Clarifying Semantics of In-Context Examples for Unit Test Generation

Chen Yang[§], Lin Yang[§], Ziqi Wang[§], Dong Wang[§], Jianyi Zhou[†], Junjie Chen[§*]
[§]College of Intelligence and Computing, Tianjin University, Tianjin, China
[†]Huawei Cloud Computing Technologies Co., Ltd., Beijing, China
{yangchenyc, linyang, wangziqi123, dong_w, junjiechen}@tju.edu.cn, zhoujianyi2@huawei.com

***Abstract*—Recent advances in large language models (LLMs) have enabled promising performance in unit test generation through in-context learning (ICL). However, the quality of in-context examples significantly influences the effectiveness of generated tests—poorly structured or semantically unclear test examples often lead to suboptimal outputs. In this paper, we propose CLAST, a novel technique that systematically refines unit tests to improve their semantic clarity, thereby enhancing their utility as in-context examples. The approach decomposes complex tests into logically clearer ones and improves semantic clarity through a combination of program analysis and LLM-based rewriting. We evaluated CLAST on four open-source and three industrial projects. The results demonstrate that CLAST largely outperforms UTgen, the state-of-the-art refinement technique, in both preserving test effectiveness and enhancing semantic clarity. Specifically, CLAST fully retains the original effectiveness of unit tests, while UTgen reduces compilation success rate (CSR), pass rate (PR), test coverage (Cov), and mutation score (MS) by an average of 12.90%, 35.82%, 4.65%, and 5.07%, respectively. Over 85.33% of participants in our user study preferred the semantic clarity of CLAST-refined tests. Notably, incorporating CLAST-refined tests as examples effectively improves ICL-based unit test generation approaches such as RAGGen and TELPA, resulting in an average increase of 25.97% in CSR, 28.22% in PR, and 45.99% in Cov for generated tests, compared to incorporating UTgen-refined tests. The insights from the follow-up user study not only reinforce CLAST's potential impact in software testing practice but also illuminate avenues for future research.**

***Index Terms*—Test Refinement, Unit Test Generation, In-Context Learning**

I. Introduction

Automated unit test generation is vital for enhancing software quality by producing tests to verify individual components. While search-based approaches that apply heuristic optimization methods to explore the test space have been extensively studied over the years [1]–[3], recent advances in large language models (LLMs) offer a new paradigm. By learning from vast code repositories, LLMs can infer semantic relationships between code and its corresponding tests, enabling the generation of more context-aware unit tests that address the limitations of traditional methods and enhance overall test effectiveness.

In-Context Learning (ICL) has emerged as a key technique for harnessing LLMs' inference capabilities in automated test generation. Methods such as Retrieval-Augmented Generation (RAG) and few-shot learning enable LLMs to adapt to specific tasks using in-context test examples, producing more syntactically valid tests with improved coverage while eliminating the need for resource-intensive fine-tuning. Prior work (i.e., RAGGen [4] and TELPA [5]) has demonstrated this potential. For instance, RAGGen retrieves unit tests corresponding to the methods similar to the focal method (i.e., method under test) as examples. TELPA selects a set of unit tests generated by a search-based approach as counter-examples to guide LLMs in generating more diverse tests. However, the effectiveness of these ICL-based approaches critically depends on the semantic clarity of the provided test examples. Specifically, semantic clarity refers to how clearly a unit test conveys its purpose and behavior, comprising two aspects: (i) logical clarity: whether the test targets a single, well-defined scenario, as mixing unrelated assertions often leads to complex logic that obscures interpretability; and (ii) textual clarity: whether identifiers and code comments accurately describe the behaviors of test components. Unfortunately, current literature has revealed that both developer-written and tool-generated unit tests suffer from semantic clarity issues, such as ambiguous identifiers and insufficient comments [4], [6]. These issues in existing tests create a "noisy curriculum" for LLMs, limiting their ability to learn clear and meaningful patterns. Even worse, when examples fail to clearly articulate testing intent, such as mixing assertions for distinct behaviors in a single test, LLMs may inherit these ambiguities, resulting in generated tests with low coverage or logical errors. Hence, enhancing the semantic clarity of in-context test examples becomes pivotal to unlocking the full potential of ICL for test generation.

To address this limitation, our work focuses on the crucial aspect, i.e., the **semantic clarity** of in-context examples. Drawing on insights from existing studies [4], [6], [7] and our empirical observations, two common factors have been identified that hinder semantic clarity of unit tests: (i) complex logic arising from multiple test scenarios within a single test, and (ii) insufficient textual clarity such as ambiguous identifiers or missing essential comments. Intuitively, eliminating these issues can improve semantic clarity of unit tests, making it easier for LLMs to comprehend and learn from test examples.

Recently, Deljouyi et al. [6] proposed UTgen, which invokes LLMs to refine the unit tests generated by search-based tools to enhance their textual clarity, making them more semantically expressive. However, its effectiveness is unsatisfactory for two main reasons: on one hand, the complex logic of unit tests

[*] corresponding author

poses difficulties for LLMs to comprehend test semantics for test refinement with enhanced semantic clarity; on the other hand, LLMs' hallucinations could compromise the original unit tests' effectiveness (e.g., test coverage and syntactic correctness) after refinement. Embedding such non-effectiveness-preserving tests as examples could even negatively affect the effectiveness of such ICL-based test generation techniques, e.g., line coverage achieved by RAGGen on the Time project is decreased from 57.32% to 22.94% after refining test examples with UTgen (Section V-B). Thus, enhancing semantic clarity of unit tests to improve ICL-based unit test generation remains a challenging and non-trivial endeavor.

In this work, we propose a novel unit test refinement technique, called **CLAST** (**CLA**rifying **S**emantics of unit **T**ests), to enhance the semantic clarity of unit tests from the two aforementioned factors. To address the limitation of the complex test logic resulting from multiple scenarios (i.e., multiple assertions with different purposes) within a single unit test, CLAST splits a complex unit test into a set of purified ones, each of which describes a single test scenario, by slicing the test code from the assertion perspective. To address the limitation of insufficient textual clarity, CLAST leverages both LLMs and program analysis to refine identifiers and generate essential comments for each purified unit test. Compared to complex unit tests, LLMs can better learn the semantics of purified tests, enabling them to generate more precise identifiers and comments. Particularly, with the assistance of program analysis, CLAST avoids potential errors caused by LLMs' hallucinations for achieving effectiveness-preserving test refinement. Specifically, CLAST identifies the generated comments and refined identifiers from the contents produced by LLMs for clarity enhancement based on Abstract Syntax Tree (AST) and textual analysis, and then integrates those refined information into the original code of the purified test via AST node matching. Using CLAST, each unit test is refined into a set of purified tests that preserve the original effectiveness but have semantically expressive comments and identifiers. Embedding these refined tests as in-context examples could enable LLMs to glean more valuable knowledge, enhancing the effectiveness of ICL-based unit test generation.

To evaluate the effectiveness of CLAST, we conducted an extensive study using seven real-world Java projects, including four open-source projects and three industrial projects. We first evaluated whether CLAST can refine unit tests more effectively compared to the state-of-the-art test refinement technique (i.e., UTgen) [6] in terms of the degree to preserving the effectiveness of original tests. Our results show that UTgen largely damages test effectiveness after refinement, with 12.90%, 35.82%, 4.65%, and 5.07% decrements in terms of compilation success rate (CSR), pass rate (PR), line coverage (Cov), and mutation score (MS) respectively, while CLAST completely preserves the effectiveness of original unit tests. A subsequent user study demonstrated that the unit tests refined by CLAST exhibited superior semantic clarity compared to both the original tests and those refined by UTgen. Over 85.33% of participants favored the CLAST-refined tests.

Furthermore, we integrated CLAST to improve ICL-based unit test generation approaches by refining the original test examples. For this purpose, we selected the state-of-the-art RAGGen and TELPA as the ICL-based approaches to be further improved. The former used *developer-written* unit tests as examples while the latter employed *tool-generated* ones as counter-examples, indicating diverse scenarios for evaluating CLAST. Our results demonstrate that CLAST-refined test examples enable both RAGGen and TELPA to achieve better effectiveness compared to using original examples or UTgen-refined examples. Specifically, we observe average improvements of 25.97%, 28.22%, and 45.99% in CSR, PR, and Cov, respectively, when compared to UTgen-refined examples. The ablation study reveals that both test purification and program analysis-based post-processing contribute significantly to CLAST's overall effectiveness.

To sum up, our work makes the following contributions:

- We propose CLAST, a novel test refinement technique to enhance the semantic clarity of unit tests by leveraging both LLMs and program analysis. We have also made the replication package publicly available [8].
- We improve the effectiveness of ICL techniques for unit test generation from the novel perspective of enhancing the semantic clarity of in-context test examples.
- We conducted an extensive study to evaluate CLAST by measuring the quality (i.e., test-effectiveness-preserving degree and semantic clarity) of its refined unit tests and the improved effectiveness of ICL-based unit test generation with its refined unit test examples. Additionally, we performed a user study to assess developers' perceptions of the refined tests, revealing their practical value not only for test generation but also for broader applications such as test maintenance and debugging.

II. Motivation

We use a real-world example to motivate our work. Listing 1 shows an original unit test (that has been simplified for ease of illustration) for the method `getColumnMatrix`, which checks two behaviors: (1) retrieving the column matrix at index 3, and (2) throwing an exception for index 5. However, its semantic clarity is poor.

Listing 1: An example of an original unit test

```
public void testGetColumnMatrix() {
    RealMatrix m = new RealMatrixImpl(subTestData);
    RealMatrix mColumn3 = new RealMatrixImpl(
          subColumn3);
    assertEquals("Column3", mColumn3, m.
          getColumnMatrix(3));
    assertThrows(MatrixIndexException.class, () -> m.
          getColumnMatrix(5));
}
```

First, it mixes two distinct scenarios (i.e.,valid and invalid index handling) within a single test. Mixing different scenarios within one test could aggravate the test complexity and thus negatively affect the semantic clarity to some degree. Second, ambiguous identifiers (e.g., `mColumn3`) fail to convey their purpose, violating naming conventions and obscuring the test's intent. Such unclear tests hinder LLMs from learning effectively about unit test generation when used as in-context examples. In fact, using this test in RAGGen (an ICL-based unit test generation approach detailed in Section IV-D) yielded no improvement in line coverage compared to not using examples at all.

We then applied the state-of-the-art test refinement technique UTgen [6] to refine this unit test based on the DeepSeek-V2.5 model. As shown in Listing 2, the refined version improves textual clarity but unfortunately misinterprets the test's intent. Specifically, the LLM treats the test as checking boundary cases rather than specific indices 3 and 5, evident from comments at Lines 5–6 and the use of `matrix.getColumnDimension()-1` at Line 8, which causes an `AssertionError` since the column at index 3 is not equal to the column at index `matrix.getColumnDimension()-1`. Similarly, Line 13 incorrectly uses `matrix.getColumnDimension()` instead of 5. We investigated whether mixing different test scenarios contributes to this misunderstanding. Specifically, we split the original test into two, each focusing on a single index, and refined them separately with UTgen. The resulting tests correctly preserved the original intent, confirming that *mixing scenarios likely caused the misinterpretation*.

Besides, such misinterpretation can aggravate LLM hallucinations and compromise the original unit tests' functionalities. As shown in Listing 2, the UTgen-refined test calls a non-existent API `getColumnDimension`, making it invalid. Using this flawed test as an example in RAGGen led to lower line coverage (60.00%) than using the original test (72.00%). This shows that *poor refinement can even produce a negative effect on LLM-based test generation*. Motivated by these challenges, we design a novel test refinement technique (called CLAST) in this work, which aims to *enhance the semantic clarity of unit tests while preserving effectiveness of the original test*, thereby enhancing ICL techniques for unit test generation by taking refined tests as in-context examples.

Note that while hallucinations in this case were addressed by splitting the original test and refining them separately with UTgen, this success stems from the simplicity of the example chosen for ease of illustration. In more complex scenarios, even on split tests, UTgen (solely relying on LLMs) still introduces errors due to hallucinations, such as API misuse, which has been confirmed by our ablation study in Section V-C. This underscores the need for a more robust approach like CLAST, which *combines test purification with program analysis to avoid errors caused by hallucinations and ensure both semantic clarity and functional correctness.*

III. Methodology

Figure 1 provides an overview of CLAST. Given a (complex) unit test, CLAST first applies a test purification component to decompose it into a set of simpler and purified tests. Next, CLAST enhances the textual clarity of these purified tests using a program-analysis-enhanced approach. This approach integrates program analysis with LLMs' strong code

Listing 2: The unit test refined by UTgen

```
1  public void
       testRetrieveColumnMatrixWithValidAndInvalidIndices
       () {
2      //Given: A RealMatrix initialized with subTestData
3      RealMatrix matrix = new RealMatrixImpl(subTestData
           );
4      RealMatrix expectedLastColumn = new RealMatrixImpl
           (subColumn3);
5      //When: Retrieving the last column matrix
6      //Then: The retrieved last column matrix should
           match the expected last column matrix
7      assertEquals(expectedLastColumn, matrix.
8      getColumnMatrix(matrix.getColumnDimension() - 1));
9      //When: Attempting to retrieve a column matrix
           with an index equal to the column dimension
10     //Then: A MatrixIndexException should be thrown
11     assertThrows(MatrixIndexException.class,
12         () -> matrix.getColumnMatrix(
13         getColumnMatrix(matrix.getColumnDimension()));
14 }
```

comprehension capabilities to generate meaningful comments and more appropriate identifiers while minimizing the risk of errors caused by hallucinations. As a result, CLAST produces a set of refined tests that retain the original effectiveness while improving semantic expressiveness through clearer comments and identifiers, each targeting a single clear test scenario. Then, these refined tests can be used as high-quality in-context examples in ICL-based unit test generation approaches, enabling LLMs to better learn effective patterns and thus improving the effectiveness of unit test generation.

A. Test Purification

Test purification aims to produce a set of purified unit tests from each original test, with certain statements removed to isolate a single, clear test scenario. While program slicing tools like Slicer4J [9] could theoretically support this process, they are ill-suited to our needs due to two main issues: (1) reliance on dynamic analysis, which requires compilation and execution to gather runtime traces, and (2) overly complex designs that introduce unnecessary overhead for simplifying small-scale unit tests. Therefore, CLAST employs a lightweight static approach tailored for test purification, comprising three steps: (1) Statement Atomization: breaking tests into atomic units to prevent syntax errors or unintended deletions in the next step; (2) Test Atomization: splitting tests into simpler ones with a single assertion each. This is achieved through test slicing, which removes statements unrelated to the assertion, thereby simplifying complex logic. (3) Test Merging: merging atomic tests with identical prefixes (indicating they target similar or identical scenarios) to reduce redundancy.

1) Term Definition: We first define some terms formally for ease of representation. An **atomized statement** S_a is a unit of code representing a single logical operation, either a standalone expression (e.g., variable declaration or method call) or a control structure (e.g., `if`, `for`, `while`). Formally, $S_a = (T, V_r, V_w, C)$, where T denotes the statement type (normal or control structure), V_r is the set of variables read,

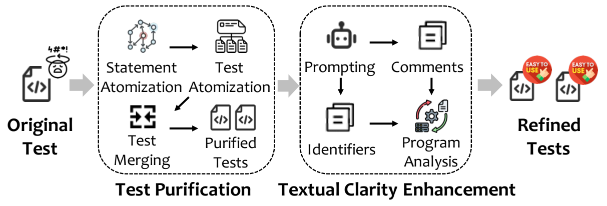

Fig. 1: Overview of CLAST

V_w is the set of variables written, and C is a control flag (C = true for control structures, i.e., S_c; otherwise, S_n). A variable is added to V_r if it appears in S_a and S_a is not an assignment or declaration, or if it appears on the right-hand side of such a statement. It is added to V_w if it appears on the left-hand side of an assignment/declaration, or if it serves as the caller or parameter in a method call whose name implies modification (e.g., "set", "add", "insert", "remove"). A full keyword list is available on our project homepage [8].

2) Statement Atomization: A single statement may contain multiple operations (e.g., `int a, b;` or `a = b = 1;`). To avoid unintended deletions or syntax errors when removing statements (at the next step), CLAST first breaks compound statements into atomic components. For a compound statement S, it produces a set of atomized statements $\{S_{a1}, S_{a2}, \ldots, S_{ak}\}$, each in the form $(T_i, V_{ri}, V_{wi}, C_i)$. Specifically, variable declarations with multiple identifiers are split into separate statements based on the type and identifiers; chained assignments are similarly decomposed. CLAST handles only multiple declarations and chain assignments, leaving other instructions (e.g., method call chains) unchanged to avoid unintended deletions later. This conservative strategy balances test-effectiveness preservation with brevity. Control structures are treated as indivisible units (e.g., $S_c = (T_c, V_{rc}, V_{wc}, \text{true})$) to avoid syntax issues such as removing only a `for` loop's condition expression, which would result in an incomplete loop. The body of the control structure is treated as a separate set of statements. This atomization reduces the risk of producing errors during subsequent test atomization.

3) Test Atomization: A unit test comprises two parts: *test prefix* and *assertion* [10]. The test prefix sets up the focal method with a series of method calls or assignments, while the assertion verifies its expected behavior. A test may contain multiple assertions, potentially testing different scenarios. To enhance simplicity and clarity, CLAST atomizes the test by splitting it into multiple tests, each containing only one assertion, and then slices each test to remove unrelated statements.

Formally, let the prefix T_p be a sequence of atomized statements $\{S_{a1}, S_{a2}, \ldots, S_{am}\}$, and $A = \{a_1, a_2, \ldots, a_n\}$ be the set of assertions. For each assertion a_i, CLAST creates a new test $T_i = T_p + a_i$. Then, CLAST performs backward slicing on each test T_i to remove statements unrelated to a_i. Specifically, CLAST builds a variable dependency graph $G = \{V, E\}$, where V are variables and E are directed edges $e_{v_i \to v_j}$ showing dependencies. Starting from each variable used in a_i, it collects all its reachable variables in G as the variables it depends on (denoted as V_{depen}). Then, for each normal statement $S_{nj} \in T_p$, CLAST removes S_{nj} if $V_{wj} \cap V_{depen} = \emptyset$. This ensures that only statements contributing to the assertion a_i are retained. Finally, any control structure S_{cj} with an empty body is removed.

4) Test Merging: After completing the above steps, we obtain tests each with a single assertion and its relevant prefix. However, some tests may share identical prefixes and validate different aspects of the same behavior through different assertions. To reduce redundancy, CLAST merges the group of tests that share the same prefix into a single test by combining the shared prefix and all the assertions within these tests. Formally, given tests $\{T_1, T_2, \ldots, T_k\}$ sharing prefix T_p, CLAST merges them into $T_{\text{merged}} = T_p + \{a_1, a_2, \ldots, a_k\}$, where a_i is the assertion from T_i. Note that while some highly-focused tests may be merged back into their original form, in most cases, the output for an original test is a set of logically clearer tests.

B. Textual Clarity Enhancement

Many studies have emphasized the importance of clear comments and meaningful identifiers for semantic clarity [11]–[13]. Hence, the goal of this component is to enhance semantic expressiveness of a given unit test by refining the two kinds of elements within the unit test. This process begins by using an LLM to generate comments or identifiers through carefully-crafted prompts, and then employs program analysis to integrate the LLM response with the original unit test, ensuring the preservation of test effectiveness and avoiding potential errors from LLM's hallucination. Note that the key contribution is not the idea of generating comments or identifiers with LLMs, which has been demonstrated by previous research [6], [13]. Rather, it is the program-analysis-based post-processing that ensures refinement accuracy and mitigates hallucination issues.

1) Comment and Identifier Generation via LLM Prompting: CLAST employs in-context learning to construct prompts by providing task-specific examples and instructions to guide the LLM. Specifically, CLAST applies one-shot in-context learning, which uses a single high-quality example to enhance the LLM's comprehension. For comment generation, the example includes a unit test alongside its enhanced version with comments following the "Arrange-Act-Assert" pattern [14]. For identifier generation, the example includes the original identifiers from the test alongside their enhanced version with carefully crafted expressive names. Our prompt design draws on the best practices derived from recent advances in prompt engineering research. Due to the space limit, the prompts used for generating comments and identifiers in CLAST are presented on our project homepage [8].

2) Post Processing via Program Analysis: In this process, CLAST extracts comments and identifiers from the above LLM-generated contents, and then employs program analysis, especially AST node matching, to seamlessly integrate them with the original unit tests.

For comment post-processing, CLAST first parses the AST of the LLM-generated test to extract block and inline comments. Block comments, typically docstrings, are placed at the beginning of the test. For inline comments, since LLM-refined tests may alter statements due to hallucinations, CLAST must map these comments back to the original test. Therefore, CLAST extracts their immediate right sibling nodes as context. Consecutive inline comments are merged into a single node before context extraction. Then, CLAST traverses the original test's AST, comparing each statement node to the comment's context node using syntactic and semantic similarity following prior work [15], [16]. Formally, for nodes v_1 (in original test) and v_2 (in refined test), the similarity is defined as:

$$\mathit{distance} = \mathit{type\_match}(v_1, v_2) \times \mathit{CodeBLEU}(v_1, v_2)$$

Here, *type_match* checks whether the node types of the two nodes are consistent (returning 1 for a match and 0 for a mismatch), while *CodeBLEU* represents the CodeBLEU similarity [17] between the nodes. If the similarity score exceeds a certain threshold, CLAST considers it a match and inserts the inline comment before the corresponding statement. Here, we conducted a preliminary study on a small test benchmark and then set the threshold to 0.7 based on the observed results.

For identifier post-processing, CLAST first extracts all the original identifiers and corresponding new identifiers from the LLM's output. Note that if the identifiers generated by the LLM contain duplicates, we re-instruct the LLM to avoid them. It then traverses the AST of the original unit test to extract all variable declaration nodes, and creates a mapping of these declaration nodes with their associated identifiers. Next, CLAST traverses the AST to identify all variable identifiers and record their positions. In reverse order, it replaces each identifier with the newly generated name suggested by the LLM, ensuring that replacements do not shift subsequent positions and cause errors. Finally, for the test name, CLAST locates the method declaration of the unit test and replaces the original test name with the newly-generated name.

In this way, CLAST outputs one or more refined tests with clearer logic, better comments, and more descriptive identifiers, for a given original test. These are then used as in-context examples to boost ICL-based test generation. When multiple tests are produced, all are included to preserve completeness and ensure fidelity to the original test. Listing 3 shows how CLAST splits mixed scenarios into two focused tests (valid and invalid index cases),enhancing clarity and intent. For example, Line 7's comment precisely describes the action at Line 8. Using these refined tests in RAGGen raised coverage to 84.00%, versus 72.00% with the original test example and 60.00% with UTgen-refined test example.

IV. Evaluation Design

A. Research Questions

RQ1: How effective is CLAST in refining unit tests? We evaluated whether CLAST preserves the effectiveness (e.g., compilation success rate, pass rate, test coverage) of original unit tests while improving their semantic clarity.

Listing 3: The unit test refined by CLAST (docstrings are omitted here for brevity)

```
1  /* Omitted for saving space */
2  public void testRetrieveColumnAsSubMatrix() {
3      // Arrange: Create a RealMatrix instance using
           subTestData
4      RealMatrix matrixUnderTest = new RealMatrixImpl(
           subTestData);
5      // Create a RealMatrix instance representing the
           expected column matrix for column 3
6      RealMatrix expectedColumnMatrix = new
           RealMatrixImpl(subColumn3);
7      // Act: Retrieve the column matrix at index 3
8      RealMatrix retrievedColumnMatrix=matrixUnderTest
9                      .getColumnMatrix(3);
10     // Assert
11     // Verify that the retrieved column matrix matches
            the expected column matrix
12     assertEquals(expectedColumnMatrix,
           retrievedColumnMatrix);
13 }
14 /* Omitted for saving space */
15 public void
16 testGetColumnMatrixWithInvalidIndicesThrowsException()
17 {
18     // Arrange
19     RealMatrix matrixInstance = new RealMatrixImpl(
           subTestData);
20     // Act and Assert
21     // Get a column matrix with an index out of bounds
22     assertThrows(MatrixIndexException.class,
23         () -> matrixInstance.getColumnMatrix(5));
24 }
```

RQ2: To what extent do the refined unit tests by CLAST enhance the effectiveness of ICL-based unit test generation? We assessed whether CLAST-refined examples improve the effectiveness of state-of-the-art ICL-based approaches (i.e., RAGGen [4], TELPA [5]) compared to using original or baseline-refined examples.

RQ3: How does each component in CLAST contribute to its effectiveness? We conducted an ablation study to evaluate the impact of CLAST's key components: test purification and program-analysis-based post-processing.

B. Subjects

Following the existing work in unit test generation [4], [18], we evaluated CLAST on Java projects using the JUnit framework [19] due to the significant popularity of Java and JUnit. In total, we used seven real-world Java projects as our subjects, including four open-source projects and three industrial projects. Following the existing work [20], we used the latest versions of four Java projects in the widely-used Defects4J benchmark [21] (i.e., Chart, Time, Lang, and Math), excluding Closure due to its lack of JUnit tests and high testing costs. While these projects may appear in the LLM training data, the models were not specifically trained for test generation or refinement, which mitigates potential data leakage concerns.

Particularly, to further reduce data leakage risks, we adopted three internal Java projects provided by our industrial partner (a global leader company in IT). The three industrial projects have different functionalities, i.e., a program analysis toolkit, an online micro-service system, and a data analysis framework involving parallel computing and adaptation of design patterns. For ease of presentation, we refer to them as PATool, Microservice and DAService in the following sections. Due to the company policy, we are unable to disclose further details. This diverse range of subjects allows us to thoroughly evaluate CLAST's generalizability.

C. Metrics

We first evaluated refined unit tests using two key measurements: *test-effectiveness-preserving degree* and *semantic clarity*. In line with the existing studies [6], we measured whether refinement preserves test effectiveness using three widely-used metrics: Compilation Success Rate (CSR), Pass Rate (PR), Line Coverage (Cov), and Mutation Score (MS). CSR and PR represent the ratios of successfully compiled and executed tests, respectively, Cov measures the average line coverage across focal methods, and MS measures the proportion of artificially injected faults (mutants) that the tests successfully detect [10], [22], [23]. We calculated the difference between refined and original tests for each metric. Zero differences indicate full effectiveness preservation, while negative values suggest effectiveness degradation.

To evaluate the semantic clarity of refined tests, following prior work [6], [12], [13], we conducted a user study with 15 participants experienced in Java and testing, averaging 6.8 years of development experience (10 from industry and 5 from academia). We randomly sampled 10 focal methods from the four Defects4J projects ($\geq$ 1 per project) for diversity and asked participants to rank the original test and the tests refined by different techniques for each method on three criteria: (1) *conciseness* (clarity of test scenarios), (2) *descriptiveness* (quality of identifiers), and (3) *comment quality* (clarity of comments). We adopted a rank-based scale to minimize rating bias and support clearer comparisons. A rank of 1^{st} indicates the best performance. To avoid bias, participants were unaware of which technique was used for each test.

We then evaluated the impact of refined tests on ICL-based test generation approaches by measuring CSR, PR, and Cov of the generated tests by these approaches using in-context examples from different refinement techniques.

D. Studied ICL-based Unit Test Generation Approaches

To evaluate CLAST's impact on ICL-based test generation, we enhanced two state-of-the-art approaches (i.e., RAGGen [4] and TELPA [5]). They incorporated different types of unit tests as in-context examples with different purposes, indicating diverse scenarios for evaluating CLAST.

RAGGen retrieves the most similar method to the focal method and its *developer-written* unit tests from a database. Then, it incorporates the identified method and all its associated unit tests as examples in the prompt, which enables LLMs to learn more knowledge on generating unit tests. TELPA first uses the widely-used search-based tool (i.e., EvoSuite [2]) to generate initial tests, then switches to LLMs when coverage is insufficient, using *tool-generated* tests as counter-examples.

For more sufficient evaluation, we used two LLMs for both approaches: CodeLlama-7b-Instruct-hf (*CL-7B*) and deepseek-coder-6.7b-instruct (*DS-7B*). Both LLMs have been demonstrated effective in code-related tasks [24]–[26], and the $\sim$7b size balances cost and effectiveness well. Note that our goal is to evaluate whether refining in-context tests examples can enhance the effectiveness of RAGGen and TELPA, rather than compare the effectiveness of different LLMs, and thus we only need to control the same LLM when comparing refined and original unit tests on an ICL-based approach. Studying two underlying LLMs for each ICL-based approach helps improve the evaluation's generalizability.

E. Baselines

First, we should understand the quality difference between unit tests refined by CLAST and the original unit tests without refinement, thus we treated the original unit tests (denoted as **Origin**) as one baseline. Then, we should understand the effectiveness of CLAST compared to other test refinement techniques. Here, we selected the state-of-the-art unit test refinement technique (i.e., **UTgen** [6]) as another baseline. It employs an LLM to contextualize test data and improve textual clarity.

F. Implementation and Environment

We implemented CLAST in Python, using the tree-sitter tool [27] for AST analysis and DeepSeek-V2.5 via its API [28] as the underlying LLM. For open-source projects, we used Defects4J's framework and Cobertura [29] for coverage collection. Industrial projects were tested using internal frameworks. Experimental scripts were developed with PyTorch 2.0.0 [30] and Transformers 4.34.1 [31], accelerated by the vLLM library [32]. For UTgen, TELPA, and RAGGen, we used their publicly released artifacts. For fair comparisons between CLAST and UTgen, we also employed DeepSeek-V2.5 as the underlying LLM for UTgen. All experiments ran on an Ubuntu 18.04 server with an Intel Xeon Gold 6240C CPU, 512GB RAM, and four NVIDIA A800 GPUs.

V. Results and Analysis

A. RQ1: Effectiveness Comparison in Unit Test Refinement

1) Process: In this RQ, we examined the quality of the refined unit tests by CLAST. We considered two types of unit tests: **developer-written tests** and **tool-generated tests**. For developer-written tests, we gathered all unit tests included in each open-source project. For tool-generated tests, we employed EvoSuite [2], a widely-used search-based test generation tool, to automatically create a test suite for each project. Then, we sampled 500 developer-written unit tests and 500 tool-generated unit tests for refinement by balancing generalizability and cost similar to the existing work [10], [20]. Note that we excluded the industrial projects in this RQ for two reasons. First, the developer-written unit tests for these projects are not accessible. Second, the three industrial projects use Java 17, but EvoSuite only supports up to Java 11, making it impossible to generate tests for them.

TABLE I: Comparison between CLAST, Origin, and UTgen in test-effectiveness-preserving degree (RQ1)

Technique	Developer-written Tests				Tool-generated Tests			
	CSR	PR	Cov	MS	CSR	PR	Cov	MS
Origin	100.00%	99.75%	48.49%	73.97%	100.00%	100.00%	43.43%	57.21%
ΔUTgen	-10.07%	-32.30%	-3.76%	-3.29%	-15.73%	-39.33%	-5.53%	-6.84%
ΔCLAST	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

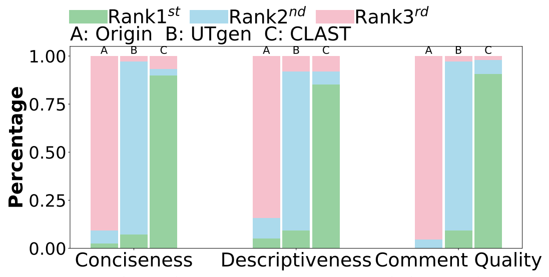

Fig. 2: Comparison between CLAST, Origin, and UTgen in terms of semantic clarity (RQ1)

We then used UTgen and CLAST to refine the two sets of original unit tests (directly used by the Origin baseline). Subsequently, we measured the quality of the original, UTgen-refined, and CLAST-refined unit tests. We assessed their test-effectiveness-preserving degree using CSR, PR, and Cov metrics. Additionally, we evaluated semantic clarity in terms of conciseness, descriptiveness, and comment quality through a user study.

2) Results: Table I presents the comparison results for the test-effectiveness-preserving degree, where Rows "ΔUTgen" and "ΔCLAST" represent the difference between refined and original unit tests in terms of the corresponding metric. From this table, CLAST maintains unit test effectiveness after refinement, while UTgen experiences significant declines across all three metrics for both developer-written and tool-generated tests. For example, for tool-generated tests, CLAST achieves identical CSR (100.00%), PR (100.00%), Cov (43.43%), and MS(57.21%) compared to Origin, while UTgen's performance deteriorates, with CSR, PR, Cov, and MS decreases by 15.73%, 39.33%, 5.53%, and 6.84% respectively. This decline likely stems from the LLM's hallucination problem.

Figure 2 shows the comparison results for the semantic clarity. In this figure, each bar shows the ranking distribution for the tests refined by a technique in terms of a metric. The user study demonstrates that CLAST-refined tests exhibit better semantic clarity, improving their readability and comprehensibility. Specifically, CLAST-refined unit tests receive the highest percentage of first-place rankings across all the three metrics compared to those refined by UTgen and the original tests: 90.00% for conciseness, 85.33% for descriptiveness, and 90.67% for comment quality. Note that among the ten randomly sampled methods for the user study, five are with developer-written tests and the other five are with tool-generated tests. The conclusions remain consistent across both types. This highlights participants' strong preference for CLAST-refined unit tests, validating the practical value of CLAST in enhancing semantic clarity.

RQ1 Summary: CLAST exhibits the superior test-effectiveness-preserving ability while significantly enhancing test semantic clarity compared to UTgen. Specifically, CLAST completely retains the original effectiveness of the unit tests, whereas UTgen decreases CSR, PR, Cov, and MS by 12.90%, 35.82%, 4.65%, and 5.07% on average. Furthermore, over 85.33% of the user study participants favored the semantic clarity (i.e., conciseness, descriptiveness, and comment quality) of the unit tests refined by CLAST.

B. RQ2: Effectiveness Comparison in Enhancing Unit Test Generation

1) Process: For each studied ICL-based test generation approach (i.e., RAGGen and TELPA) with each studied LLM (i.e. CL-7B and DS-7B), we incorporated the original, UTgen-refined, CLAST-refined unit test examples into the prompt for generating unit tests, respectively. To better understand the effect of incorporating in-context test examples, we also ran these approaches without any test examples (denoted as "Base" for ease of presentation). Then, we measured the effectiveness of the generated unit tests by each technique in terms of CSR, PR, and Cov. As previously mentioned, the three industrial projects are based on Java 17 but the EvoSuite tool used by TELPA only supports up to Java 11, and thus we just applied TELPA to the four open-source projects in this experiment.

2) Results: Table II compares the effectiveness of generated unit tests. In this table, Columns "Base" represent the results of these ICL-based unit test generation approaches without test examples, while "Origin", "UTgen", and "CLAST" represent the results of these ICL-based approaches with original, UTgen-refined, and CLAST-refined examples, respectively. The best results among the four techniques (Base, Origin, UTgen, and CLAST) are marked as bold on each project in terms of each metric. Note that RAGGen targets all focal methods within a project while TELPA just targets the focal methods that are not fully covered by the used search-based tool within the given testing period, and thus it is meaningless to compare RAGGen and TELPA based on this table.

From Table II, we first compare the effectiveness of Origin and Base to investigate the effect of incorporating test examples into the prompt. In most cases, Origin indeed helps generate more effective unit tests than Base. For example, Origin improves the effectiveness of RAGGen with CL-7B and DS-7B by 29.03% and 11.38% compared to Base in terms of the average line coverage across all the studied projects.

TABLE II: Comparison between CLAST, Origin, and UTgen in terms of the effectiveness for test generation (RQ2)

App.	**Metric**	**Project**	**CL-7B**				**DS-7B**			
			Base	**Origin**	**UTgen**	**CLAST**	**Base**	**Origin**	**UTgen**	**CLAST**
RAGGen	CSR	Time	41.02%	55.66%	44.12%	**65.00%**	59.26%	66.03%	27.64%	**67.96%**
		Math	44.51%	41.86%	22.22%	**66.67%**	51.63%	53.85%	48.15%	**54.41%**
		Lang	33.70%	55.75%	62.55%	**72.14%**	75.70%	76.84%	51.19%	**87.54%**
		Chart	50.45%	63.13%	46.30%	**64.38%**	70.00%	79.07%	52.03%	**79.12%**
		MicroService	35.71%	**42.86%**	40.00%	**42.86%**	**38.46%**	23.08%	15.38%	**38.46%**
		PATool	82.86%	77.14%	82.86%	**88.57%**	85.71%	82.86%	71.43%	**91.43%**
		DAService	41.18%	44.12%	41.18%	**52.94%**	31.43%	37.14%	37.14%	**48.57%**
	PR	Time	22.09%	33.03%	31.15%	**34.30%**	37.04%	42.31%	6.54%	**45.63%**
		Math	26.40%	**30.23%**	22.58%	28.42%	30.59%	34.62%	39.58%	**39.71%**
		Lang	16.87%	22.71%	31.85%	**32.30%**	30.39%	24.29%	25.49%	**50.33%**
		Chart	22.97%	36.60%	24.05%	**41.37%**	35.71%	42.44%	32.12%	**53.82%**
		MicroService	18.73%	29.82%	15.69%	**32.31%**	25.34%	27.87%	18.49%	**26.43%**
		PATool	40.76%	34.72%	38.58%	**49.92%**	47.36%	51.42%	49.13%	**54.98%**
		DAService	22.14%	27.40%	25.35%	**30.74%**	33.73%	35.43%	37.29%	**47.42%**
	Cov	Time	51.92%	57.32%	22.94%	**63.41%**	43.42%	56.63%	18.12%	**70.06%**
		Math	45.57%	52.43%	14.52%	**62.97%**	49.41%	51.88%	42.44%	**61.42%**
		Lang	33.10%	63.22%	50.31%	**71.80%**	55.95%	71.56%	25.39%	**79.10%**
		Chart	28.54%	59.03%	43.59%	**63.31%**	48.96%	60.01%	27.04%	**70.52%**
		MicroService	19.64%	29.21%	30.79%	**31.29%**	20.46%	16.54%	11.54%	**29.54%**
		PATool	72.03%	66.11%	65.03%	**76.06%**	71.86%	70.26%	60.31%	**78.91%**
		DAService	23.06%	26.03%	22.97%	**30.38%**	24.60%	23.60%	25.00%	**35.49%**
TELPA	CSR	Time	68.77%	70.00%	59.26%	**77.33%**	41.38%	67.01%	66.34%	**69.91%**
		Math	30.00%	71.43%	**80.00%**	**80.00%**	33.33%	**88.89%**	**88.89%**	**88.89%**
		Lang	81.79%	**92.63%**	86.96%	92.55%	69.57%	93.59%	91.04%	**94.17%**
		Chart	72.49%	83.02%	80.00%	**94.12%**	78.75%	89.47%	84.31%	**90.29%**
	PR	Time	17.89%	**58.00%**	37.04%	57.33%	25.86%	42.27%	40.59%	**43.36%**
		Math	20.00%	57.14%	**60.00%**	**60.00%**	0.00%	**55.56%**	**55.56%**	**55.56%**
		Lang	34.97%	**71.05%**	55.43%	70.21%	36.96%	63.46%	58.96%	**65.83%**
		Chart	22.61%	33.96%	50.77%	**60.50%**	41.25%	52.15%	52.18%	**56.00%**
	Cov	Time	21.75%	36.64%	32.19%	**41.78%**	15.78%	28.05%	30.60%	**33.77%**
		Math	29.17%	31.13%	**42.16%**	**42.16%**	17.95%	40.72%	**44.99%**	**44.99%**
		Lang	20.59%	23.08%	26.40%	**35.20%**	18.87%	32.81%	34.43%	**37.86%**
		Chart	17.92%	19.78%	28.03%	**30.94%**	18.37%	25.00%	25.76%	**26.31%**

This demonstrates the value of incorporating unit test examples into the prompt for LLM-based unit test generation. However, the improvement of Origin over Base is limited in many cases, e.g., Cov is just improved from 49.41% to 51.88% for RAGGen with DS-7B on Math. This may be attributed to the lack of semantic clarity for the test examples, thereby preventing the LLM from effectively learning from them.

We then compare the effectiveness of CLAST, Origin, and UTgen, for enhancing ICL-based unit test generation. From Table II, CLAST almost always performs the best among them. For instance, on average across all the studied projects, using RAGGen with CL-7B, CLAST improves the Base, Origin, and UTgen methods by 37.38%, 18.93%, and 33.41% in CSR; 46.72%, 16.25%, and 31.76% in PR; and 45.78%, 12.98%, and 59.59% in Cov, respectively. For RAGGen with DS-7B, CLAST enhances the Base, Origin, and UTgen methods by 13.42%, 11.61%, and 54.31% in CSR; 32.54%, 23.20%, and 52.57% in PR; and 35.08%, 21.27%, and 102.55% in Cov, respectively. These trends persist for both TELPA with CL-7B and TELPA with DS-7B. In the few instances where CLAST is not the top performer, these occurrences are limited to CSR and PR measurements with only slight underperformance. However, the corresponding Cov achieved by CLAST is significantly higher than those by baselines, potentially offering greater practical value.

Besides, we find that UTgen-refined unit tests often produce a negative influence on ICL-based unit test generation compared to directly using the original tests. For example, Cov across all studied projects achieved by UTgen varies for different configurations: 35.74% for RAGGen with CL-7B, 29.98% for RAGGen with DS-7B, 32.20% for TELPA with CL-7B, and 33.95% for TELPA with DS-7B. In contrast, the average Cov achieved by Origin is 50.48%, 50.07%, 27.66%, and 31.65% for the same configurations, respectively. The underperformance of UTgen is mainly due to its tendency to introduce incorrect semantic information, such as altering original test functionalities or introducing invalidity, which misleads the LLM during the unit test generation process.

RQ2 Summary: Incorporating CLAST-refined test examples into prompts enhances ICL-based unit test generation, with an average improvement of 10.07%/13.88%/20.71% and 25.97%/28.22%/45.99% compared to using original and UTgen-refined test examples in terms of CSR/PR/Cov, respectively.

TABLE III: Comparison between CLAST and its variants in test-effectiveness-preserving degree (RQ3)

Technique	**Developer-written Tests**				**Tool-generated Tests**			
	CSR	**PR**	**Cov**	**MS**	**CSR**	**PR**	**Cov**	**MS**
Origin	100.00%	99.75%	94.07%	73.97%	100.00%	100.00%	43.43%	57.21%
ΔCLAST	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Δ*w/o purify*	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Δ*w/o post*	-19.66%	-21.29%	-17.30%	-2.34%	-5.26%	-6.01%	-1.24%	-0.76%
Δ*w/o both*	-23.10%	-25.38%	-17.91%	-4.18%	-12.22%	-12.87%	-5.49%	-0.86%

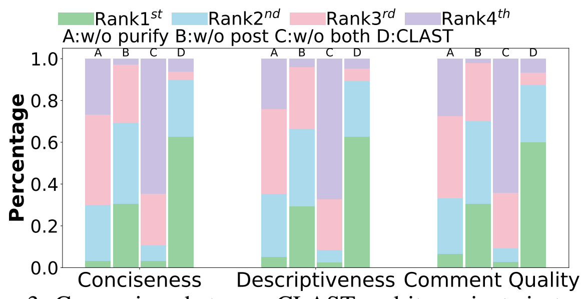

Fig. 3: Comparison between CLAST and its variants in terms of semantic clarity (RQ3)

C. RQ3: Ablation Study

1) Process: In this RQ, we conducted an ablation study to examine the contribution of each core component (i.e., test purification and program-analysis-based post-processing) to the overall effectiveness of CLAST. Accordingly, we constructed three variants of CLAST:

- "*w/o purify*", which removes the component of test purification from CLAST. That is, it directly performs textual clarity enhancement on the original unit tests rather than the purified ones.
- "*w/o post*", which removes the component of program-analysis-based post-processing from CLAST. That is, it instructs the LLM to enhance textual clarity on each purified unit test and then directly utilizes the LLM-generated test as the refined one.
- "*w/o both*", which removes both components (i.e., test purification and program-analysis-based post-processing). The prompting process for "*w/o both*" is similar to UTgen's, but the former directly adopts the initial LLM-generated test as the refined version while the latter includes an iterative refinement process with LLMs.

We evaluated these variants in two scenarios: (1) refining unit tests (as in RQ1), and (2) using refined unit tests to enhance ICL-based unit test generation (as in RQ2).

2) Results: Table III compares the test-effectiveness-preserving degree across CLAST and its variants. The program-analysis-based post-processing component is crucial for preserving test effectiveness. The "*w/o post*" variant significantly degrades effectiveness, with average decreases of 12.46%, 13.65%, 9.27%, and 1.55% in CSR, PR, Cov, and MS respectively, due to LLM hallucinations. The "*w/o both*" variant, which removes both purification and post-processing, further damages effectiveness, highlighting the importance of simplifying tests before refinement. Notably, "*w/o purify*" achieves the same results as CLAST because it retains post-processing, ensuring that the effectiveness of the test cases remains intact.

Figure 3 shows the semantic clarity results. Test purification significantly enhances clarity, as "*w/o purify*" and "*w/o both*" receive fewer first-place rankings and more last-place rankings compared to CLAST and "*w/o post*". This demonstrates that purification improves LLMs' comprehension, leading to more accurate comments and identifiers.

Table IV evaluates the impact of refined tests on ICL-based test generation. CLAST consistently outperforms its variants across all metrics, ICL-based unit test generation approaches (RAGGen and TELPA), LLMs (CL-7B and DS-7B), and project types (open-source and industrial projects). Removing post-processing reduces CSR, PR, and Cov by 8.78%, 9.01%, and 12.25%, on average across all ICL-based approaches, LLMs, and projects, while removing purification reduces them by 4.92%, 11.23%, and 6.77%. The results confirm that both test effectiveness preservation (contributed by program-analysis-based post-processing) and clarity enhancement (contributed by test purification) are crucial to the capability of CLAST in improving ICL-based test generation.

RQ3 Summary: Both core components (i.e., test purification and program-analysis-based post-processing) contribute significantly to CLAST's effectiveness, which improves both the quality of refined tests and the effectiveness of ICL-based unit test generation.

VI. Discussion

Perceptions on CLAST. To further assess the practical value of CLAST, we asked user study participants to rank their willingness to incorporate refined tests into real-world projects. CLAST-refined tests received the highest first-place ranking (90.67%), compared to UTgen (8.00%) and original tests (1.33%), confirming their practicality and effectiveness. Participants also provided insights on desirable test characteristics. We analyzed their answers using a card sorting method [33]. The two key insights are summarized:

- *Concise Scenarios and Clear Textual Clarity*: 80.0% of participants emphasized that concise scenarios and straightforward naming conventions improve comprehension for both developers and LLMs. CLAST-refined tests were praised for their clarity and structure, which reduce ambiguity and enhance LLM-friendliness.

TABLE IV: Comparison between CLAST and its variants in terms of the effectiveness for ICL-based test generation (RQ3)

LLM	Technique	RAGGen						TELPA		
		Open-source Projects			Industrial Projects			Open-source Projects		
		CSR	PR	Cov	CSR	PR	Cov	CSR	PR	Cov
CL-7B	CLAST	**67.05%**	**36.60%**	**65.37%**	**61.46%**	**37.66%**	**45.91%**	**86.00%**	**62.01%**	**37.52%**
	w/o purify	62.72%	35.00%	60.82%	58.54%	36.26%	42.39%	84.33%	48.82%	37.39%
	w/o post	64.02%	34.19%	57.20%	49.97%	34.87%	38.68%	84.23%	60.23%	36.28%
	w/o both	58.97%	34.10%	55.91%	51.79%	34.64%	40.39%	76.70%	56.21%	29.25%
DS-7B	CLAST	**72.26%**	**47.37%**	**70.28%**	**59.49%**	**42.94%**	**47.98%**	**85.82%**	**55.19%**	**35.73%**
	w/o purify	68.90%	39.56%	65.79%	51.50%	36.69%	40.39%	84.83%	53.80%	35.50%
	w/o post	61.59%	38.66%	57.73%	52.16%	35.49%	41.86%	82.16%	52.95%	33.96%
	w/o both	65.48%	36.35%	55.99%	48.64%	34.32%	39.11%	82.75%	52.95%	29.98%

TABLE V: Impact of CLAST on HITS Performance

Metric	w/o RAG	Origin	UTgen	CLAST
CSR	32.37%	32.46%	27.31%	**34.27%**
PR	9.99%	14.59%	14.13%	**16.18%**
Cov	21.60%	39.19%	38.50%	**41.90%**

TABLE VI: Generalizability of CLAST on larger LLMs

Technique	Open source projects			Industrial projects		
	CSR	PR	Cov	CSR	PR	Cov
Base	68.34%	61.99%	75.96%	70.25%	71.42%	47.54%
Origin	75.76%	67.94%	87.66%	75.31%	72.84%	48.14%
UTgen	70.62%	63.77%	77.61%	70.12%	65.28%	45.08%
CLAST	**79.31%**	**71.44%**	**90.09%**	**78.21%**	**73.98%**	**50.12%**

- *Avoiding Excessive Documentation*: While necessary comments are valuable, 33.3% of participants mentioned overly detailed documentation, which can complicate tests and affect semantic clarity. This necessitates adopting a more adaptive approach such as incorporating a complexity-based filtering mechanism to avoid redundancy.

Efficiency of CLAST. While CLAST demonstrates remarkable performance, it is essential to consider the trade-off between its effectiveness and cost. Specifically, we measured the time efficiency of both CLAST and UTgen. The results indicate that UTgen takes an average of 93.327s to refine a single unit test, whereas CLAST requires only 55.133s (i.e., allocating 0.003s for test purification and 55.13s for textual clarity enhancement). The reason may be that UTgen iteratively invokes the LLM for refinement, but CLAST only requires a single round of invocation. Moreover, the potential for parallel execution could further accelerate the refinement process of CLAST, boosting its practicality. These findings suggest that CLAST offers a significantly more efficient and scalable refinement process.

Generalizability. To highlight the benefit of ICL in LLM-based test generation and CLAST's generalizability, we integrated CLAST into HITS [34], a state-of-the-art LLM-based test generation approach that originally lacks in-context examples. We enhanced HITS with Retrieval-Augmented Generation (RAG) to retrieve similar methods and unit tests for in-context learning. Specifically, we adopted RAGGen's well-established RAG approach. We evaluated HITS in four settings: without RAG, with original test examples, with UTgen-refined examples, and with CLAST-refined examples, using four open-source projects from Defects4J and DS-7B as the representative. Table V shows the average results across all projects. From the table, incorporating in-context examples indeed improves HITS' effectiveness, underscoring the importance of in-context examples. Notably, CLAST-refined examples consistently outperform both the original and UTgen-refined examples, demonstrating CLAST's effectiveness and generalizability in enhancing LLM-based unit test generation.

To investigate CLAST's generalizability across LLM scales, we evaluated DeepSeek-V3 (610B), a state-of-the-art model surpassing GPT-4 on code/math tasks [35] on RAGGen. The results are summarized in Table VI. From the results, our core findings are reinforced: while larger models perform better, CLAST-refined examples consistently enhance performance, improving Base/Origin/UTgen by 13.69%/4.27%/11.92% in CSR, 9.41%/3.26%/12.68% in PR, and 12.01%/3.44%/13.63% in Cov. Though improvements are smaller than with 7B models, likely due to the already high performance of larger models (e.g., DeepSeek-V3 alone improves Cov by 28.19% over DS-7B), CLAST still boosts performance, demonstrating its generalizability across LLM scales.

Regarding the generalizability to different languages and real-world settings. CLAST's idea is language-agnostic. Adapting it involves replacing language-specific implementations (e.g., AST parsing) and refining prompts to match the target language's syntax and testing conventions. Tools like Tree-sitter (for JavaScript/Python/C++) provide support for program analysis, while prompt adjustments just require updating examples to align with frameworks like unittest. This makes CLAST's extension highly feasible. Additionally, CLAST takes an average of 55.13 seconds per test, with most of the overhead coming from LLM inference rather than static analysis/transformation. This is comparable to existing LLM-based tools like Copilot, which are already integrated into real-world workflows as asynchronous aids, not in latency-critical paths. This makes CLAST similarly deployable.

Future Work. Several promising directions include: (I) Mitigating the risk of excessive documentation by guiding LLMs to insert comments at key points, identified based on factors such as statement complexity and importance. (II) Incorporating runtime information, such as test coverage, to enhance naming and commenting by clarifying each test's purpose. (III) Extending CLAST to enhance supervised fine-tuning of LLM-based test generation by refining training data and provide high-quality unit tests to aid developers in software debugging and maintenance.

VII. Threats to Validity

The threats to **external** validity mainly lie in subject selection and studied ICL-based unit test generation approaches. To mitigate these, we evaluated CLAST on both open-source and industrial projects, and integrated it with two advanced ICL-based approaches, RAGGen and TELPA, covering diverse test example usage scenarios. In future work, we can extend CLAST to more approaches, including those without test example incorporation originally.

The threat to **internal** validity primarily lies in the implementation. CLAST underwent rigorous code review and testing by three authors. For UTgen, RAGGen, and TELPA, we used their publicly released artifacts [36]–[38]. LLMs (CL-7B, DS-7B from Hugging Face [39] and DeepSeek-V2.5 API [28]) were used following their official guidelines.

The threats to **construct** validity include LLM randomness, data leakage, metric selection, and potential semantic drift. Following the existing work, we set LLM temperature to zero and repeated all quantitative experiments for 10 times and reported the average results to reduce randomness [4], [5], [40]–[42]. To prevent data leakage, we used internal industrial projects and tool-generated tests free from LLM training data. To address the metric threat, we employed diverse metrics (CSR, PR, Cov, and MS) to comprehensively evaluate CLAST's effectiveness. To address the final threat, CLAST uses carefully-designed prompts with task-specific examples to generate identifiers and comments that preserve code semantics. Though minor semantic drift (e.g., mColumn3→expectedColumnMatrix) may occur, this is mitigated via complementary reinforcement: descriptive comments clarify identifiers (e.g., "expected column matrix for column 3"), and vice versa. Additionally, embedding-based similarity checks further align comments with intent.

VIII. Related Work

Test Purification. Xuan and Martin [43] first introduced test purification, which reduces the size of a test case via program slicing from the assertion to improve fault localization. Their approach extracts minimal failure-inducing statement subsets using dynamic slicing. In contrast, CLAST enhances semantic clarity for LLMs by producing each example that focuses on a clear scenario with thorough intra-scenario coverage, making the example focused yet comprehensive for LLMs to learn high-quality testing patterns. Since LLMs learn statically, aggressive pruning (as in dynamic slicing) may remove useful structural or semantic cues. Static slicing is also lighter and avoids runtime instrumentation.

Test Quality Improvement. Prior work has focused on improving test quality, primarily addressing test smells and readability. Studies like Soares et al. [44] and Peruma et al. [45] explored strategies to reduce test smells, while Lucas et al. [46] and Gao et al. [47] investigated LLMs for detecting and repairing test smells. Other efforts, such as Zhang et al. [48] and Daka et al. [12], aimed to enhance readability by generating descriptive test names or summaries. Recently, Deljouyi et al. [6] introduced UTgen, leveraging LLMs to improve semantic clarity. However, it struggles with complex test scenarios and LLM hallucinations. Unlike them, CLAST proposes a novel two-step approach that purifies complex tests by splitting them into clearer ones and refines their semantic clarity by leveraging LLMs and program analysis.

LLM-based Unit Test Generation. LLM-based test generation approaches fall into two categories: training-based and prompting-based. Training-based methods, such as ATHENATEST [49] and A3Test [50], train LLMs on large datasets of unit tests, achieving strong results but requiring significant resources. Prompting-based approaches, like ChatTester [10], SymPrompt [22], and HITS [34], use contextual prompts to guide LLMs in generating tests, offering flexibility and reduced reliance on fine-tuning. Recent advancements, including RAGGen and TELPA, demonstrate the effectiveness of incorporating in-context-learning (i.e., providing in-context test examples in prompts) for prompting-based test generation. *Unlike these approaches, which focus on context construction or example selection, CLAST enhances in-context-learning effectiveness by improving the semantic clarity of test examples.* In general, our work is orthogonal to existing methods and can be integrated to further improve their performance.

IX. Conclusion

Existing ICL-based unit test generation techniques are hindered by the limited semantic clarity of unit test examples. To tackle this issue, we developed CLAST, an innovative refinement technique that enhances the quality of unit test examples by splitting a complex test into a set of purified ones and improving their textual clarity using a combination of LLMs and program analysis. Our extensive evaluation on real-world projects demonstrates that CLAST significantly outperforms the state-of-the-art test refinement technique UTgen in both preserving test effectiveness and enhancing the semantic clarity of unit tests. Results also show that incorporating CLAST-refined unit tests can effectively enhance LLM-based unit test generation, i.e., RAGGen and TELPA.

Acknowledgment

We thank all the ASE anonymous reviewers for their valuable comments. This work was supported by the National Key Research and Development Program of China (Grant No. 2024YFB4506300), the National Natural Science Foundation of China (Grant Nos. 62322208, 62232001), and the Emerging Frontiers Cultivation Program of Tianjin University Interdisciplinary Center.

References

[1] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for python," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 168–172.

[2] 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.

[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] L. Yang, C. Yang, S. Gao, W. Wang, B. Wang, Q. Zhu, X. Chu, J. Zhou, G. Liang, Q. Wang, and J. Chen, "On the evaluation of large language models in unit test generation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, p. 1607–1619.

[5] C. Yang, J. Chen, B. Lin, Z. Wang, and J. Zhou, "Advancing code coverage: Incorporating program analysis with large language models," *ACM Transactions on Software Engineering and Methodology*, 2024.

[6] A. Deljouyi, R. Koohestani, M. Izadi, and A. Zaidman, "Leveraging large language models for enhancing the understandability of generated unit tests," in *ICSE*. ACM, 2024.

[7] C. Yang, J. Chen, J. Jiang, and Y. Huang, "Dependency-aware code naturalness," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 2355–2377, 2024.

[8] "Project homepage," 2025, https://github.com/chenyangyc/CLAST.

[9] K. Ahmed, M. Lis, and J. Rubin, "Slicer4j: a dynamic slicer for java," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1570–1574.

[10] 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.

[11] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, "The impact of test case summaries on bug fixing performance: An empirical investigation," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 547–558.

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

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

[14] C. Wei, L. Xiao, T. Yu, X. Chen, X. Wang, S. Wong, and A. Clune, "Automatically tagging the "aaa" pattern in unit test cases using machine learning models," *IEEE Transactions on Software Engineering*, vol. 49, no. 5, pp. 3305–3324, 2023.

[15] Y. Gao, Z. Wang, S. Liu, L. Yang, W. Sang, and Y. Cai, "Teccd: A tree embedding approach for code clone detection," in *2019 IEEE international conference on software maintenance and evolution (ICSME)*. ieee, 2019, pp. 145–156.

[16] S. Oh and S. Yoo, "Csa-trans: Code structure aware transformer for ast," *arXiv preprint arXiv:2404.05767*, 2024.

[17] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[18] 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.

[19] "Junit," 2024, https://junit.org/junit5/.

[20] Z. Zeng, Y. Wang, R. Xie, W. Ye, and S. Zhang, "Coderujb: An executable and unified java benchmark for practical programming scenarios," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 124–136.

[21] 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.

[22] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, "Code-aware prompting: A study of coverage-guided test generation in regression setting using llm," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 951–971, 2024.

[23] J. Chen, Y. Liang, Q. Shen, J. Jiang, and S. Li, "Toward understanding deep learning framework bugs," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–31, 2023.

[24] "Hugging face big code model leaderboard," 2024, https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard.

[25] B. Roziere, 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.

[26] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[27] "tree-sitter," 2024, https://tree-sitter.github.io/tree-sitter.

[28] "Deepseek," 2024, https://github.com/deepseek-ai/DeepSeek-Coder.

[29] "cobertura," 2024, https://github.com/cobertura/cobertura.

[30] "Pytorch," 2024, http://pytorch.org.

[31] "Transformers," 2024, https://github.com/huggingface/transformers.

[32] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.

[33] D. Spencer and T. Warfel, "Card sorting: a definitive guide," *Boxes and arrows*, vol. 2, no. 2004, pp. 1–23, 2004.

[34] Z. Wang, K. Liu, G. Li, and Z. Jin, "Hits: High-coverage llm-based unit test generation via method slicing," *arXiv preprint arXiv:2408.11324*, 2024.

[35] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[36] "Utgen repository," 2024, https://github.com/amirdeljouyi/UTGen.

[37] "Raggen repository," 2024, https://github.com/LeonYang95/LLM4UT.

[38] "Telpa repository," 2025, https://github.com/chenyangyc/TELPA.

[39] "Hugging face," 2024, https://huggingface.co.

[40] C. Yang, Z. Wang, Y. Jiang, L. Yang, Y. Zheng, J. Zhou, and J. Chen, "Reflective unit test generation for precise type error detection with large language models," in *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*, 2025.

[41] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 700–711.

[42] C. Yang, J. Chen, X. Fan, J. Jiang, and J. Sun, "Silent compiler bug de-duplication via three-dimensional analysis," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 677–689.

[43] J. Xuan and M. Monperrus, "Test case purification for improving fault localization," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 52–63.

[44] E. Soares, M. Ribeiro, R. Gheyi, G. Amaral, and A. Santos, "Refactoring test smells with junit 5: Why should developers keep up-to-date?" *IEEE Transactions on Software Engineering*, vol. 49, no. 3, pp. 1152–1170, 2022.

[45] A. Peruma, K. S. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "On the distribution of test smells in open source android applications: An exploratory study," 2019.

[46] K. Lucas, R. Gheyi, E. Soares, M. Ribeiro, and I. Machado, "Evaluating large language models in detecting test smells," *arXiv preprint arXiv:2407.19261*, 2024.

[47] Y. Gao, X. Hu, X. Yang, and X. Xia, "Context-enhanced llm-based framework for automatic test refactoring," *CoRR*, vol. abs/2409.16739, 2024.

[48] B. Zhang, E. Hill, and J. Clause, "Towards automatically generating descriptive names for unit tests," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 625–636.

[49] 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.

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