capability or using too sophisticated prompts that are uncommon in practice, we design our basic prompt by carefully following the common practice in existing unit test generation work [54, 73, 79] and widely-adopted experience on using ChatGPT [2, 25]. In particular, our basic prompt includes two part: (i) the natural language description part (*i.e.,* NL part) that explains the task to ChatGPT, and

Basic Prompt for ChatGPT

```
public class Travis {                                                     Focal class name
public static final String TRAVIS_NAME = ""travis-ci"";
public static final String TRAVIS = ""TRAVIS"";
public static final String TRAVIS_JOB_ID = \"TRAVIS_JOB_ID\";            Fields
public static final String TRAVIS_PULL_REQUEST = \"TRAVIS_PULL_REQUEST\";
public Travis(final Map<String, String> env);
public boolean isSelected();                                             Method        CC
public String getName();                                                 signature     Part
public String getJobId();
public String getBranch();
public String getPullRequest();
// Focal method
public Properties getEnvironment() {
  Properties environment = new Properties();
  addProperty(environment, ""travis_job_id"", getProperty(TRAVIS_JOB_ID));
  addProperty(environment, ""travis_pull_request"",getProperty(TRAVIS_PULL_REQUEST));
  return environment;
  }                                                                                    NL
}                                                                                      Part
You are a professional who writes Java test methods.
Please write a test method for the method "getEnvironment()" based on the given information using Junit4.
```

ChatGPT Output

```
@Test
 public void testGetEnvironment() {
   Map<String, String> env = new HashMap<>();
   env.put(Travis.TRAVIS_JOB_ID, "123");
   env.put(Travis.TRAVIS_PULL_REQUEST, "false");
   Travis travis = new Travis(env);
   Properties expected = new Properties();
   expected.setProperty("travis_job_id", "123");
   expected.setProperty("travis_pull_request", "false");
   Properties actual = travis.getEnvironment();
   assertEquals(expected, actual);
 }
```

Fig. 1. Basic Prompt for *getEnvironment* [34] Method

(ii) the code context part (*i.e.,* CC part) that contains the focal method and the other relevant code context. We then explain each part in detail.

CC Part. Following existing learning-based unit test generation work [73], we include the following code context into the CC part: (i) the complete focal method, including the signature and body; (ii) the name of the focal class (*i.e.,* the class that the focal method belongs to); (iii) the field in the focal class; and (iv) the signatures of all methods defined in the focal class.

NL Part. Based on the widely-acknowledged experience on using ChatGPT, we include the following contents in the NL part: (i) a role-playing instruction (*i.e.,* "You are a professional who writes Java test methods.") to inspire ChatGPT's capability of test generation, which is a common prompt optimization strategy [2, 25]; and (ii) a task-description instruction (*i.e.,* "Please write a test method for the {focal method name} based on the given information using {Junit version}").

The top half of Figure 1 shows an example of our basic prompt. After querying with the basic prompt, ChatGPT then returns a test as shown in the bottom half of Figure 1.
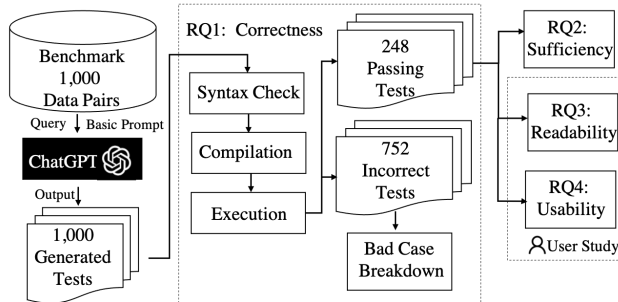
Fig. 2. The Workflow of Our Empirical Study

3.3 Baselines

We further include two state-of-the-art traditional and learning-based unit test generation techniques as baselines. We focus on techniques applicable to generating JUnit tests due to our constructed benchmark. For traditional techniques, we use Evosuite [31] as the baseline with its default setting (*e.g.,* "assertion_strategy" = MUTATION and "assertion_timeout = 60s"). For learning-based techniques, we include AthenaTest [73] as the baseline. We do not consider the recent learning-based test completion technique Teco [54] since it mainly targets at statement-level test completion while in this work we focus on directly generating a complete test method. We do not include the two Codex-based test generation techniques (*e.g.,* CODAMOSA [46] and LIBRO [43]), since they focus on different test generation scenarios as ours (More detailed explanation can be found in Section 8.2). For AthenaTest, since it has not released its pre-trained BART-based model or its fine-tuned model, we try our best to reproduce it according to its paper. Since AthenaTest has not released its pretraining dataset neither, it is infeasible (also very resource-consuming) to re-pretrain an LLM. Therefore, we reproduce AthenaTest by fine-tuning the widely-used LLM CodeT5 on the same fine-tuning dataset used in AthenaTest. We choose CodeT5 since it has been pre-trained on both textual and code corpus, which is also the best pre-training setting shown in the AthenaTest paper [73]. In addition, to avoid potential data leakage, we remove the overlap data from the fine-tuning dataset if it is duplicated in our benchmark based on character matching.

3.4 Experimental Procedure

Figure 2 shows the overview of our experimental procedure. For each data pair in the benchmark constructed in Section 3.1, we query ChatGPT with our basic prompt designed in Section 3.2 and take the test generated by ChatGPT as the output. To automate our experiments, we use the official ChatGPT API [57] with the default setting. In this work, we focus on the gpt-3.5-turbo model rather than GPT-4, due to the limited rate of GPT-4 API for non-industrial users. We then put the generated test in the same directory of its focal class, and attempt to compile and execute it for further analysis. We then explain the detailed procedure in each RQ, respectively.

RQ1: Correctness. Following existing learning-based test generation work [54, 73], we measure the correctness of the generated tests with three metrics, including (i) syntactic correctness (whether the test could pass the syntax checker), (ii) compilation correctness (whether the test could be successfully compiled), and (iii) execution correctness (whether the test could pass the execution). Here we leverage AST parser (such as JavaParser [1]) as a syntax checker. In addition, we further investigate the common error types in those incorrect tests to better understand the limitations of ChatGPT in test generation. Specifically, we automatically extract the error messages thrown during the compilation and execution, including different compilation and execution error types.

RQ2: Sufficiency. In this RQ, we include three metrics to assess the sufficiency of the tests generated by ChatGPT: (i) the statement coverage of the test on the focal method; (ii) the branch coverage of the test on the focal method; (iii) the number of assertions in the test case. In particular, we leverage Jacoco [3] to collect the coverage.

RQ3 & RQ4: User study for readability and usability. In these two RQs, we conduct a user study to investigate the readability and usability of the tests generated by ChatGPT. Here, we only focus on 248 passing tests generated by ChatGPT since it is less meaningful to recommend tests with compilation errors or execution errors to developers in practice. We invite five participants whose Java development experiences vary from 4 years to 5 years. For each ChatGPT-generated test and its corresponding manually-written test, we ask each participant to score their readability and usability via the detailed criteria in Table 1. In particular, the score range of both readability and usability is from 2 to 6, which are the sum of its sub-properties, *i.e.,* readability consists of the

naming intuitiveness and the code layout while the usability consists of the assertion quality and the adoption efforts. Given the large manual efforts required by such an assessment procedure, we set a reward mechanism (*i.e.,* 250$ per participant) and sufficient time (*i.e.,* two weeks) to mitigate arbitrary responses. In addition, participants are not informed which test is generated by ChatGPT or which is written manually.

Table 1. Scoring Criteria for Readability and Usability

		Scoring criteria
Readability (2 ~ 6)	**Naming Intuitiveness (1 ~ 3):** the clarity and descriptiveness of variable and test method names	3: Most names are intuitive and easy to understand.
		2: Some names are intuitively named and easy to understand.
		1: Most names are hard to understand without extensive context.
	Code Layout (1 ~ 3): the code structure, logic, and formatting.	3: Well-structured code with clear logic.
		2: Overall reasonable structure with minor issues.
		1: Chaotic logic or massive redundant code
Usability (2 ~ 6)	**Assertion Quality (1 ~ 3):** validating the focal method with appropriate assertions.	3: Accurate assertions withnin the test context.
		2: Reasonable but weak assertions.
		1: semantically-incorrect assertions.
	Adoption Efforts (1 ~ 3): how easily the generated tests can be adopted into the practical usage.	3: No efforts required before direct adoption.
		2: Simple modifications required before direct adoption.
		1: Extensive efforts required to optimize.

Table 2. Correctness of Generated Tests

Metrics (%)	ChatGPT	AthenaTest	Evosuite
Syntactical correct	≈ 100.0	54.8	100.0
Success compilation	42.1	18.8	97.6
Passing execution	24.8	14.4	91.4

4 STUDY RESULTS

4.1 RQ1: Correctness

Table 2 presents the correctness of the 1,000 tests generated by ChatGPT and other techniques. Overall, we could observe that a large portion of tests generated by ChatGPT suffer from correctness issues, *i.e.,* 42.1% of the generated tests are successfully compiled while only 24.8% of the generated tests are executed successfully without any execution errors. We further manually inspect the failed tests to check whether they actually reveal bugs in the focal method under test, but we find that all of them are caused by the improper test code itself.

As for the learning-based baseline AthenaTest, ChatGPT has a substantial improvement over AthenaTest in terms of syntactic correctness, compilation correctness, and executable correctness. For example, almost all the tests generated by ChatGPT (except the one has an incorrect parenthesis generated) are syntactically correct, but nearly a half of the tests generated by AthenaTest are syntactically incorrect. The reason might be that the significantly-larger model scale in ChatGPT helps better capture syntactical rules in the massive pre-training code corpus. As for the traditional search-based baseline Evosuite, we could observe a higher compilable rate and passing rate in its generated tests. In fact, it is as expected since Evosuite prunes invalid test code during its search procedure and generates assertions exactly based on the dynamic execution values, while learning-based techniques (*i.e.,* ChatGPT and AthenaTest) directly generate tests token by token without any post-generation validation or filtering. Therefore, we do not intend to conclude that Evosuite is better at generating more correct tests, since the correctness of tests generated by learning-based techniques could be further improved if they also incorporate similar post-generation validation to filter those incorrect tests.

Finding 1: ChatGPT substantially outperforms existing learning-based techniques in terms of syntactic, compilation, and execution correctness. However, only a portion of its generated tests can pass the execution while a large ratio of its generated tests still suffer from compilation errors and execution errors.

Bad Case Breakdown. We further analyze the common error types in the failed tests generated by ChatGPT (*i.e.,* those tests failed on the compilation or execution). In particular, we first automatically categorize each test based on the error message; and then we manually summarize and merge similar types into high-level categories. Table 3 shows the breakdown for tests with compilation errors, while Table 4 shows the breakdown for tests with execution failures.

Table 3. Compilation Error Breakdown

Category	Detailed Errors	Frequency
Symbol Resolution Error	Cannot find symbol class	1,934
	Cannot find symbol method	471
	Cannot find symbol variable	466
	Cannot find symbol	169
Type Error	Incompatible types	73
	Constructor cannot be applied to given types	46
	Methods cannot be applied to given types	11
Access Error	Private access	75
Abstract Class Initiation Error	Abstract class cannot be instantiated	33
Unsupported Operator	Diamond operator is not supported	15

Table 4. Execution Error Breakdown

Category	Detailed Errors	Frequency
Assertion Error	java.lang.AssertionError/ org.opentest4j.AssertionFailedError/ org.junit.ComparisonFailure/ org.junit.internal.ArrayComparisonFailure	148
Runtime Error	java.lang.IllegalArgumentException	6
	java.lang.RuntimeException	5
	java.lang.NullPointerException	3
	others	11

Failed Compilation. In Table 3, the column "Frequency" shows the number of each compilation errors in 579 uncompilable tests. Note that there could be multiple compilation errors in one test and thus the sum is larger than 579. Due to space limits, we only present the frequent compilation errors observed more than 10 times. As shown in the table, the generated tests have diverse compilation errors. First, the most frequent compilation errors are caused by un-resolvable symbols, *e.g.,* the generated tests include some undefined classes, methods, or variables. Second, another large category of compilation errors are related to type errors, *e.g.,* the parameter type in the method invocation is inconsistent with the one defined in the method declaration. Third, ChatGPT also frequently generates test code that invalidly accesses private variables or methods (*i.e.,* access errors). In addition, some generated tests encounter compilation errors by invalidly instating abstract class or using unsupported operators.

Failed Execution. In Table 4, we group all the infrequent errors (*i.e.,* less than three times) into the "others" category due to space limits. As shown in the table, the majority of failed executions (85.5%) are caused by assertion errors, *i.e.,* the assertions generated by ChatGPT consider the behavior of the program under test violates the specification. As mentioned above, we manually inspect these assertion errors to identify whether they are caused by the bugs in the focal method or the

incorrect assertions themselves, and we find all of them as a result of incorrect assertions generated by ChatGPT. It implies that ChatGPT might fail to precisely understand the focal method and the quality of the assertions generated by ChatGPT should be largely improved. In addition, we observe that the remaining execution errors are related to different runtime exceptions. For example, Figure 3 presents an example of the failed execution in the test generated by ChatGPT. The test throws *NullpointerException* when executing line 3. The error occurs because the created object *"url"* assesses an external resource *"/test.jar"* which does not exist (in line 2). It actually shows an inherent limitation in ChatGPT, *i.e.,* the unawareness of external resources during test generation.

```
1   public void testGetManifestFromJarURLConnection() throws IOException {
2     URL url = getClass().getResource("/test.jar");
      //java.lang.NullpointerExeception
3     JarURLConnection connection = (JarURLConnection)url.openConnection();
      ...}
```

Fig. 3. NullPointerException Example

Finding 2: The ChatGPT-generated tests encounter diverse compilation errors, such as symbol resolution errors, type errors, and access errors; the majority of failed executions are caused by the incorrectly-generated assertions.

4.2 RQ2: Sufficiency

Table 5 presents the statement and branch coverage of generated tests that could pass the execution. As the un-executable tests exhibit zero coverage which can bias the overall results, the coverage is uniformly calculated on the focal methods for which both ChatGPT and Evosuite can generate executable tests. We further include the coverage of the manually-written tests (*i.e.,* the original test for the focal method in the project) for reference. Since Evosuite might generate more than one tests for a focal method, we present the first, worse, average, and union coverage of its generated tests. As shown in the table, we could observe that the tests generated ChatGPT achieve the highest coverage compared to existing learning-based and search-based techniques and it also achieve comparable coverage as manually-written tests.

Table 5. Coverage of Generated Tests

Coverage (%)	ChatGPT	AthenaTest	Evosuite					Manual
			First	Worst	Best	Average	Union	
Statement	82.3	65.5	68.0	39.7	76.2	56.9	81.1	84.2
Branch	65.6	56.2	61.2	31.9	62.1	45.8	77.3	68.9

Figure 4 presents a distribution plot for the number of assertions in each test generated by different techniques. Interestingly, we observe that ChatGPT-generated tests exhibit most similar distribution as manually-written tests in the number of assertions per test. In particular, Evosuite tends to generate tests with less assertions while the learning-based technique AthenaTest would generate some tests with abnormally-larger number of assertions (*i.e.,* more than 15 assertions per test) than manually-written ones. The potential reason might be that RLHF helps ChatGPT generate more human-like test code.

Finding 3: The ChatGPT-generated tests resemble manually-written ones in terms of test sufficiency. ChatGPT achieves comparable coverage as manual tests, and also the highest coverage compared to existing techniques; ChatGPT also generate more human-like tests with similar number of assertions per test as manually-written tests.
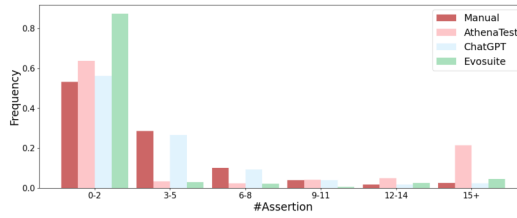
Fig. 4. Number of Assertions in Generated Tests

4.3 RQ3: Readability

Figure 5 shows the score distribution of readability in a stacked bar chart, where the x-axis represents each participant (*i.e.,* from A to E) and the y-axis represents the ratio of different scores. Overall, most ChatGPT-generated tests are assessed with decent readability, and they are also considered with comparable and sometimes even better readability compared to manually-written tests.

Finding 4: The tests generated by ChatGPT have reasonable and comparable readability as manually-written ones.
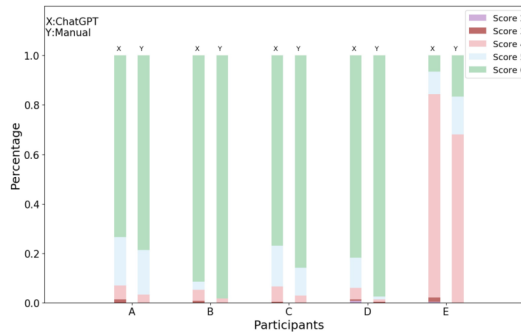
Fig. 5. Response to Readability

4.4 RQ4: Usability

Figure 6 shows the score distribution of usability in a stacked bar chart, where the x-axis represents each participant (*i.e.,* from A to E) and the y-axis represents the ratio of different scores. Interestingly, we find the ChatGPT -generated tests exhibit comparable usability compared to manually-written ones. Based on the scoring details of the two sub-properties in usability (*i.e.,* assertion quality and adoption efforts, we find that most ChatGPT-generated tests are assessed with high-quality assertions for immediate usage, which makes the participants' high willing of direct usage.

Finding 5: ChatGPT-generated tests show a large potential in practical usability. In a considerable portion of cases, participants are willing to directly adopt ChatGPT-generated tests.

4.5 Enlightenment

Based on our results above, we further discuss the implications on the strengths and the limitations of ChatGPT-based unit test generation.

Limitations. As shown in our results, a large portion of ChatGPT-generated tests fail in compilation or execution, which might result from two inherent limitations in generative language models.

First, most compilation errors might be caused by ChatGPT's unawareness of the "deep knowledge" in the code. Although being pre-trained on a massive code corpus could help ChatGPT capture the syntactical rules in the code, the nature of ChatGPT is still a probabilistic token-sequence
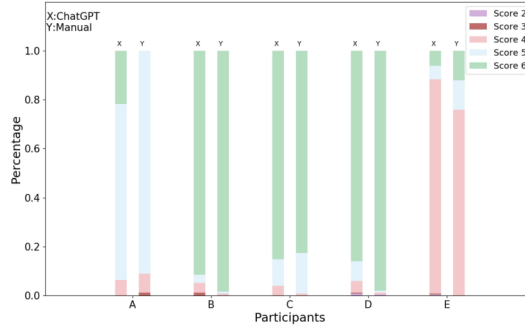
Fig. 6. Response to Usability

generator and thus it is challenging for ChatGPT to be fully aware of the deep rules in the code, *e.g.,* only the public fields could be accessed outside the class and the abstract classes cannot be instantiated. *Therefore, to help ChatGPT overcome this limitation, it is important to remind ChatGPT of such deep knowledge during its generating tests.*

Second, most execution errors (*i.e.,* assertion errors) result from ChatGPT's lack of understanding about the intention of the focal method. As a result, it is challenging for ChatGPT to write proper assertions as specification for the focal method under test. *Therefore, to help ChatGPT overcome this limitation, it is essential to help ChatGPT to better understand the intention of the focal method.*

Strengths. Although generating a lot of tests failed in compilation or execution, the good thing is that most of the passing tests generated by ChatGPT are often of high quality in terms of the sufficiency, the readability, and the usability in practice. These passing tests could mostly be put into direct use to alleviate manual test-writing efforts. *Therefore, leveraging ChatGPT to generate unit tests is a promising direction if the correctness issues in its generated tests could be further addressed.*

> **Enlightenment:** ChatGPT-based unit test generation is promising since it is able to generate a number of high-quality tests with comparable sufficiency, readability, and usability as manually-written tests. However, further efforts are required to address the correctness issues in the ChatGPT-generated tests. The two directions to this end are (i) to provide ChatGPT with deep knowledge about the code and (ii) to help ChatGPT better understand the intention of the focal method, so as to reduce its compilation errors and assertion errors, respectively.

5   APPROACH OF CHATTESTER

Overview. Inspired by our findings and enlightenment above, we then propose CHATTESTER, a novel ChatGPT-based unit test generation approach, which improves the correctness of ChatGPT-generated tests by ChatGPT itself. In particular, CHATTESTER contains two components, *i.e.,* an initial test generator and an iterative test refiner. Figure 7 shows the workflow of CHATTESTER.

Instead of directly asking ChatGPT to generate a test for the given focal method, the initial test generator decomposes the test generation task into two sub-tasks: (i) first understanding the intention of the focal method, and (ii) then generating a unit test for the focal method based on the intention. Compared to the basic prompt, the initial test generator aims to generate tests with higher-quality assertions based on the help of the intermediate step of intention generation.

The iterative test refiner iteratively fixes the compilation errors in the tests generated by the initial test generation. As mentioned in our enlightenment, the key to eliminating most uncompilable tests is to provide "deep knowledge" to ChatGPT during test generation. However, given the large number of such potential rules, it is infeasible to include all of them in the prompt in advance to the test generation. Therefore, we follow a validate-and-fix paradigm to iteratively refine the
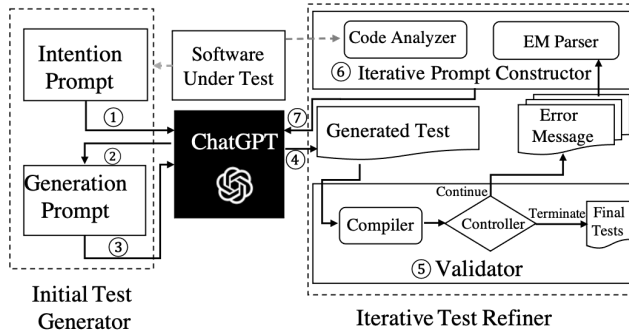
Fig. 7. The Workflow of ChatTester

uncompilable test by prompting ChatGPT with compilation error messages and additional relevant code context. In other words, the iterative test refiner actually leverages the error messages from the compiler as the violation instances of the "deep knowledge", so as to fix compilation errors in the generated tests.

(a) Basic prompt

(b) Prompts in the initial test generator

Fig. 8. Basic Prompt v.s. Prompts in the Initial Test Generator

5.1 Initial Test Generator

The initial test generator decomposes test generation into two steps: (i) first leveraging ChatGPT to understand the focal method via the *intention prompt*, and (ii) then leveraging ChatGPT to generate a test for the focal method along with the generated intention via the *generation prompt*.

The intention prompt asks ChatGPT to return a natural language description of the intended functionality of the focal method under test. In particular, the code context part is similar to the basic prompt in Section 3.2, including the class declaration, constructor signatures, relevant fields, and the focal method itself; and the natural language instruction is to ask ChatGPT to infer the intention of the focal method. Then, the generation prompt further includes the generated intention and asks ChatGPT to generate a unit test for the focal method.

Figure 8 presents an example comparing how the basic prompt and the initial test generator generate a test for the same given focal method *"setCharAt()"*. As shown in Figure 8 (a), given the basic prompt without any intention inference, ChatGPT generates a test with an incorrect assertion (*i.e., "assertEquals(Hello-World, strBuilder.toString())"*. However, in Figure 8 (b), with the intention prompt, ChatGPT first correctly generates the intention for the focal method *"setCharAt()"*; and then with the generation prompt, ChatGPT generates a test with a correct assertion (*i.e., "assertEquals(Hello Jorld, strBuilder.toString())"*. The additional intention inference is designed to enhance ChatGPT's understanding about the focal method, which further leads to more accurate assertion generation.
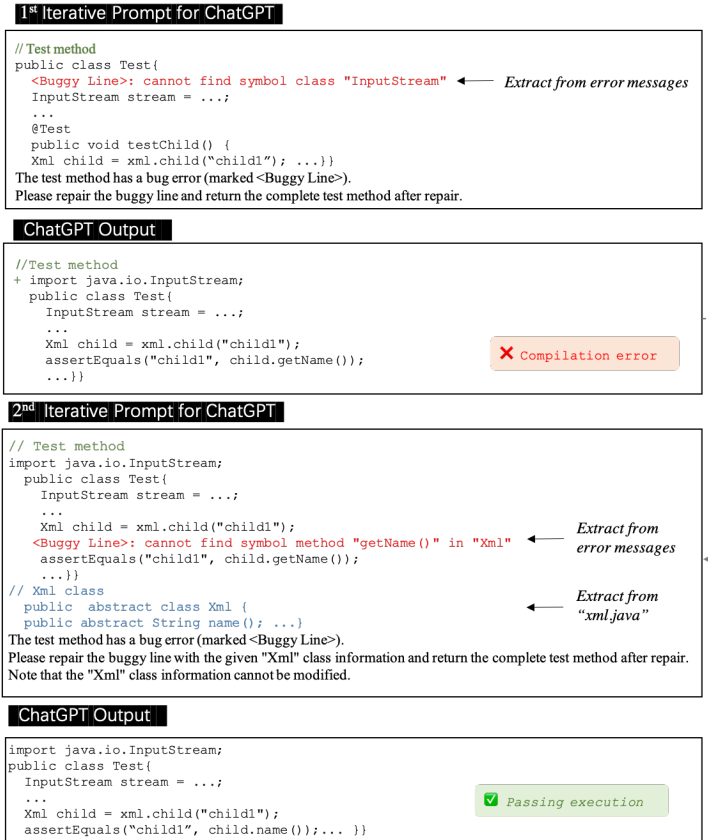

Fig. 9. Prompt in the Iterative Test Refiner

5.2 Iterative Test Refiner

The iterative test refiner iteratively fixes the compilation errors in the tests generated by the initial test generation. Each iteration successively leverages two steps: (i) first validating the generated test by compiling it in a validation environment; (ii) second constructing a prompt based on the error message during compilation and the extra code context related to the compilation error. The new prompt is then queried into ChatGPT to get a refined test. Such a procedure repeats until the generated test can be successfully compiled or the maximum number of iterations is reached. Currently, we only focus on fixing compilation errors instead of execution errors, since in practice it is challenging to identify whether a test execution failure is caused by the incorrect test code or by the incorrect focal method. We then explain each step with the illustration example in Figure 9.

Validator. For ease of compiling the generated test, we directly create a test file in the same directory of the focal class. In particular, the generated test method is encapsulated in a test class with relevant import statements. Then, the test file is compiled with the Java compiler. A controller then decides the next step based on the compilation status:

- *Successful compilation:* if there is no compilation error, the controller would terminate the iterative refinement procedure and return the final test;
- *Valid refinement:* if the number of compilation errors is less than that in the last iteration, the current refinement is considered as a valid refinement. The controller then proceeds to the iterative prompt constructor so as to continue the refinement;
- *Invalid refinement:* if the number of compilation errors is larger than or the same as that in the last iteration, the current refinement is considered an invalid refinement. The controller would terminate the refinement if the accumulated number of invalid refinements is larger than the maximum (*e.g.,* 3 in our experiments); or proceeds to the iterative prompt constructor.

Iterative Prompt Constructor. The iterative prompt constructor is built on top of (i) an EM parser that analyzes the error message about the compilation error, and (ii) a code analyzer that extracts the additional code context related to the compilation error.

In particular, the EM parser collects three types of information by parsing the error message:

- *Error type*: the high-level description about the error, which is often the first sentence in the error message. For example, *"cannot find simple class ..."* and *"cannot find symbol method ..."* are extracted error types in the illustration example.
- *Buggy location*: the line number of the test code triggering compilation errors. With such location information, the prompt constructor is able to insert the relevant information around the buggy line, *i.e.,* starting with the tag *"⟨Buggy line⟩"* as shown in the example.
- *Buggy element*: the objects or variables in the buggy location. For example, for the second iteration in Figure 9, we analyze the buggy line with the error message, and find that they are associated with the class *"Xml"* (which is the buggy element in this example).

With the buggy elements, the code analyzer is then able to extract additional code context from other Java files rather than the focal class. In particular, the code analyzer first parses the whole project to find the class file that the buggy element belongs to, and then extracts the class declaration and public method signature from the class file. This extracted class information would further be added to the prompt as additional information, *e.g., "//Xml class ..."* highlighted in blue. In fact, such additional information from other classes could be very important for generating high-quality tests, since it is very common that the test code involves not only the focal class but also other classes. However, given the limited input length for ChatGPT, it is infeasible to directly include the whole program in the prompt (which would also lead to bad performance since the

focus of ChatGPT might be confused). Therefore, in CHATTESTER, we propose to append necessary additional code contexts in such an iterative way.

6 EVALUATION OF CHATTESTER

We systematically evaluate CHATTESTER for its effectiveness (RQ5), its generalization capability on different LLMs (RQ6), and its project-level effectiveness (RQ7).

6.1 Evaluation Setup

RQ5 Setup. As for the evaluation dataset in RQ5, we re-sample another 100 data pairs from the remaining 748 data pairs in Section 3.1 (recall that we collect 1,748 data pairs in total and 1,000 of them are included in the benchmark for the empirical study). The reason that we construct such an additional evaluation dataset for evaluation is to avoid using the same benchmark that has been extensively analyzed in our previous study. Since our approach is inspired by the findings from our study, evaluating it on a separate dataset could eliminate the potential overfitting issues.

As for the studied techniques in RQ5, to evaluate the overall effectiveness of CHATTESTER and the individual contribution of each component (*i.e.,* the initial test generator and the iterative test refiner) in CHATTESTER, we study four techniques (1) ChatGPT: the default ChatGPT with the basic prompt, which is the one used in our empirical study; (2) CHATTESTER-ITE: a variant of CHATTESTER without the iterative test refiner, which actually enhances the default ChatGPT with the initial test generator of CHATTESTER; (3) CHATTESTER-INI: a variant of CHATTESTER without the initial test generator, which actually enhances the default ChatGPT with the iterative test refiner of CHATTESTER; (4) CHATTESTER: the complete CHATTESTER with both the initial test generator and the iterative test refiner. To mitigate the randomness in ChatGPT, we repeat all experiments three times and present the average results.

RQ6 Setup. To evaluate the generalization capability of CHATTESTER with different LLMs, we replace ChatGPT in CHATTESTER with two different open-source LLMs, *i.e.,* CodeLlama-Instruct-34B [4] and CodeFuse-34B [15]. For both studied LLMs, we compare their effectiveness of test generation with (i) the basic prompt in a default way and (ii) with the CHATTESTER framework.

RQ7 Setup. To evaluate the project-level effectiveness of CHATTESTER, we select three representative projects from the 185 Java projects in Section 3.1. Table 6 presents the characteristics of the selected projects, which belong to different domains with 3k to 6k lines of code. We apply CHATTESTER and all baselines (*i.e.,* AthenaTest, ChatGPT, and Evosuite) to generate tests for each method in the project. To evaluate the test sufficiency in such a project-level setting, we further compare the statement coverage and branch coverage of the generated tests on each entire project. For fair comparison, we randomly select one test when Evosuite generates multiple tests for one focal method. In addition, we further perform a user study to manually compare the readability and usability of the tests generated by CHATTESTER and baselines on a statistically-sampled subset (*i.e.,* 219 tests for each technique, which are sampled within the 0.05 error margin at a 95% confidence level [70, 75]). We follow the similar user study methodology in Section 3.4, *i.e.,* the five participant are asked to score the readability and usability of the generated tests based on the scoring details in Table 1.

Table 6. Overview of Projects in RQ7

Project Name	# Line Of Code	Domain
zappos-json [86]	3,552	Serialization
tabula-java [72]	5,586	File processing
jInstagram [41]	6,303	Instagram API wrapper

Table 7. Effectiveness of CHATTESTER

Metrics (%)	ChatGPT	CHATTESTER-ITE	CHATTESTER-INI	CHATTESTER
Syntactical correct	100.0	100.0	100.0	100.0
Success compilation	39.0	50.7	60.6	73.3
Passing execution	22.3	29.7	34.0	41.0

6.2 Evaluation Results

6.2.1 RQ5: Effectiveness Evaluation. Table 7 presents the correctness of the tests generated by ChatGPT and our approaches. Overall, we could observe a substantial improvement in both the compilation rate and passing rate of the tests generated by CHATTESTER compared to the default ChatGPT. For example, additional 34.3% tests (= 73.3% - 39.0%) can be successfully compiled and additional 18.7% tests (= 41.0% - 22.3%) can pass the execution. In summary, the proposed approach CHATTESTER effectively improves the correctness of the tests generated by ChatGPT.

In addition, we could observe that the variant CHATTESTER-ITE outperforms the default ChatGPT by achieving 11.7% and 7.4% improvements in the compilation rate and passing rate. In particular, we find that among the ChatGPT-generated tests with incorrect assertions, 12.5% of them are fixed into correct assertions in CHATTESTER-ITE, indicating the effectiveness of the initial test generator. Figure 8 is an example of how the intention prompt improves the correctness of assertions in our dataset. In addition, when comparing CHATTESTER against the variant CHATTESTER-INI, we can find that removing the initial test generator would decrease 12.7% compilation rate and 7.0% passing rate, further confirming the contribution of the initial test generator. Moreover, we could observe a further improvement from CHATTESTER-ITE to CHATTESTER, *i.e.,* additional 22.6% tests and 11.3% tests are fixed by the iterative test refiner into compilable tests and passing tests. Figure 9 is an example of how the iterative test refiner fixes the compilation errors in two iterations. In summary, both components (*i.e.,* the initial test generator and the iterative test refiner) positively contribute to the effectiveness of CHATTESTER.

Table 8. Distribution of Iteration Numbers in CHATTESTER

# Iteration	0	1	2	3	4	5	6	7	8	9	10
# Tests	37	12	9	0	1	12	18	10	0	0	1
# Success Compilation	31	10	6	0	1	0	15	6	0	0	0
# Passing Execution	31	9	2	0	0	0	0	0	0	0	0

Costs and number of iterations. Table 8 presents the distribution of iterative test refinement times of CHATTESTER. The row "# Iteration" shows the number of iterations; the row "# Tests " shows the number of tests that are still being refined in this iteration; the row "# Success Compilation" and the row "# Passing Execution" show the number of tests that are successfully compiled and executed in the current iteration, respectively. Based on the table, we find that the iterative refinement continuously improves the quality of the generated tests. The average time cost of CHATTESTER is around 99.0 seconds, where the initial test generator takes 15.0 seconds while each iteration in the iterative refiner takes 30.0 seconds on average. In fact, the majority of the costs come from the latency of querying the ChatGPT API (*i.e.,* around 10s in our environment) and the test compilation/execution. Therefore, the efficiency of CHATTESTER could be further improved if it is applied on locally-deployed small LLMs or with more efficient compilation optimization (*e.g.,* incrementally compiling only the generated test instead of compiling the whole project).

Effectiveness Evaluation Summary: CHATTESTER effectively improves the correctness of ChatGPT-generated tests by substantially reducing the compilation errors and incorrect assertions in the generated tests. In particular, both the initial test generator and the iterative test refiner positively contribute to the effectiveness of CHATTESTER.

Table 9. Generalization of CHATTESTER

Metric (%)	CodeLlama-34B		CodeFuse-34B	
	CodeLlama	*CodeLlama*$_{\text{CHATTESTER}}$	*CodeFuse*	*CodeFuse*$_{\text{CHATTESTER}}$
Syntactical Correct	93.0	93.0	72.0	89.0
Success Compilation	22.0	43.0	1.0	24.0
Passing execution	3.0	21.0	0.0	11.0

Table 10. Statement and Branch Coverage in Project-level Evaluation

Coverage (%)	zappos-json		tabula-java		jInstagram	
	SC	BC	SC	BC	SC	BC
AthenaTest	7.2	1.3	3.2	0.4	18.9	1.6
EvoSuite	30.5	16.1	15.2	9.9	45.3	13,3
ChatGPT	14.4	5.2	9.4	5.0	38.6	12.7
ChatTester	35.1	21.5	15.6	9.2	41.8	13.4

6.2.2 RQ6: Generalization Evaluation. Table 9 presents the effectiveness of CHATTESTER on different LLMs. Overall, CHATTESTER can consistently improve the quality (*e.g.,* syntactical, compilation, or execution correctness) of the tests generated by different open-source LLMs. For the studied LLMs CodeLlama-Instruct-34B and CodeFuse-34B, CHATTESTER achieves 21.0%/ 23.0% improvement in compilation rate and 11.0% /18.0% improvement in passing execution rate. The results indicate that the approach of CHATTESTER can be generalized to different LLMs and is not specific to the commercial LLM ChatGPT. The average time costs of CodeLlama-Instruct, *CodeLlama*$_{\text{CHATTESTER}}$, CodeFuse and *CodeFuse*$_{\text{CHATTESTER}}$ are 5s, 105s, 5s and 90s.

> **Generalization Evaluation Summary:** CHATTESTER can be generalized to different LLMs and consistently improve their effectiveness of test generation.

6.2.3 RQ7: Project-level Evaluation. We present the results of coverage comparison and the user study results of readability and usability as follows.

Project-level Coverage. Table 10 shows the statement coverage ("SC") and branch coverage ("BC") of the tests generated by CHATTESTER and all the baselines over the entirety of each project. We can find that CHATTESTER achieves comparable and even better coverage than the baselines in the project-level evaluation setting. We further inspect some cases and analyze the potential reason for the observation. In particular, compared to the deep-learning-based technique AthenaTest and the basic ChatGPT, CHATTESTER is capable of generating more compilable and executable tests, thus inherently resulting in higher coverage; for the search-based technique Evosuite, although it is able to generate more executable tests, its generated tests fall short in containing unnatural inputs and thus cannot reach some specific statements.

Readability and Usability. Figure 10 presents the overall score distribution of the readability and usability. In particular, A to E groups represent different participants, and X/Y/Z represent different test generation techniques. As shown in the figure, all the participants find that CHATTESTER generate more tests with better readability and usability than baselines. In addition, based on the scoring details of naming intuitiveness, code layout, assertion quality, and adoption efforts, we find that participants consider the CHATTESTER-generated tests with more intuitive naming styles and more structured code (thus with better readability), and participants also consider the CHATTESTER-generated tests with better assertions and less adoption efforts (thus with better usability).

> **Project-level Evaluation Summary:** CHATTESTER consistently outperforms baselines in coverage, readability, and usability in the project-level evaluation setting.
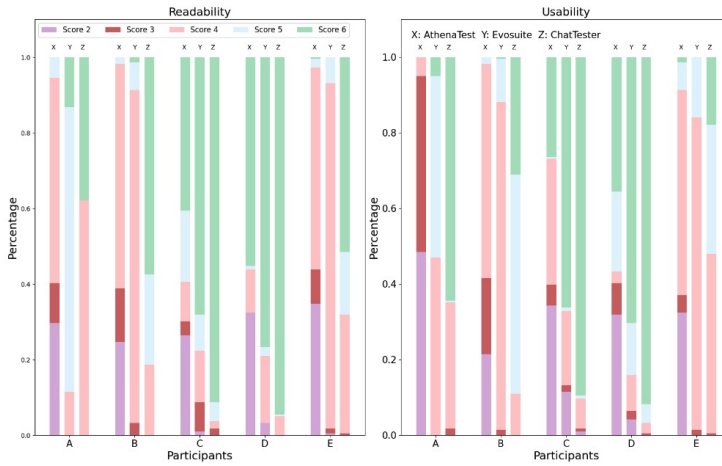
Fig. 10. Score Distribution of Readability and Usability

7 THREATS TO VALIDITY

(i) One threat lies in the randomness in ChatGPT. To alleviate this issue, we repeat our experiments three times and present the average results when automatically evaluating the effectiveness of ChatTester. We do not repeat our experiments in the empirical study, due to the large manual efforts involved in the user study. However, we actually observe similar correctness results of the tests generated by ChatGPT on two different datasets (Table 2 and Table 7), indicating the consistency of our results. (ii) Another threat is in the benchmarks used in this work. Our findings might not generalize to other datasets. To eliminate this issue, we construct our datasets to include more high-quality projects and diverse focal methods and test methods, and we evaluate the proposed ChatTester on a different evaluation dataset to avoid overfitting issues. In addition, to evaluate the generalization capability of ChatTester on different programming languages, we further perform an additional evaluation of ChatTester on a Python dataset, HumanEval [38]. In particular, we leverage ChatTester to generate tests for 164 Python methods in HumanEval. We observed the consistent effectiveness of ChatTester by improving the successful execution rate of ChatGPT-generated tests from 31.7% to 43.2%. The results indicate the generalization of ChatTester to other programming languages. (iii) Another threat is the potential data leakage of the manually-written tests being part of the training data in ChatGPT, which might lead to the overestimation of ChatGPT's capability in test generation. In fact, we find there is quite long Levenshtein edit distance between the ChatTester-generated tests and manually-written tests (*i.e.,* average of 1,027 characters and median of 609 characters), and only 5% of ChatTester-generated tests have less than 200-characters Levenshtein distance to manually-written ones, implying that directly copying from training data might be less prevalent in our benchmark.

8 RELATED WORK

8.1 ChatGPT for SE Tasks

ChatGPT has attracted wide attention due to its outstanding capability of solving various tasks [30, 40, 47, 88], including the SE tasks [13, 25, 71, 77], *e.g.,* program repair [71, 82], code generation [25, 32, 66]. Our work performs the first study to explore the ChatGPT's ability for unit test generation; and then proposes ChatTester, a novel ChatGPT-based unit test generation approach, which improves the quality of the tests generated by ChatGPT.

8.2 LLMs for Test Generation

One typical category of LLM-based test generation techniques mainly regard test generation as a neural machine translation problem [54, 73] (*i.e.,* from the focal method to the corresponding test prefix or the test assertion) and fine-tune the LLMs on the test generation dataset. For example, AthenaTest [73] fine-tunes BART [14] on a test generation dataset where the input is the focal method with the relevant code context while the output is the complete test case. Our results show that ChatGPT outperforms AthenaTest in generating tests of higher correctness and coverage.

More recently, given the rapid development of instruction-tuned LLMs, there are an increasing number of test generation techniques that leverage instructed LLMs via suitable prompts instead of fine-tuning models [21, 22, 81]. For example, breaking down complex tasks into smaller tasks (such as chain of thought reasoning [80] and tree of thought reasoning [85]) has been demonstrated as effective prompting strategies, which are also high-level ideas inspiring CHATTESTER. Nashid et al. [53] propose a retrieval-based prompt construction strategy that queries Codex with few shots to generate test assertions. Different from their work, our work focuses on the zero-shot learning scenario and generates complete tests of both test prefixes and test assertions. CODAMOSA [46] enhances traditional search-based techniques by using tests generated by Codex to escape from the "plateaus" during the search procedure. In fact, CHATTESTER is complementary to CODAMOSA, since CHATTESTER aims at generating one test for the given focal method via LLM while CO-DAMOSA aims at generating a set of tests for the given module via the combination of LLM and search-based algorithm. Therefore, CHATTESTER could provide higher-quality of tests as the seed for CODAMOSA during its search procedure. LIBRO [43] leverages Codex to generate tests for the given bug report. Different from LIBRO, our work focuses on the test generation scenario without bug reports. This September, Li et al. [48] propose differential prompts to generate failure-inducing test inputs via ChatGPT, which focuses on generating *test inputs* via LLMs, while our work focuses on generating a complete unit test case of both test inputs and test code (*e.g.,* the test prefix). Ring System [42] and TestPilot [69] also employ LLM for test generation and bug repair, respectively. Ring System adopts a one-turn approach focusing on a specific code snippet, whereas our tool, CHATTESTER, leverages iterative refinement through multi-round dialogues with LLMs and incorporates additional code context for enhanced accuracy. TestPilot differs from CHATTESTER in its feedback mechanism by searching for code snippets using the focal methods. In contrast, CHATTESTER extracts and utilizes additional contextual information from error messages for refining test generation.

9 CONCLUSION

In this work, we perform the first empirical study to evaluate ChatGPT's capability of unit test generation, by systematically investigating the correctness, sufficiency, readability, and usability of its generated tests. We find that the tests generated by ChatGPT still suffer from correctness issues, including diverse compilation errors and execution failures (mostly caused by incorrect assertions); but the passing tests resemble manually-written tests by achieving comparable coverage, readability, and even sometimes developers' preference. Inspired by our findings above, we further propose CHATTESTER, which leverages ChatGPT itself to improve the quality of its generated tests. Our evaluation demonstrates the effectiveness of CHATTESTER by generating 34.3% more compilable tests and 18.7% more tests with correct assertions than the default ChatGPT.

REFERENCES

[1] 2019. http://javaparser.org/. (2019).

[2] 2022. https://the-decoder.com/chatgpt-guide-prompt-strategies/. (2022).

[3] 2022. https://www.jacoco.org/jacoco/. (2022).

[4] CodeLlama 34b Instruct. 2023. (2023). https://huggingface.co/codellama/CodeLlama-34b-Instruct-hf

[5] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021.* Association for Computational Linguistics, 2655–2668.

[6] Mohammad Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017.* IEEE Computer Society, 263–272.

[7] Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. 2022. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862* (2022).

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

[9] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2022. Call Me Maybe: Using NLP to Automatically Generate Unit Test Cases Respecting Temporal Constraints. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022.* ACM, 19:1–19:11.

[10] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2022. Call Me Maybe: Using NLP to Automatically Generate Unit Test Cases Respecting Temporal Constraints. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022.* ACM, 19:1–19:11.

[11] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T. Devanbu, and Baishakhi Ray. 2022. NatGen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022.* ACM, 18–30.

[12] ChatTESTER. 2023. (2023). https://github.com/FudanSELab/ChatTester/tree/main

[13] Eason Chen, Ray Huang, Han-Shin Chen, Yuen-Hsien Tseng, and Liang-Yi Li. 2023. GPTutor: A ChatGPT-Powered Programming Tool for Code Explanation *(Communications in Computer and Information Science, Vol. 1831)*. Springer, 321–327.

[14] Hugh A Chipman, Edward I George, and Robert E McCulloch. 2010. BART: Bayesian additive regression trees. (2010).

[15] CodeFuse-CodeLlama-34B. 2023. (2023). https://huggingface.co/codefuse-ai/CodeFuse-CodeLlama-34B

[16] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: dynamic symbolic execution for invariant inference. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008.* ACM, 281–290.

[17] Andrew M. Dai and Quoc V. Le. 2015. Semi-supervised Sequence Learning. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada.* 3079–3087.

[18] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. 2015. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015.* ACM, 107–118.

[19] Ermira Daka and Gordon Fraser. 2014. A Survey on Unit Testing Practices and Problems. In *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014.* IEEE Computer Society, 201–211.

[20] Pedro Delgado-Pérez, Aurora Ramírez, Kevin J. Valle-Gómez, Inmaculada Medina-Bulo, and José Raúl Romero. 2023. InterEvo-TR: Interactive Evolutionary Test Generation With Readability Assessment. *IEEE Trans. Software Eng.* 49, 4 (2023), 2580–2596.

[21] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023.* ACM, 423–435.

[22] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024.* ACM, 70:1–70:13.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, 4171–4186.

[24] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. TOGA: A Neural Method for Test Oracle Generation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2130–2141.

[25] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration Code Generation via ChatGPT. *CoRR* abs/2304.07590 (2023). arXiv:2304.07590

[26] Xueying Du, Mingwei Liu, Juntao Li, Hanlin Wang, Xin Peng, and Yiling Lou. 2023. Resolving Crash Bugs via Large Language Models: An Empirical Study. *CoRR* abs/2312.10448 (2023). arXiv:2312.10448

[27] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation. *CoRR* abs/2308.01861 (2023). arXiv:2308.01861

[28] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1-3 (2007), 35–45.

[29] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*. Association for Computational Linguistics, 1536–1547.

[30] Mohammad Fraiwan and Natheer Khasawneh. 2023. A Review of ChatGPT Applications in Education, Marketing, Software Engineering, and Healthcare: Benefits, Drawbacks, and Research Directions. *CoRR* abs/2305.00237 (2023). arXiv:2305.00237

[31] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. ACM, 416–419.

[32] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R. Lyu. 2023. What Makes Good In-Context Demonstrations for Code Intelligence Tasks with LLMs?. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 761–773.

[33] Shuzheng Gao, Hongyu Zhang, Cuiyun Gao, and Chaozheng Wang. 2023. Keeping Pace with Ever-Increasing Data: Towards Continual Learning of Code Intelligence Models. *CoRR* abs/2302.03482 (2023). arXiv:2302.03482

[34] getEnvironment(). 2016. (2016). https://github.com/trautonen/coveralls-maven-plugin/blob/master/src/main/java/org/eluder/coveralls/maven/plugin/service/Travis.java#L75

[35] Giovanni Grano, Fabio Palomba, Dario Di Nucci, Andrea De Lucia, and Harald C. Gall. 2019. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. *J. Syst. Softw.* 156 (2019), 312–327.

[36] Giovanni Grano, Simone Scalabrino, Harald C. Gall, and Rocco Oliveto. 2018. An empirical investigation on the readability of manual and generated test cases. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*. ACM, 348–351.

[37] Mark Harman and Phil McMinn. 2010. A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. *IEEE Trans. Software Eng.* 36, 2 (2010), 226–247.

[38] HumanEval. 2021. (2021). https://github.com/openai/human-eval

[39] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[40] Sajed Jalil, Suzzana Rafi, Thomas D. LaToza, Kevin Moran, and Wing Lam. 2023. ChatGPT and Software Testing Education: Promises & Perils. *CoRR* abs/2302.03287 (2023). arXiv:2302.03287

[41] jInstagram. 2015. (2015). https://github.com/sachin-handiekar/jInstagram

[42] Harshit Joshi, José Pablo Cambronero Sánchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radicek. 2023. Repair Is Nearly Generation: Multilingual Program Repair with LLMs. AAAI Press, 5131–5140.

[43] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2312–2323.

[44] Claus Klammer and Albin Kern. 2015. Writing unit tests: It's now or never!. In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 1–4.

[45] Elson Kurian, Daniela Briola, Pietro Braione, and Giovanni Denaro. 2023. Automatically generating test cases for safety-critical software via symbolic execution. *J. Syst. Softw.* 199 (2023), 111629.

[46] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 919–931.

[47] Bo Li, Gexiang Fang, Yang Yang, Quansen Wang, Wei Ye, Wen Zhao, and Shikun Zhang. 2023. Evaluating ChatGPT's Information Extraction Capabilities: An Assessment of Performance, Explainability, Calibration, and Faithfulness. *CoRR* abs/2304.11633 (2023). arXiv:2304.11633

[48] Tsz On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. 2023. Nuances are the Key: Unlocking ChatGPT to Find Failure-Inducing Tests with Differential Prompting. *38th IEEE/ACM International Conference on Automated Software Engineering (ASE 2023), 11-15 September 2023, Kirchberg, Luxembourg* (2023).

[49] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated Unit Test Generation for Python. In *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*. ACM/IEEE, 168–172.

[50] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2023. An empirical study of automated unit test generation for Python. *Empir. Softw. Eng.* 28, 2 (2023), 36.

[51] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader-Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 336–347.

[52] Webb Miller and David L. Spooner. 1976. Automatic Generation of Floating-Point Test Data. *IEEE Trans. Software Eng.* 2, 3 (1976), 223–226.

[53] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2450–2462.

[54] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2111–2123.

[55] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

[56] Michael Olan. 2003. Unit testing: test early, test often. *Journal of Computing Sciences in Colleges* 19, 2 (2003), 319–328.

[57] OpenAI. 2023. ChatGPT: Optimizing Language Models for Dialogue. *https://openai.com/blog/chatgpt/* (2023).

[58] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, and etc. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

[59] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 75–84.

[60] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. 2016. On the diffusion of test smells in automatically generated test code: an empirical study. In *Proceedings of the 9th International Workshop on Search-Based Software Testing, SBST@ICSE 2016, Austin, Texas, USA, May 14-22, 2016*. ACM, 5–14.

[61] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2016. Automatic test case generation: what if test code quality matters?. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. ACM, 130–141.

[62] Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2024. AgentFL: Scaling LLM-based Fault Localization to Project-Level Context. *CoRR* abs/2403.16362 (2024). arXiv:2403.16362

[63] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[64] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67.

[65] Prajit Ramachandran, Peter J. Liu, and Quoc V. Le. 2017. Unsupervised Pretraining for Sequence to Sequence Learning. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*. Association for Computational Linguistics, 383–391.

[66] Xiaoxue Ren, Xinyuan Ye, Dehai Zhao, Zhenchang Xing, and Xiaohu Yang. 2023. From Misuse to Mastery: Enhancing Code Generation with Knowledge-Driven AI Chaining. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 976–987.

[67] Per Runeson. 2006. A Survey of Unit Testing Practices. *IEEE Softw.* 23, 4 (2006), 22–29.

[68] Simone Scalabrino, Giovanni Grano, Dario Di Nucci, Michele Guerra, Andrea De Lucia, Harald C. Gall, and Rocco Oliveto. 2018. OCELOT: a search-based test-data generation tool for C. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018.* ACM, 868–871.

[69] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Trans. Software Eng.* 50, 1 (2024), 85–105.

[70] Ravindra Singh and Naurang Singh Mangat. 2013. *Elements of survey sampling.* Vol. 15. Springer Science & Business Media.

[71] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. *CoRR* abs/2301.08653 (2023). arXiv:2301.08653

[72] tabula java. 2017. (2017). https://github.com/tabulapdf/tabula-java

[73] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020).

[74] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA.* 5998–6008.

[75] Dennis Wackerly, William Mendenhall, and Richard L Scheaffer. 2014. *Mathematical statistics with applications.* Cengage Learning.

[76] Chong Wang, Jianan Liu, Xin Peng, Yang Liu, and Yiling Lou. 2023. Boosting Static Resource Leak Detection via LLM-based Resource-Oriented Intention Inference. *CoRR* abs/2311.04448 (2023). arXiv:2311.04448

[77] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2023. Software Testing with Large Language Model: Survey, Landscape, and Vision. *CoRR* abs/2307.07221 (2023). arXiv:2307.07221

[78] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021.* Association for Computational Linguistics, 8696–8708.

[79] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020.* ACM, 1398–1409.

[80] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022.*

[81] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal Fuzzing via Large Language Models. *CoRR* abs/2308.04748 (2023). arXiv:2308.04748

[82] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *CoRR* abs/2304.00385 (2023). arXiv:2304.00385

[83] Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. 2013. Characteristic studies of loop problems for structural test generation via symbolic execution. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013.* IEEE, 246–256.

[84] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada.* 5754–5764.

[85] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023.*

[86] zappos json. 2016. (2016). https://github.com/Zappos/zappos-json

[87] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. The fuzzing book.

[88] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, and etc. 2023. A Survey of Large Language Models. *CoRR* abs/2303.18223 (2023). arXiv:2303.18223

[89] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.* 29, 4 (1997), 366–427.