A Cascaded Pipeline for Self-Directed, Model-Agnostic Unit Test Generation via LLMs

Chao Ni*
Zhejiang University
Hangzhou, China
chaoni@zju.edu.cn

Xiaoya Wang*
Douyin Co., Ltd.
Shenzhen, China
wangxiaoya.2000@bytedance.com

Xin Yin
Zhejiang University
Hangzhou, China
xyin@zju.edu.cn

Liushan Chen†
Douyin Co., Ltd.
Shenzhen, China
chenliushan@bytedance.com

Guojun Ma
Douyin Co., Ltd.
Shenzhen, China
maguojun@bytedance.com

***Abstract*—While existing ML-based unit test generation methods show promising results, they face three key limitations: (1) incomplete test case generation with excessive focus on test oracles, (2) semantic inconsistencies between test components, and (3) dependency on closed-source models compromising data security. In this paper, we propose a novel approach named CasModaTest, a cascaded, model-agnostic, and end-to-end unit test generation framework, to alleviate the above limitations. Specifically, CasModaTest first splits the unit test generation task as two cascaded steps: test prefix generation and test oracle generation. Then, to better stimulate models' learning ability, we manually build large-scale demo pools to provide CasModaTest with high-quality test prefixes and test oracles examples. Finally, CasModaTest assembles test components and validates their functionality through execution, with error correction during compilation/runtime. Our evaluation on the Defects4J benchmark demonstrates CasModaTest's superiority over five state-of-the-art approaches, showing significant improvements in both accuracy and focal method coverage. Further validation across 1,625 methods from six real-world projects reveals that CasModaTest achieves substantially higher code coverage metrics (method/line/branch coverage) compared to the dedicated coverage tool EvoSuite.**

***Index Terms*—Unit Test Generation, Large Language Model, Model-agnostic, Cascaded Pipeline**

I. Introduction

Unit testing plays a critical role in software maintenance to assure the quality of software systems, which helps developers identify defects and errors as early as possible in the development process, reducing the overall cost of the product and hence improving developers' productivity [1], [2]. However, manually writing high-quality unit tests is a difficult and time-consuming task. Thus, various automated test case generation approaches have been proposed and these approaches can be divided into two categories: traditional unit test generation [3], [4] and machine learning-based unit test generation [5]–[7]. These approaches have achieved promising results and indeed make progress in unit test generation, especially the machine learning-based approaches that have attracted much attention from academia to industry. However, machine learning-based approaches still have several limitations in previous studies including (1) Impractical Usage, (2) Unpaired Prefix and Oracle, and (3) Close-sourced Binded Model. More precisely, a complete unit test case should consist of two parts: test prefix and test oracle. In practical applications, developers need to construct sufficiently diverse test prefixes and sufficiently accurate test oracles (assertions or exceptions) to ensure the correctness of software units. However, currently, most machine learning-based approaches focus on a part of unit test, such as generating test oracle only with test prefix given [8]–[10], which is not suitable for practical usage since the participants still need to carefully prepare the partial unit test. Meanwhile, there are two types of unit tests [10]: tests with assertion oracles and tests with expected exception oracles. The former verifies the correctness of the return behavior, although they fail if any exception occurs, while the latter verifies whether the execution of the test prefix with invalid usage can raise a particular exception. Therefore, the test prefix should be well semantically connected with the corresponding test oracle. For example, the test prefix with the semantics of assertion should be assembled with the assertion oracle, the same applies to the test prefix that embodies the semantics of expected exception. However, though several approaches [5], [6], [11] can generate a complete unit test, they may not well consider the connection between the test prefix and test oracle, which leads to an incorrect unit test.

To address the aforementioned limitations, we proposed an automatic, cascaded, and LLM-agnostic unit test generation framework, CasModaTest, with two phases: the generation phase and the verification phase. The former phase generates a test prefix for the focal method and a corresponding semantic-related test oracle. In the latter phase, the test prefix and the test oracle will be combined as a complete unit test for verification and following that, an automatic self-debug process will be triggered for addressing compile or execution errors with limited interactions with LLMs (e.g., ChatGPT or CodeLlama). To achieve better unit tests by adopting LLM's in-

*Equal contribution.
†Corresponding author.

context learning ability, we manually built two demonstration pools based on the widely used Defect4J dataset [12] for generating test prefixes and test oracles separately. We evaluate the effectiveness of CasModaTest by comparing with two state-of-the-art deep learning-based (i.e., AthenaTest [5], A3Test [6]) approaches, one ChatGPT-based approach (i.e., ChatUnitTest [7]) and one approach based on genetic algorithms (i.e., EvoSuite [4]) with three widely used performance metrics (i.e., accuracy, focal method coverage and code coverage).

The experimental results indicate the priority of CasModaTest. More precisely, CasModaTest equipped with closed source LLM (i.e., gpt-3.5-turbo) significantly improves SOTAs by 60.62%-352.55% and 2.83%-87.27% in terms of accuracy and focal method coverage, respectively. CasModaTest equipped with open-sourced LLM (i.e., DeepSeek, Phind-CodeLlama) also obtains a great improvement over SOTAs by 39.82%-293.96% and 9.25%-98.95% in terms of accuracy and focal method coverage, respectively. Finally, we evaluate the code coverage of CasModaTest on 1625 representative methods from 6 projects. The experimental results show that CasModaTest achieves 88.46%, 64.88%, and 50.26% in terms of method coverage, line coverage, and branch coverage, respectively, surpassing EvoSuite, which targets code coverage. Eventually, our contributions are listed as follows:

- **A. Novel Self-directed LLM-based Unit Test Generation:** CasModaTest advances LLM-agnostic unit test generation. We show that the two-stage and LLM-agnostic unit test generation framework can achieve better results over existing unit test generation directions.
- **B. Structured Demonstration Pool:** We manually build two types of demonstration pools for further prompting the two-stage unit test generation with the in-context learning ability of LLM.
- **C. Extensive Empirical Evaluation:** We evaluate CasModaTest against current SOTA deep learning-based and LLM-based tools on the widely studied Defects4J [12]. CasModaTest outperforms the existing SOTA unit test generation approaches.

We release the replication package of our work to facilitate future research at: https://figshare.com/s/bee2ca4b1801cfc62bd2.

II. Motivation Example

Fig. 1 shows a method named `add` in the class *ArrayUnits* from the Apache Commons Lang project [13] and its two unit test cases. The function *add* is to add an element to an array at a particular position (i.e., *index*). Particularly, before adding an element to the array, it is necessary to check that both the element's value and the array itself are not *NULL*. Meanwhile, the given index should be inside the array's range. If the given element and array are not *NULL*, this method inserts the element into the target array at the given position. Therefore, there are two types of unit tests: tests with assertion oracles and tests with expected exception oracles, defined by previous work [10] by observing almost 200K developer-written tests. The **Test with Assertion Oracles** verify the correctness of the return behavior, although they fail if any exception occurs, which have several common assertion patterns, such as Boolean Assertions (e.g., `assertTure`, `assertFalse`), Nullness Assertion (e.g., `assertNull`, `assertNotNull`) and Equality Assertions (e.g., `assertEquals`, `assertArrayEquals`). The **Tests with Expected Exception Oracles** verify whether the execution of the test prefix with invalid usage can raise an exception. Usually, they are frequently expressed with the `try{···} catch(Exception e){···}` structure or `assertThrows` statement. The unit test illustrated in Fig. 1(b) verifies whether the element is successfully inserted into the target array, while the unit test illustrated in Fig. 1(c) checks to catch the expected exceptions. Based on the example above, we have the following observations:

Observation 1. A unit test is well structured with two parts: a prefix and an oracle. A complete unit test must contain both a test prefix and an oracle. Prior works [8]–[10] are impractical due to the lack of ability to generate full unit test cases that can be compiled and executed. We applied some SOTAs on the above example and they failed to generate complete tests, for example, EditAS [8]. Even though the SOTAs are designed to only generate assertions, such as TOGA [10], they are ineffective in generating the correct assertions due to the missing analysis of test prefixes.

Observation 2. Test prefixes and oracles should be semantically connected and correctly aligned in the test case generation. More precisely, a prefix for the test with an assertion oracle should be paired with a test oracle for the test with an assertion oracle, not the expected exception oracle unit test, and vice versa. Prior works [5]–[7] are designed to provide end-to-end capabilities of generating complete unit test cases. However, the alignment and different semantics of prefixes and oracles are not well considered and analyzed in the existing approaches, which leads to the incorrect unit test case generation that cannot be complied with and executed. For example, AthenaTest [5] can generate a complete unit test for the above example in Fig. 1(a), but the prefix and oracle are not matched, thus the generated unit test cannot execute successfully.

Observation 3. The method under test can be complex, in order to generate correct unit test cases, it requires the learning model to have extraordinary learning and understanding capabilities. Some recent LLM-based approaches, such as ChatTester [14] and ChatUniTest [7], leverage the eminent powerful learning capability of ChatGPT in the unit test case generation and can generate the correct test cases for the above example. However, the design of their approaches is heavily dependent on ChatGPT's learning capability. For example, the prompts designed by ChatUniTest [7] can be as long as 2700 tokens, which is impractical for models with a smaller text window (e.g., 1024 tokens). Moreover, even if some models can handle sequences of such length, the lack of strong command understanding capability makes it difficult for them to generate ideal responses.

Based on these observations, we have the following insights:

(1) **Practical Unit Test Generation.** Some previous works [5], [6], [10] only focus on generating a part of a

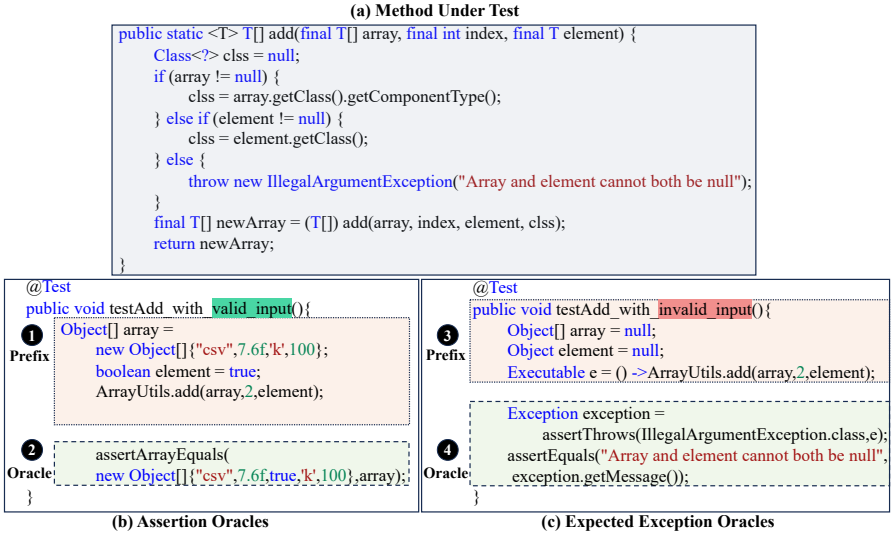

Fig. 1: (a) the method named `add` in class `ArrayUnits`; (b) an assertion oracle unit test to verify correct return behavior; (c) an expected exception oracle to verify that executing the test prefix with some invalid usage raises an exception.

unit test, mostly test oracle generation, which is impractical in the real industry setting as it still requires developers/testers to write a large chunk of a unit test case. Thus, we aim to provide an end-to-end unit test case generation capability that takes the focal method under test as input and generate a fully executable unit test case with prefix and test oracles.

(2) **Cascaded Self-directed for Unit Test Generation.** Automatically generating test prefixes and test oracle is not a trivial task since it requires understanding the semantics of the method under test. Even with powerful understanding capabilities, LLMs still require some guidance in orchestrating target test prefixes and test oracles. Instead of directly generating test prefixes or test oracles, several examples can help LLM to better under the faced task, but the quality of examples will highly affect the capabilities. Thus, we design an automatic approach that selects effective examples for in-context learning and composes a prompt for unit test generation.

(3) **Model-agnostic Framework for Unit Test Generation.** Most LLM-based unit test generation approaches (e.g., ChatTester [14] and ChatUniTest [7]) highly rely on the powerful learning capability of the close-source model (i.e., ChatGPT), which means that the users have to transfer their sensitive data to that model and then obtain the corresponding responses. However, privacy data protection is extremely important for individual users, especially for organizations, which prevents the large usage of that type of model. Thus, we design a model-agnostic framework for unit test generation that can utilize both closed-source models and open-source models, which can ensure better performance and data safety simultaneously.

III. Approach

We propose a cascaded self-directed framework, CasModaTest, for unit test generation. CasModaTest uses instruction tuning combined with few-shot learning to enhance the analysis to understand the semantics of both functions and components in unit tests. **CasModaTest** has two main phases (illustrated in Fig. 2): ❶ **generation phase** and ❷ **verification phase**. The generation phase is to generate the prefix and oracle of a test case. The verification phase verifies the generated test case optionally appending with several attempts to fix compile or execution errors. The details of each phase are presented in the following subsections.

A. Generation Phase

This phase aims to generate separate parts (i.e., prefix and oracle) of a unit test. To achieve this, we need to address three tasks: (1) Demonstration Pool Construction, (2) Prompt Preparation, and (3) Prefix and Oracle Generation.

1) Task 1: Demonstration Pool Construction: In few-shot learning [15], [16], high-quality demonstrations are extremely important to help LLMs better understand downstream tasks. Meanwhile, considering the inherent structure of the method under test written by object-oriented programming language, we split a unit test case into several parts for better representation. For example, one Java method is part of a particular class and its corresponding unit test case (i.e., another Java method) usually involves two parts: test prefix and test oracle. Thus, we separately build two types of demonstration pools: test prefix demo pool and test oracle demo pool.

In the prefix demo pool, each demo is split into five parts: (1) the class where the focal method is located; (2) the constructor parameters of the class; (3) the signature of the focal method; (4) the name of the test method; (5) the corresponding test prefix written by developers.

In the oracle demo pool, each demo can be also split into three parts: (1) the signature of the focal method; (2) the implementation body method under test but WITHOUT the test oracle written by developers (i.e., replaced by a special placeholder of `<OraclePlaceHolder>`); (3) the corresponding test oracle written by developers. The structure of the prefix and oracle demo is illustrated in the third part in Fig. 3 (marked as ③ Few-Shot Selection).

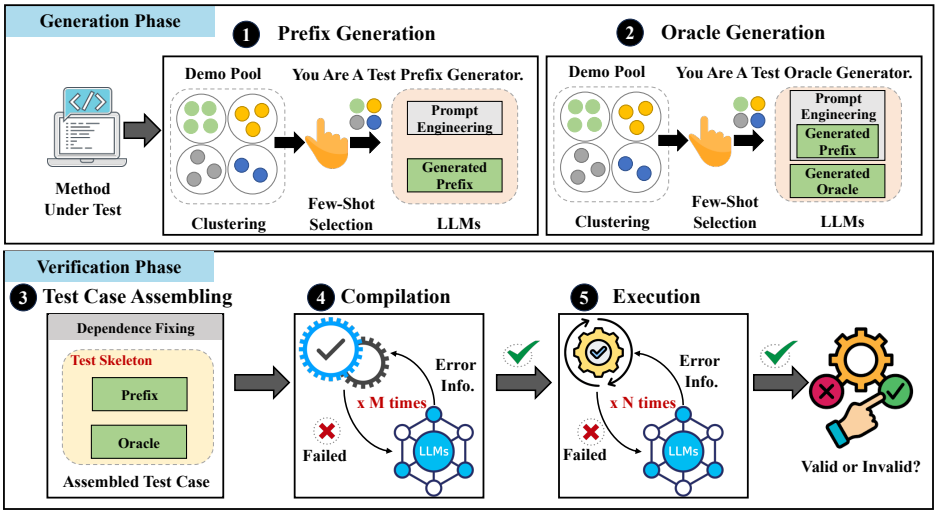

Fig. 2: The framework of CasModaTest.

2) Task 2: Prompt Preparation: The prompt used in CasModaTest for both test prefix generation and test oracle generation involves four important components as illustrated in Fig. 3:

- **Role Definition** (marked as ①). CasModaTest starts a role for a large language model with an instruction like *"You are a proficient and helpful assistant in java testing with JUnit framework"*. We adopt the same instruction for prefix and oracle generation.
- **Task Description** (marked as ②). The LLM is provided with different descriptions for different tasks. As for prefix generation, we instruct LLMs with such a description like *"Your task now is only to construct the test inputs, not the test assertions. Use CLASS_CONSTRUCTOR to get CLASS_NAME, then call TEST_METHOD_NAME. Use Java without comments. End your reply with END_OF_DEMO."* As for oracle generation, we instruct LLMs with another description like *"Your task now is to generate a test assertion to replace the <OraclePlaceHolder> in UNIT_TEST. Only variables that occur in the last UNIT_TEST can be used. Use Java without comments. End your reply with END_OF_DEMO."*. Meanwhile, the variables with the same names in the prefix prompt and oracle prompt indicate the same instances for building a better connection between the two parts in one test case.
- **Few-Shot Selection** (marked as ③). LLMs usually need a high-quality prompt to instruct themselves to finish the downstream tasks, and it is the focus of many works [17], [18]. Similarly, we design a structured template for representing a high-quality test case example (i.e., human-written ones), especially, we split a test case into several parts (e.g., driven class name, focal method signature, test prefix, etc.). To better inspire LLMs' capability, we need to identify the most beneficial examples from the demonstration pool. Meanwhile, previous work [19] also concludes that a few diverse examples may assist LLMs in achieving a better generalization ability. To achieve this diversity, CasModaTest clusters these examples inside the demonstration pool on the basis of their semantic similarity to pick out distinct ones [19], [20]. In particular, we adopt different structures to represent a test case for a better abstraction. In the prefix generation phase, a test case is composed of the class name (i.e., *demo.classname)*, the constructor of the class (i.e., *demo.constructor*), the signature of the method under test (i.e., *demo.focal_method_signature*) and the human-written test prefix (i.e., *demo.test_prefix*). In the oracle generation phase, a test case is composed of the signature of the method under test (i.e., *demo.focal_method_signature*), the test method containing placeholders (i.e.,*demo.test_name*, *demo.test_prefix* and *<OraclePlaceHolder>*) and the human-written test oracle (i.e., *demo.test_oracle*). Then, each example in the demonstration pool is encoded with a pre-trained language model (i.e., UniXcoder [21]). Furthermore, considering the limitation of LLMs' conversation windows, we cluster all examples in the demonstration pool into five clusters, and one sample with the highest cosine similarity with the target focal model is picked from each cluster (i.e., five shots used in this paper for both prefix and oracle generation). Eventually, three distinct sorting strategies are considered for ordering selected examples, and the details are elaborated as follows.
  - **Randomly Selection** first select the highest cosine similarity with the target focal method from each cluster and then randomly sort them in the prompt design.
  - **Ascending Selection** follows the "Randomly Selection" strategy to pick five examples and then sort them in ascending by cosine similarly in the prompt design.
  - **Descending Selection** follows the "Randomly Selection" strategy to pick five examples and then sort them in descending by cosine similarly in the prompt design.
- **Target Template** (marked as ④). Previous work [22] concludes that both the examples and target generation having the same structure in the few-shots learning scenario may yield a better performance. Therefore, we encode the target method under test in the same way as the examples in corresponding demonstration pools (i.e., prefix and oracle).

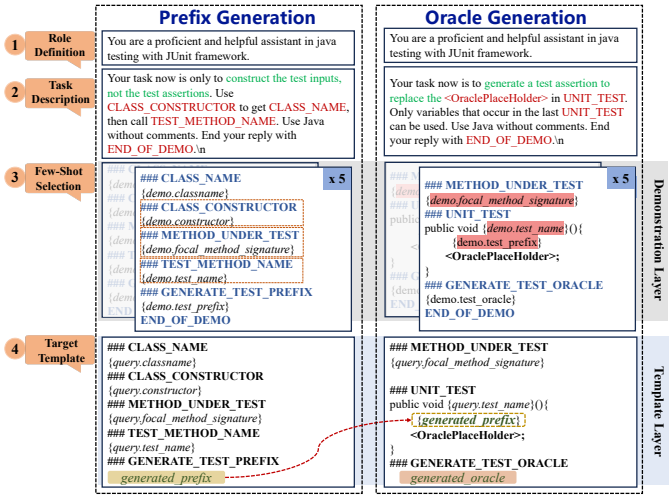

Fig. 3: The design of generation prompts: involving prefix generation and oracle generation.

3) Task 3: Prefix and Oracle Generation: CasModaTest utilizes the selected examples and the target method under test to construct a prompt (i.e., ① + ② + ③ + ④ marked in Fig. 3). Then, CasModaTest uses this prompt to interact with LLMs and help them infer how to generate the corresponding prefix and oracle solution in order. CasModaTest first interacts with LLM through a well-designed prefix prompt and obtains the output: `generated_prefix`. Then, CasModaTest subsequently makes another interaction with LLM through an oracle prompt with previously generated content (i.e., `generated_prefix`) and obtain corresponding output: `generated_oracle`. Notice that, to make a better combination between prefix generation and oracle generation, the variables with the same names in the counterpart layer (i.e., Demonstration Layer and Template Layer) inside a prompt have the same meaning and point to identical content. For example, in the few-shot selection part (i.e., labeled as "Demonstration Layer" in Fig. 3), the variable *demo.test_name* refers to the particular and identical name of a test, which makes a connection between prefix and oracle generation.

B. Verification Phase

This phase aims to verify previously generated separate parts (i.e., prefix and oracle) of a test. To achieve this, we need to address three tasks: (1) Test Case Assembling, (2) Compilation Verification, and (3) Execution Verification.

1) Task 1: Test Case Assembling: A complete test case is the combination of test prefix and test oracle in an appropriate way optionally appending with dependencies fixing. In previous steps, the generated prefix and generated oracle are couples of Java statements even with only one statement. Therefore, we combine the prefix and oracle together and put them into a test method skeleton with the import statements of JUnit framework. Meanwhile, methods under test may be connected with other classes or libraries. It is necessary to address the corresponding dependencies with appropriate importing operations (e.g., `import java.io.File`) or adding annotations (e.g., `@Test`). The process of the whole assembling is illustrated in Fig. 4 for exampling. In Fig. 4, there is a method named `printRecord` under test inside a class named `CSVPrinter` from Apache Commons project. This information is organized according to the prompt template shown in the left of Fig. 4. Then, CasModaTest interacts with LLMs and obtains two generated parts: test prefix and test oracle. Both of them are several Java statements shown in the middle of Fig. 4. Finally, these statements are organized into a testing function (i.e., `testPrintRecord()` annotated with `@Test`) nested in testing driven class (i.e., `CSVPrinterTest`) based on the requirement of JUnit framework and fixing the dependencies by importing a couple of packages.

2) Task 2: Compilation Verification: CasModaTest compiles the candidate test case to verify the correctness. If the compilation is successfully executed (e.g., without syntax errors), CasModaTest will turn to the execution verification stage. Otherwise, CasModaTest collects failing test information, which can aid LLMs in understanding the failure causes and provide guidance to fix compilation errors.

Inspired by mutation-based testing [11], [23]–[25], the generated material may be a good starting point when mutating a good test case even if it may contain a few syntax errors. Thus, we do not stop at the stage when complication meets error but instead interact with LLMs for a fine-tuned one for better efficiency. Then, CasModaTest reconstructs the prompt, appends the failing information (i.e., illustrated in Fig. 5(a)), and feeds it back to the LLM for test case repair. The feedback prompt template involves several aspects: (1) the focal function's signature, (2) the class containing the focal method, (3) the compiled failure errors, and (4) the originally assembled non-compilable test case (i.e., no interaction with LLM) or previously generated non-compilable test case (i.e., after interacting with LLMs). Following that, CasModaTest interacts with LLMs using the new prompt to generate a new candidate test case.

This iterative process continues until a compilable test case is achieved (i.e., successfully compiled by Java) or exceeds the maximum M number of interactions (i.e., four times for a better balance between effectiveness and cost).

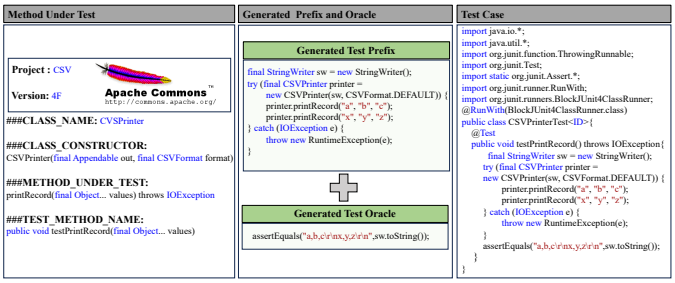

Fig. 4: The process of test case assembling from test prefix and test oracle.

3) Task 3: Execution Verification: Similar to compilation verification, CasModaTest executes the compilable candidate test case to verify functionality correctness.

Compilation Failure Feedback Prompt Template	Execution Failure Feedback Prompt Template

The last test case you generated against
`{data.focal_method_signature}`
in `{data.class_name}` had **some compilation error(s) and compiled stderr** as follows:
`{data.failed_stderr}`.
The original test case is as follows:
```{data.test_unit}```
Please generate a new test case to pass the test based on the above information.
And don't give a test class, a test method is enough.
	The last test case you generated against
	`{data.focal_method_signature}`
	in `{data.class_name}` had **some test failure and stdout** as follows:
	`{data.failed_stderr}`.
	The failed test case is as follows:
	```{data.test_unit}```
	Please generate a new test case to pass the test based on the above information.
	And don't give a test class, a test method is enough.

(a)	(b)

Fig. 5: The design of feedback prompts during fixing compilation and execution errors with LLMs.

If the execution is successful (i.e., free of runtime errors and producing the expected results), we obtain a good test case and the process is terminated. Otherwise, CasModaTest also collects failing test information and reconstructs the prompt, appends the failing information (i.e., illustrated in Fig. 5(b)), and feeds it back to the LLM for test case repair. The feedback prompt template also involves several aspects: (1) the tested method signature, (2) the class containing the focal method, (3) the executed failure errors, and (4) the originally assembled non-executable test case (i.e., no interaction with LLM) or previously generated non-executable test case (i.e., after interacting with LLMs). This iterative process continues until an executable test case is achieved (i.e., successfully executed by Java) or the maximum N number of interactions exceeds (i.e., three times is used for this study since the context window limitation of LLMs).

IV. Experimental Setup

A. Dataset

1) Studied Dataset: We follow prior studies [5]–[7] and use Defects4J [12] as our studied dataset. Specifically, we select five popular and commonly evaluated projects included in the dataset: Apache Common Cli, Apache Common Csv, Google Gson, JFreeChart and Apache Commons Lang. These projects represent different domains, namely cmd-line interface, data processing, serialization, visualization and utility. Table I presents detailed information of our studied projects in the Defects4J dataset.

TABLE I: Defects4J projects on evaluation.

Project	Abbr.	Domain	# Focal Class	# Focal Method (Original)	# Focal Method (Manually Built)
Commons-Cli	Cli	Cmd-line interface	13	645	488
Commons-Csv	Csv	Data processing	5	373	430
Gson	Gson	Serialization	15	224	224
Jfreechart	Chart	Visualization	24	1,318	2,030
Commons-Lang	Lang	Utility	28	2,712	4,535
All			**85**	**5,278**	**7,707**

2) Demonstration Pool: To better serve CasModaTest which is a two-stage approach for unit test generation, we manually separately build two types of demonstration pools. As introduced in Section III-A1, we represent one demonstration with several parts by considering the inherent structure among method, class as well as the structure of the unit test case.

To ensure quality, the demo pools are also built from the unit test cases from the studied datasets, which were written by developers originally. We first download the source code of each project from GitHub with the latest version for full access. Following that, for each test class (i.e., located in the directory of "src/test"), we retrieve the corresponding test class under test according to the naming patterns based on a deeper analysis. For example, the test class for the class "JsonReader" is most likely named "JsonReaderTest". Then, we can identify the source code of the method that is under test according to the name of each test case in the test class. Following that, the signature of the test class's constructor and the signature of the method under test can be obtained. Finally, we separate the unit test to obtain the corresponding test prefix and test oracle and build the corresponding demonstrations.

Notably, when dividing each unit test into two parts (i.e., test prefix and test oracle), we encounter two scenarios: one merely contains a test oracle and the other contains more than one test oracle, especially the mixed types of oracles (i.e., assertion oracle and expected exception oracle). As for the former, we directly extract the test prefix and test oracle and replace the test oracle with a placeholder. As for the latter, taking the mixed scenario as an example, we need to extract the two types of test oracles individually. For the part of the test prefixs, we extract them and optionally merge the test prefix located in `try` statement with the one located in front of the unit test. Therefore, we obtain two test instances of this scenario and also replace the test oracle with a placeholder.

Meanwhile, to avoid data leaks, we exclude demos that have both the same class name and the method signature when building the pools since these demos may be opted as query examples for the few-shots. Eventually, we obtain 85 focal classes and 7,707 focal methods as shown in the last column of Table I. Notice that, such a demonstration pool only needs to be built once and can be applicable to other scenarios with the same programming language.

B. Baselines

To investigate the effectiveness of CasModaTest, we consider five baselines for a comprehensive comparison.

- **AthenaTest** [5] is the first approach to formulate unit test cases as a sequence-to-sequence learning task. Compared to previous methods, AthenaTest's main contribution is to take the understandability of test cases into consideration rather than solely focusing on code coverage.
- **A3Test** [6] also treats test generation as a sequence-to-sequence task. It fine-tunes PLBART [26] specifically for the assertion generation task and incorporates validation mechanisms for naming conventions.
- **ChatUniTest** [7] is the first automatic unit test generation method based on ChatGPT by leveraging the capabilities of large language models to follow natural language instructions. It can perform both verification and repair of test results after test generation.
- **CAT-LM** [27] is a GPT-style aligned code and tests language model with 2.7 billion parameters, trained on a corpus of Python and Java projects. Its key feature is a novel pretraining signal that considers code-test mappings when available.

- **EvoSuite** [4] is a fully automated JUnit test suite generator for Java classes and targets code coverage criteria by deriving test suites with an evolutionary approach.

C. Evaluation Metrics

We adopt three widely used metrics to evaluate the performance of CasModaTest and the baseline approaches.

- **Accuracy**. It represents the percentage of correct tests in all generated tests. A test case is regarded as correct if and only if it passes and invokes the method under test directly or indirectly.
- **Focal Method Coverage**. It represents the percentage of focal methods that are correctly tested by the generated unit tests. If there is at least one correct attempt in all the generated test cases that tests the focal method correctly, then the method is considered covered.
- **Code Coverage.** It represents the percent of executed line/method/branch. We utilize the built-in code coverage analysis tool in IntelliJ IDEA to calculate the proportion of code covered by all the successfully executed test cases generated by each approach in a specific run.

D. Implementation Details

We implemented CasModaTest in Python by wrapping the large language models' ability through the API support (e.g., ChatGPT [28]) or source code availability (e.g., CodeLlama [29]) and adhere to the best-practice guide [30] for each prompt. We utilize the *gpt-3.5-turbo* model from the ChatGPT family by default, which is the version used uniformly for our experiments for better generation efficiency. We employ five few-shot examples for prefix generation and oracle generation and set the maximum interaction number for compilation failures to 3 and for runtime failures to 2 during validation. We also consider several other types of LLMs (e.g., GPT-4) for demonstrating the generalization of our model, especially the open-source ones: DeepSeek-Coder [31], CodeLlama [29] and its variant [32].

For baselines, the experimental results on the prior study of Xie et al. [7] are directly adopted for the first three methods due to consistent benchmarks. For CAT-LM, a replication is conducted. The replication process involves initially locating the model [33] and data processing code [34]. By combining the content of the paper with the outcomes of executing the data processing code, the appropriate prompts to be provided to CAT-LM are determined. Considering CAT-LM's approach to generate code at the granularity of entire classes for generating test classes, whereas Defects4j requires the generation of test cases for individual methods, other public methods in the class under test are removed from the code repository for each query. For the evaluation of results, a method is employed where test cases are generated ten times for each query, all test cases are aggregated, and the generation is considered successful if at least one correct test case exists among them.

V. Results

We present the results by proposing and answering the following three research questions:

- **RQ1 Effectiveness Comparison**. How is the performance of CasModaTest compared to state-of-the-art (SOTA) approaches?
- **RQ2 Cascade and Model-agnostic**. How is the performance of CasModaTest when generating in a cascaded manner compared to non-staged generation using the same model? Does the efficacy of CasModaTest hinge on particular large language models?
- **RQ3 Order of Demonstrations**. How does the order of demonstrations inside a prompt affect the performance of CasModaTest?

A. RQ1: Effectiveness Comparison

TABLE II: Accuracy of CasModaTest compared with the baselines (RQ1).

Projects	AthenaTest	A3Test	ChatUniTest	CAT-LM	CasModaTest
Cli	11.07%	25.19%	38.77%	45.60%	**58.29%**
Csv	8.98%	25.73%	44.26%	43.66%	**69.17%**
Gson	2.89%	14.09%	23.3%	24.17%	**63.18%**
Chart	11.7%	31.3%	39.02%	37.84%	**76.66%**
Lang	23.35%	49.5%	39.85%	56.15%	**84.13%**
Total	16.21%	40.05%	40.41%	48.04%	**77.16%**

Motivation. In this RQ, we aim to investigate the overall performance of CasModaTest on unit test generation by comparing the results with four baseline approaches discussed in Section IV. These baselines are recently proposed and have good performance in unit test generation. We consider two widely used performance metrics (i.e., accuracy and focal method coverage) which are commonly adopted by these SOTA methods. Besides, to the best of our knowledge, EvoSuite has the best performance in terms of code coverage though its generated unit test is not friendly to developers. Thus, we particularly want to compare CasModaTest with EvoSuite to figure out the code coverage differences.

Approach. We use the datasets discussed in Section IV to conduct the experiments using CasModaTest and compare the accuracy with the baselines. For each focal method, we extract its corresponding class name, constructor signature, and method signature from the datasets to generate unit tests with CasModaTest. We use *gpt-3.5-turbo* as the backbone model of CasModaTest. To compare with EvoSuite, we adopt the runnable package with default parameters and generate corresponding unit tests for the studied Java project. We follow previous work [6] to select the most representative classes that not only cover the main functional modules of a particular project but also cover the frequently modified classes in their change history. In addition, to avoid data leakage for "gpt-3.5-turbo", we consider another java project (i.e., "the binance-connector-java v3.1.0"), which is an unseen project to *gpt-3.5-turbo*. In addition, to mitigating data leakage for *gpt-3.5-turbo*, we consider another Java project (i.e., *the binance-connector-java v3.1.0*). This project's first release was after September 2021, ensuring it is definitely unseen by *gpt-3.5-turbo*.

TABLE III: Focal method coverage of CasModaTest compared with the baselines (RQ1).

Projects	AthenaTest	A3Test	ChatUniTest	CAT-LM	CasModaTest
Cli	29.46%	37.2%	**70.34%**	59.22%	61.98%
Csv	34.31%	37.8%	76.94%	66.76%	**77.42%**
Gson	9.54%	40.9%	55.91%	47.27%	**63.35%**
Chart	32.00%	34.40%	79.56%	49.25%	**82.55%**
Lang	56.97%	58.30%	84.05%	70.06%	**88.49%**
Total	43.75%	46.80%	79.67%	62.32%	**81.93%**

Results. Overall, **we find that CasFlowTester outperforms the baseline approaches in terms of accuracy, focal method coverage, and code coverage.** Below, we discuss the specific experimental results for each metric.

Accuracy. Table II shows the percentage of test cases that are correctly generated on five projects (i.e., accuracy). Following our baselines, we employ a weighted mean approach based on the number of queries provided by the five projects to calculate the average. Subsequent average calculations also follow this way. The best result among different approaches is marked in bold. Overall, CasModaTest achieves the accuracy (i.e., 77.16% on average), while the average accuracy of the baselines ranges from 16.21% for `AthenaTest` to 48.04% for `CAT-LM`. Upon examining some of the test methods successfully generated by CasModaTest, which other baselines failed to accomplish, it is found that generating the test prefix independently allows the LLM to better handle the instantiation of classes under different design patterns.

Focal Method Coverage. Table III presents the results of focal method coverage for CasModaTest and the four baselines. Notably, the performance of CasModaTest is obtained with a single round with LLM, which means CasModaTest's pipeline is executed only once for each item in the dataset. In contrast, ChatUniTest's performance is a cumulative result of six rounds, and the other two baselines' results are also the accumulation of several dozen rounds. Overall, CasModaTest outperforms all the baselines on four out of the five studied projects (i.e., Csv, Gson, Chart and Lang), with an average focal method coverage of 81.93%. The average focal method coverage of the four baselines ranges from 43.75% for `AthenaTest` to 79.67% for `ChatUniTest`.

Although accuracy and focal method coverage generally show a positive correlation, some results deviate from this pattern. Despite CAT-LM performing ten generations for each focal method coverage, the final set of all correct test cases does not stand out in terms of focal method coverage. It is speculated that this is because CAT-LM, as a language model with a smaller parameter size that has been trained on five test projects, tends to overfit, which is not conducive to generating more diversified test cases.

As for `Cli` project, ChatUniTest performs better than CasModaTest. Through deep analysis, we find that ChatUniTest generates a test suite (usually containing several unit tests) for one focal method, while CasModaTest just generates one unit test for a particular focal method. As an example shown in Fig. 6, CasModaTest generates only unit test for the target method but ChatUniTest generates two. Therefore, it brings a higher possibility of covering the focal method to ChatUniTest.

Test Case Generated By CasModaTest$_{gpt3.5}$

```
@Test(timeout = 4000)
public void testFlush() {
    StringWriter stringWriter = new StringWriter();
    JsonWriter writer = new JsonWriter(stringWriter);
    try {
        writer.beginObject();
        writer.name("name").value("John");
        writer.name("age").value(30);
        writer.endObject();
        writer.flush();
        String output = stringWriter.toString();
        String expectedOutput =
        "{\"name\":\"John\",\"age\":30}";
        assertThat(output).
                isEqualTo(expectedOutput);
    } catch (IOException e) {
        fail("Exception occurred: "
                + e.getMessage());
    }
}
```

Test Case Generated By ChatUniTest

```
@Mock
private Writer out;
private JsonWriter jsonWriter;
@BeforeEach
void setUp() {
    MockitoAnnotations.initMocks(this);
    jsonWriter = new JsonWriter(out);
}
@Test
@Timeout(8000)
void flush_shouldFlushWriter() throws IOException {
    // Arrange and Act
    jsonWriter.flush();
    // Assert
    verify(out).flush();
}
@Test
@Timeout(8000)
void flush_shouldThrowIllegalStateException() throws IOException {
    // Arrange
    jsonWriter.close();
    // Act and Assert
    assertThrows(IllegalStateException.class, () -> jsonWriter.flush());
}
```

Fig. 6: The Test Cases Generated by CasModaTest and ChatUniTest for method flush().

TABLE IV: Code coverage of CasModaTest compared with EvoSuite (RQ1).

Projects	# Class	# Method	CasModaTest	EvoSuite	CasModaTest	EvoSuite	CasModaTest	EvoSuite
			Method Coverage		Line Coverage		Branch Coverage	
Cli	3	75	82.54%	97.24%	56.47%	81.77%	42.88%	70.32%
Csv	4	69	86.29%	86.05%	56.70%	76.33%	40.08%	78.95%
Gson	5	85	88.98%	93.51%	64.88%	83.66%	46.23%	53.60%
Chart	11	764	91.03%	92.01%	56.92%	65.30%	44.65%	49.42%
Lang	15	368	87.40%	96.77%	65.75%	78.69%	47.16%	57.41%
Binance	8	264	89.87%	0	81.43%	0	74.15%	0
Average	46	1,625	**88.46%**	77.55%	**64.88%**	62.34%	**50.26%**	47.82%

Code Coverage. Table IV presents the results of both CasModaTest and EvoSuite in terms of code coverage. CasModaTest generates test cases for each method six times, and we present the summarized results of executable case coverage. We use command-line execution with default parameter configurations for generating test cases using EvoSuite. Due to compatibility issues with the JDK version, EvoSuite fails to generate test cases for the Binance project. Ultimately, CasModaTest outperforms EvoSuite with a slight advantage in method coverage(12.33%), line coverage(3.91%), and branch coverage(4.86%). The experimental results show that EvoSuite exhibits excellent line coverage and branch coverage for methods where it can generate test cases. However, it slightly underperforms compared to CasModaTest in terms of coverage of the tested methods. Therefore, overall coverage is better with CasModaTest.

Summary of RQ1: *CasModaTest achieves an accuracy of 77.16% and a focal method coverage of 81.93% on average, which outperforms the four baselines. Regarding code coverage, CasModaTest achieved 88.46% method coverage, 64.88% line coverage, and 50.26% branch coverage, surpassing EvoSuite, which aims to maximize coverage.*

B. RQ2: Cascade and Model-agnostic

Motivation. In RQ1, we use *gpt-3.5-turbo* as the backbone model to study the effectiveness of CasModaTest compared

with the baselines. The results show that CasModaTest outperforms existing baselines and achieves promising performance for unit test case generation. In this RQ, we further study the effectiveness of CasModaTest from two aspects: **Cascade** and **Model-agnostic**. For the aspect of Cascade, we study the effectiveness of CasModaTest's cascaded pipeline to the general pipeline of directly generating the complete unit test cases. For Model-agnostic, we study if CasModaTest is effective when using different models.

Approach. We design different experiments to study the aspects of cascade and model-agnostic of CasModaTest, respectively.
Cascade. We use *gpt-3.5-turbo* as the backbone model to compare the results of CasModaTest's cascaded pipeline (i.e., generate test prefix and test oracle in order) with the direct pipeline (i.e., generate the complete unit test case directly). Specifically, for the cascaded pipeline, we follow the same setting of RQ1 in using CasModaTest. For the direct pipeline, we generate the complete unit test case directly by replacing the part of the test prefix (i.e., as shown in Fig. 3) with the complete body of the unit test case.
Model-agnostic. Apart from *gpt-3.5-turbo*, we use additional state-of-the-art open-source LLMs to investigate if CasModaTest is still effective when using other models (i.e., model-agnostic). Specifically, the additional models include *gpt-4.0* and three open-source LLMs: (1) *CodeLlama-Instruct* [35], (2) *Phind-CodeLlama-Instruct* [32], and (3) *DeepSeek-Coder-Instruct* [31]. We follow the experimental setup discussed in Section IV and compare the results of each model with the baselines. When conducting experiments using backbone models, we ensure that the demo selection technique, cascaded strategy, and the number of fixes in the feedback stage remain consistent.

TABLE V: Results of cascaded and direct pipelines using CasModaTest $_{GPT3.5}$ (RQ2).

Projects	# Query	% Accuracy		% Focal Method Coverage	
		Cascaded	Direct	Cascaded	Direct
Cli	645	**58.29%**	51.32%	**61.98%**	53.02%
Csv	373	**69.17%**	61.93%	**77.42%**	69.44%
Gson	220	**63.18%**	50.91%	**63.35%**	53.18%
Chart	1,328	**76.66%**	69.20%	**82.55%**	75.15%
Lang	2,712	**84.13%**	73.64%	**88.49%**	78.47%
All	5,278	**77.16%**	68.02%	**81.93%**	72.83%

Results. We discuss the results from the aspects of Cascade and Model-agnostic, respectively.
Cascade. Table V presents the results of unit test generation using CasModaTest's cascaded pipeline and the direct pipeline. Overall, **we find CasModaTest's cascaded pipeline achieves higher accuracy and focal method coverage in all the five studied projects.** For example, in the Cli project, the accuracy of cascaded and direct pipelines are 58.29% and 51.32% respectively. The focal method coverage of cascaded and direct pipelines are 61.98% and 53.02%, respectively. In the Gson project, we observe the largest difference between the results of cascaded and direct pipelines among the five projects, with the cascaded pipeline outperforming the direct pipeline by 24.10% in accuracy and 19.12% in focal method coverage.
Model-agnostic. Table VI and Table VII present the results of CasModaTest in using different backbone models in terms of accuracy and focal method coverage. The numbers that are higher than all baselines are marked in bold. Overall, **we find that CasModaTest can mostly outperform all the four baselines when using different backbone models.** For accuracy on average, the results of CasModaTest in using different models range from 42.29% of CasModaTest $_{cl}$ to 85.75% of CasModaTest $_{GPT4.0}$, which outperform three of the four baselines(i.e., except for CAT-LM). For focal method coverage on average, the results range from 64.72% of CasModaTest $_{cl}$ to 91.85% of CasModaTest $_{GPT4.0}$, which outperform three of the four baselines (i.e., except for ChatUniTest). Among the three open-source models, *DeepSeek-Coder-Instruct* achieves the best overall accuracy (i.e., 67.17% on average) and *Phind-CodeLlama-Instruct* achieves the best overall focal method coverage (i.e., 87.04% on average).

Summary of RQ2: *CasModaTest's cascaded pipeline outperforms the direct pipeline by a large margin. Besides, CasModaTest can also achieve promising results when using additional open-source LLMs, which shows the capability of model-agnostic in unit test generation.*

C. RQ3: Impacts of Demonstration Sorting

Motivation. Prior studies investigated the construction of in-context examples in the prompts and observed that the order of examples in the prompts can have a significant impact on the results [36]. In this RQ, we study the impact of the order of demonstrations in the few-shot examples on CasModaTest's performance of unit test generation.

Approach. As discussed in Section III-B3, CasModaTest retrieves similar examples based on the cosine similarity between the UniXCoder-encoded vectors of the query and demonstrations as the few-shot examples. We cluster all the examples in the demonstration pool into five clusters and select one sample from each of the clusters. We use three different strategies to sort the five selected examples based on their cosine similarity when composing the few-shot examples in the prompt: *Random, Ascending, and Descending*. We use *gpt-3.5-turbo* as the backbone model to study the results when using different sorting strategies. Additionally, we consider another setting where samples are obtained completely randomly, regardless the similarity.

Results. Table VIII shows the results of unit test generation using three sampling orders (Random, Ascending, and Descending) and the completely random extraction way. Apart from accuracy and focal method coverage, we also discuss Average Repair Attempts, which are the times of attempts required by the LLM to repair the test cases in the verification stage.
Accuracy. Across all the studied projects, the Random sampling order demonstrates a small advantage, with three projects (Cli, Gson and Lang) achieving the highest accuracy. While the

TABLE VI: Comparison of accuracy between different models (RQ2).

Projects	# Query	% Accuracy								
		CasModaTest $_{GPT4.0}$	CasModaTest $_{GPT3.5}$	CasModaTest $_{ds}$	CasModaTest $_{phind-cl}$	CasModaTest $_{cl}$	ChatUniTest	A3Test	AthenaTest	CAT-LM
Cli	645	**66.82%**	**58.29%**	35.00%	27.47%	16.43%	38.77%	25.19%	11.07%	45.60%
Csv	373	**76.68%**	**69.17%**	**51.20%**	**54.95%**	37.00%	44.26%	25.73%	8.98%	43.66%
Gson	220	**95.45%**	**63.18%**	**45.00%**	**30.85%**	**24.09%**	23.30%	14.09%	2.89%	24.17%
Chart	1,328	**87.80%**	**76.66%**	**72.10%**	**44.37%**	**40.59%**	39.02%	31.30%	11.70%	37.84%
Lang	2,712	**89.71%**	**84.13%**	**76.40%**	**60.32%**	**51.47%**	39.85%	49.50%	23.35%	56.15%
Total	5,278	**85.75%**	**77.16%**	**67.17%**	**50.68%**	**42.29%**	40.41%	40.05%	16.21%	48.04%

TABLE VII: Comparison of focal method coverage between different models (RQ2).

Projects	# Query	% Focal Method Coverage								
		CasModaTest $_{GPT4.0}$	CasModaTest $_{GPT3.5}$	CasModaTest $_{ds}$	CasModaTest $_{phind-cl}$	CasModaTest $_{cl}$	ChatUniTest	A3Test	AthenaTest	CAT-LM
Cli	645	**75.97%**	61.98%	55.35%	58.76%	39.38%	70.34%	37.20%	29.46%	59.22%
Csv	373	**87.40%**	**77.42%**	75.34%	**92.23%**	51.74%	76.94%	37.80%	34.31%	66.76%
Gson	220	**95.45%**	**63.35%**	**59.55%**	**72.27%**	40.45%	55.91%	40.90%	9.54%	47.27%
Chart	1,328	**94.95%**	**82.55%**	**87.35%**	**87.65%**	55.65%	79.56%	34.40%	32.00%	49.25%
Lang	2,712	**94.43%**	**88.49%**	**93.69%**	**93.95%**	78.95%	84.05%	58.30%	56.97%	70.06%
Total	5,278	**91.85%**	**81.93%**	**84.69%**	**87.04%**	64.72%	79.67%	46.80%	43.75%	62.32%

TABLE VIII: Comparison between different orders of demos (RQ3).

Projects	# Query	% Accuracy				% Focal Method Coverage				# Average Repair Attempts			
		Random	Ascending	Descending	Totally Random	Random	Ascending	Descending	Totally Random	Random	Ascending	Descending	Totally Random
Cli	645	**58.29%**	53.02%	55.04%	46.05%	**61.98%**	57.83%	61.40%	50.23%	1.237	**1.229**	1.234	1.327
Csv	373	69.17%	**69.44%**	68.63%	64.08%	77.42%	**82.31%**	76.94%	75.60%	**1.011**	1.091	1.172	1.196
Gson	220	**63.18%**	62.73%	59.09%	59.55%	63.35%	**63.64%**	61.82%	60.91%	**1.814**	1.982	1.905	1.941
Chart	1328	76.66%	76.51%	**80.05%**	75.15%	82.55%	83.28%	**84.56%**	78.61%	0.672	0.666	**0.622**	0.697
Lang	2712	**84.13%**	84.07%	83.63%	79.13%	88.49%	**88.68%**	86.58%	83.15%	0.501	**0.481**	0.489	0.537
All	5278	**77.16%**	76.45%	77.15%	72.21%	81.93%	**82.06%**	81.28%	76.52%	0.725	0.725	**0.721**	0.779

Ascending order closely follows the Random order in terms of accuracy, the Descending order exhibits a notable performance in the Chart project, reaching 80.05%. Overall, the Random sorting strategy slightly outperforms the other two with an average accuracy of 77.16%. The accuracy obtained with the totally random order is the lowest, being on average 6.12% lower than the first three arrangements that underwent similarity filtering. This indicates that choosing demos as similar as possible to the query is very important when guiding the LLM in test generation.

Focal Method Coverage. Contrary to the results for accuracy, the highest focal method coverage is achieved when demos in the context are sorted in ascending order. It is due to the fact that: when demos closer to the query have higher similarity, the LLM is more likely to invoke methods that share the same name but have different parameters as the method under test, and these methods happen to be included in the benchmark, thereby increasing the focal method coverage.

Average Repair Times. The differences in repair attempts across various projects are pronounced. For the projects Cli, Csv, and Gson, the average attempts for all four sorting methods exceed one attempt, whereas for the Chart and Lang projects, the average attempts for all sorting methods are below 0.7. This metric indirectly reflects the varying difficulties in generating executable test cases that pass across different projects. The first three sorting strategies, which have undergone similarity filtering, perform similarly, whereas the totally random sorting strategy clearly requires more repair attempts.

Summary of RQ3*: CasModaTest is not sensitive to the order of cases after clustering and filtering out examples with high similarity.*

VI. Discussion

A. The impact of demo selection strategy

We conducted preliminary experiments using *gpt-3.5-turbo*, adjusting the number of demos from 0 to 10, and tested on a randomly selected 10% subset of the benchmark dataset. We found that when the number of demos was too small (less than 5), the model tends to overfit to the demo(s), resulting in a decreased compilation success rate. Conversely, when the number of demos was too large, the model tended to produce nearly repetitive test predictions, leading to a decreased success rate. We attempted a similarity retrieval strategy based on BM25. On the randomly selected 10% subset of the benchmark dataset, with all other experimental settings held constant, the BM25-based retrieval strategy yielded results 15.7% lower than the embedding-based strategy. We hypothesize that using embeddings better captures the semantic relationships between code, thus yielding superior results.

B. Cross-project Demo Selection

We conduct a cross-project demo selection study to show the generalization of CasModaTest. 20% of the data from both the `Csv` and `Gson` projects are extracted, and they swap demo pools for retrieval and generation. Similarly, 10% of the data from both the `Lang` and `Chart` projects are extracted for the

TABLE IX: The Results of cross-project demo selection strategy (Discussion).

Target	Demo	% Size	# Query	Standard Acc.	Sample Acc.	Decline	Standard fmc.	Sample fmc.	Decline
csv	gson		75	69.17%	57.33%	-17.12%	77.42%	61.33%	-20.78%
gson	csv	20%	43	63.18%	55.81%	-11.67%	63.35%	58.14%	-8.22%
chart	lang		132	76.66%	67.67%	-11.73%	82.55%	70.68%	-14.38%
lang	chart	10%	272	84.13%	76.84%	-8.67%	88.49%	84.93%	-4.02%
Average			522	78.37%	69.99%	-10.69%	83.33%	75.73%	-9.12%

same experiment. Table IX shows that retrieving demos from other projects in the stage of generation may lead to a certain decrease in both accuracy and coverage metrics. However, such results still represent the highest accuracy and the second-highest ranking in coverage compared to baselines.

C. Threats of Validity

Internal Validity. The first one comes from potential data leakage since the referenced unit test written by developers may be part of the training data of LLMs. To prevent data leakage issues, this paper specifically incorporates the Binacce dataset, which remains unseen for *gpt-3.5-turbo*. The second one arises from potential errors in implementing our approach and baselines. To mitigate implementation errors, we develop our model via pair programming and use original baseline code (with identical settings) provided by the authors. All experimental scripts are rigorously reviewed.

External Validity. The main external threat to validity comes from our evaluation datasets. The effectiveness observed in CasModaTest's performance may not be applicable across different datasets, especially for unit tests written in other programming languages (e.g., C/C++, Python, etc).

VII. Related Work

A. Automated Testing

The end-to-end unit testing solutions can be divided into two types: traditional approaches and large language model (LLM) based approaches. Traditional approaches include strategies based on search, randomization, or constraints-based, with the goal of maximizing test coverage (e.g., Evosuite [4] and Randoop [3]. However, these approaches have limited interpretability, especially the method's names and variables, which are hard to understand. Recently, large language models have achieved great success and can also be applied to software testing scenarios. They can be further divided into three subgroups: approaches based on pre-training and fine-tuning paradigms (e.g., AthenaTest [5], A3Test [6], and CAL-LM [27]), prompt-based testing generation (e.g., ChatTester [14], ChatUniTest [7], and AUGER [37]), and approaches combining static analysis techniques with LLMs (e.g., CodaMosa [38], RATester [39], and HITS [40]).

B. Prompt Engineering

Prompt engineering refers to the methods of interacting with LLMs to guide their behavior and achieve desired outcomes without the need for updating the model weights [41]–[43]. The few-shot learning [15], [44] and the instruction tuning [45]–[47] are the most effective ways in prompt engineering. Many studies [48]–[50] have explored how to select demonstrations to optimize performance and have observed that the choice of prompt format, the selection of examples of demonstrations, the number of the demonstrations, and the order of examples can lead to significant differences in performance. Providing examples is a way of implicitly describing tasks, while instruction tuning can directly convey human intent to LLMs. Instructing models with natural language commands can guide LLMs as precisely as possible. The studies [48]–[50] underscore the pivotal role of strategically selecting in-context examples and address the challenges of in-context and few-shot learning in large language models like GPT-3, highlighting the importance of example choice to enhance model generalization and consistency across diverse tasks.

VIII. Conclusion

This paper proposes a cascaded, model-agnostic, and end-to-end unit test generation framework. CasModaTest first treats the unit test generation task as two cascaded tasks: test prefix generation and test oracle generation. Then, to better stimulate models' learning ability, this paper manually builds large-scale demo pools to provide CasModaTest with high-quality test prefixes and test oracles examples. Furthermore, the paper also implements a static analysis-based approach for context extraction and construction. Finally, CasModaTest automatically assembles the generated test prefixes and test oracles and compiles or executes them to check their effectiveness, optionally appending with several attempts to fix the errors outputted by the JUnit and Mockito frameworks. The experimental results show the effectiveness and priority of CasModaTest over the studied four baselines with three common metrics.

Acknowledgements

This work was supported by Zhejiang Pioneer (Jianbing) Project (2025C01198(SD2)), the National Natural Science Foundation of China (Grant No.62202419), the Fundamental Research Funds for the Central Universities (No. 226-2022-00064), Zhejiang Provincial Natural Science Foundation of China (No. LY24F020008), the Ningbo Natural Science Foundation (No. 2022J184), the Key Research and Development Program of Zhejiang Province (No.2021C01105), the State Street Zhejiang University Technology Center, and Douyin Co., Ltd.

References

[1] V. Garousi and J. Zhi, "A survey of software testing practices in canada," *Journal of Systems and Software*, vol. 86, no. 5, pp. 1354–1376, 2013.

[2] J. Lee, S. Kang, and D. Lee, "Survey on software testing practices," *IET software*, vol. 6, no. 3, pp. 275–282, 2012.

[3] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 815–816.

[4] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[5] 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.

[6] S. Alagarsamy, C. Tantithamthavorn, and A. Aleti, "A3test: Assertion-augmented automated test case generation," *arXiv preprint arXiv:2302.10352*, 2023.

[7] Z. Xie, Y. Chen, C. Zhi, S. Deng, and J. Yin, "Chatunitest: a chatgpt-based automated unit test generation tool," *arXiv preprint arXiv:2305.04764*, 2023.

[8] W. Sun, H. Li, M. Yan, Y. Lei, and H. Zhang, "Revisiting and improving retrieval-augmented deep assertion generation," *arXiv preprint arXiv:2309.10264*, 2023.

[9] Y. Zhang, Z. Jin, Z. Wang, Y. Xing, and G. Li, "Saga: Summarization-guided assert statement generation," *arXiv preprint arXiv:2305.14808*, 2023.

[10] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "Toga: A neural method for test oracle generation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2130–2141.

[11] A. Moradi Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, "Effective test generation using pre-trained large language models and mutation testing," *arXiv e-prints*, pp. arXiv–2308, 2023.

[12] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 437–440.

[13] Apache, "Apache commons lang," https://commons.apache.org/proper/commons-lang/, 2023.

[14] 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.

[15] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[16] I. Beltagy, A. Cohan, R. Logan IV, S. Min, and S. Singh, "Zero-and few-shot nlp with pretrained language models," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics: Tutorial Abstracts*, 2022, pp. 32–37.

[17] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with chatgpt," *arXiv preprint arXiv:2302.11382*, 2023.

[18] S. Feng and C. Chen, "Prompting is all your need: Automated android bug replay with large language models," *arXiv preprint arXiv:2306.01987*, 2023.

[19] Z. Zhang, A. Zhang, M. Li, and A. Smola, "Automatic chain of thought prompting in large language models," *arXiv preprint arXiv:2210.03493*, 2022.

[20] X. Li and X. Qiu, "Mot: Pre-thinking and recalling enable chatgpt to self-improve with memory-of-thoughts," *arXiv preprint arXiv:2305.05181*, 2023.

[21] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[22] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*, 2023.

[23] K. Jain, U. Alon, A. Groce, and C. L. Goues, "Contextual predictive mutation testing," *arXiv preprint arXiv:2309.02389*, 2023.

[24] K. Zhang, X. Zhu, X. Xi, M. Xue, C. Zhang, and S. Wen, "Shapfuzz: Efficient fuzzing via shapley-guided byte selection," *arXiv preprint arXiv:2308.09239*, 2023.

[25] N. Louloudakis, P. Gibson, J. Cano, and A. Rajan, "Mutatenn: Mutation testing of image recognition models deployed on hardware accelerators," *arXiv preprint arXiv:2306.01697*, 2023.

[26] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

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

[28] chatgptendpoint, "Introducing chatgpt and whisper apis," https://openai.com/blog/introducing-chatgpt-and-whisper-apis, 2023.

[29] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[30] J. Shieh, "Best practices for prompt engineering with openai api," *OpenAI, February https://help.openai. com/en/articles/6654000-best-practices-for-prompt-engineering-with-openai-api*, 2023.

[31] "DeepSeek-Coder," 2023. [Online]. Available: https://github.com/deepseek-ai/DeepSeek-Coder

[32] "Phind/Phind-CodeLlama-34B-v2," 2023. [Online]. Available: https://huggingface.co/Phind/Phind-CodeLlama-34B-v2

[33] "nikitharao/catlm," 2023. [Online]. Available: https://huggingface.co/nikitharao/catlm/tree/main

[34] "Zenodohome/catlm," 2023. [Online]. Available: https://zenodo.org/records/7909299

[35] "codellama/CodeLlama-34b-Instruct-hf," 2023. [Online]. Available: https://huggingface.co/codellama/CodeLlama-34b-Instruct-hf

[36] Y. Lu, M. Bartolo, A. Moore, S. Riedel, and P. Stenetorp, "Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity," *arXiv preprint arXiv:2104.08786*, 2021.

[37] X. Yin, C. Ni, X. Xu, and X. Yang, "What you see is what you get: Attention-based self-guided automatic unit test generation," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2025, pp. 1039–1051.

[38] 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 *International conference on software engineering (ICSE)*, 2023.

[39] X. Yin, C. Ni, X. Li, L. Chen, G. Ma, and X. Yang, "Enhancing llm's ability to generate more repository-aware unit tests through precise contextual information injection," *arXiv preprint arXiv:2501.07425*, 2025.

[40] Z. Wang, K. Liu, G. Li, and Z. Jin, "Hits: High-coverage llm-based unit test generation via method slicing," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1258–1268.

[41] X. Han, Z. Zhang, N. Ding, Y. Gu, X. Liu, Y. Huo, J. Qiu, Y. Yao, A. Zhang, L. Zhang *et al.*, "Pre-trained models: Past, present and future," *AI Open*, vol. 2, pp. 225–250, 2021.

[42] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[43] X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng, and X. Yang, "Thinkrepair: Self-directed automated program repair," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1274–1286.

[44] X. Yin, C. Ni, and S. Wang, "Multitask-based evaluation of open-source llm on software vulnerability," *IEEE Transactions on Software Engineering*, 2024.

[45] S. Zhang, L. Dong, X. Li, S. Zhang, X. Sun, S. Wang, J. Li, R. Hu, T. Zhang, F. Wu *et al.*, "Instruction tuning for large language models: A survey," *arXiv preprint arXiv:2308.10792*, 2023.

[46] L. Reynolds and K. McDonell, "Prompt programming for large language models: Beyond the few-shot paradigm," in *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–7.

[47] S. Mishra, D. Khashabi, C. Baral, and H. Hajishirzi, "Natural instructions: Benchmarking generalization to new tasks from natural language instructions," *arXiv preprint arXiv:2104.08773*, pp. 839–849, 2021.

[48] S. M. Xie, A. Raghunathan, P. Liang, and T. Ma, "An explanation of in-context learning as implicit bayesian inference," *arXiv preprint arXiv:2111.02080*, 2021.

[49] J. Liu, D. Shen, Y. Zhang, B. Dolan, L. Carin, and W. Chen, "What makes good in-context examples for gpt-3?" *arXiv preprint arXiv:2101.06804*, 2021.

[50] J. Lu, P. Gong, J. Ye, J. Zhang, and C. Zhang, "A survey on machine learning from few samples," *Pattern Recognition*, vol. 139, p. 109480, 2023.