Reference-Based Retrieval-Augmented Unit Test Generation

ZHE ZHANG[1], XINGYU LIU[1], YUANZHANG LIN[1], XIANG GAO[1,2*], HAILONG SUN[1,2*], and YUAN YUAN[1,2],

[1]Beihang University, China and [2]Hangzhou Innovation Institute of Beihang University, China

Automated unit test generation has been widely studied, with Large Language Models (LLMs) recently showing significant potential. LLMs like GPT-4, trained in vast text and code data, excel in various code-related tasks, including unit test generation. However, existing LLM-based approaches often focus solely on the context within the code itself, such as referenced variables, while neglecting broader task-specific contexts, such as the utility of referring to existing tests of relevant methods in unit test generation. Moreover, in the context of unit test generation, these tools prioritize high code coverage, often at the expense of practical usability, correctness, and maintainability.

In response, we propose *Reference-Based Retrieval Augmentation*, a novel mechanism that extends LLM-based Retrieval-Augmented Generation (RAG) to retrieve relevant information by considering task-specific context. In the unit test generation task, for a given focal method, the *reference relationships* is defined as the reusability or referentiality of tests between the focal method and other methods. To generate high-quality unit tests for the focal method, the test reference relationships are then used to retrieve relevant methods and their existing unit tests. Specifically, we account for the unique structure of unit tests by dividing the test generation process into *Given*, *When*, and *Then* phases. When generating unit tests for a focal method, we retrieve pre-existing tests of other relevant methods, which can provide valuable insights for any of the *Given*, *When*, and *Then* phases. We implement this approach in a tool called *RefTest*, which sequentially performs preprocessing, test reference retrieval, and unit test generation, using an incremental strategy in which newly generated tests guide the creation of subsequent ones. We evaluated RefTest on 12 open-source projects with 1515 methods, and the results demonstrate that RefTest consistently outperforms existing tools in terms of correctness, completeness, and maintainability of the generated tests.

ACM Reference Format:
Zhe Zhang[1], Xingyu Liu[1], Yuanzhang Lin[1], Xiang Gao[1,2], Hailong Sun[1,2], and Yuan Yuan[1,2]. 2025. Reference-Based Retrieval-Augmented Unit Test Generation. 1, 1 (November 2025), 33 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Automated unit test generation has been extensively studied, resulting in a variety of approaches. Prominent methods include random-based strategies [11, 41, 60], search-based software testing (SBST) [9, 14, 27, 30, 34, 52, 64, 73] and constraint-driven techniques [17, 35]. Although existing approaches have achieved promising results, they still struggle with low code coverage, test case quality, and high computational costs [36]. Recently, deep learning-based approaches, such as

*Corresponding author.

Authors' Contact Information: Zhe Zhang[1], zhangzhe2023@buaa.edu.cn; Xingyu Liu[1], lxingyu@buaa.edu.cn; Yuanzhang Lin[1], yuanzhanglin@buaa.edu.cn; Xiang Gao[1,2], xiang_gao@buaa.edu.cn; Hailong Sun[1,2], sunhl@buaa.edu.cn; Yuan Yuan[1,2], yuan21@buaa.edu.cn,
[1]Beihang University, Beijing, China and [2]Hangzhou Innovation Institute of Beihang University, Hangzhou, China.

ACM XXXX-XXXX/2025/11-ART
https://doi.org/XXXXXXX.XXXXXXX

AthenaTest [56], have emerged, utilizing neural models to generate more diverse test inputs and better capture the functional intent of code [5, 14, 57, 67, 72].

Additionally, Large Language Models (LLM)-based techniques are becoming increasingly popular in automated testing [12, 20, 27, 65, 66]. Tools such as ChatUniTest [10] and ChatTester [69] leverage ChatGPT for unit test generation. Relying on proper prompt engineering and rich code context, LLM-based tools often outperform traditional methods like SBST in certain cases. Hybrid approaches [59, 65] combine the strengths of LLMs and traditional techniques to improve the quality of the tests. For instance, TELPA [65], uses program analysis with LLMs by integrating refined counterexamples into prompts, guiding the LLM to generate diverse tests for hard-to-cover branches, and HITS [59] decomposes focal methods into slices and asks the LLM to generate tests slice by slice, improving coverage for complex methods. Although producing great results, those tools still struggle to achieve high coverage and generate high-quality tests. By analyzing the underlying reasons, we have the following two main observations:

Observation 1: Focus on Code Itself in LLM-based Code Tasks, Overlooking Task-Specific Context. Existing LLM-based code generation tasks, particularly Retrieval-Augmented Code Generation (RACG), enhance the capabilities of LLMs by retrieving relevant code snippets or structures from code repositories. However, these approaches often focus solely on contextualizing the original code using various retrieval techniques, such as similarity-based retrieval (e.g. BM25 [25]), AST-based retrieval [71], graph-based retrieval [32], iterative file-based retrieval [70] and vector similarity-based retrieval [43]. Despite their effectiveness, these methods do not fully account for the inherent nature of the specific task at hand. In the case of unit test generation, when generating tests for a focal method, can we go beyond just the general code context? The broader context, including pre-existing test cases, can greatly improve the quality and relevance of the generated tests. This underscores the necessity for developing retrieval mechanisms that are specifically designed to support the nuances of different code tasks, thereby enriching the overall process of automated test generation.

Key question: How can we develop a retrieval mechanism that leverages task-specific information, such as considering existing test cases in the context of unit test generation, to enhance LLM-based code generation for specific tasks?

Observation 2: Focus on Code Coverage, Overlooking Quality of the Generated Unit Tests. Despite advancements in automated test generation, many tools remain focused on maximizing coverage metrics. However, a recent study by Yu et al. [68] reveals that practitioners value other attributes more. Although coverage is often emphasized in academic discussions, the study highlights that the correct rate (the proportion of generated test code that is syntactically correct, compilable, and runnable) and passing rate (the proportion of test cases that accurately reflect the requirements) are the most critical factors for adopting test generation tools in the real world. Moreover, the understandability (an aspect of auto-generated code that allows developers to quickly read and understand it) [18] and maintainability (developers can easily modify, extend, and adapt generated test cases over time) of the generated tests are also highly prioritized, reflecting the practical needs of developers in real-world settings. Furthermore, their study reveals that a majority of test generation tools (70%) focus predominantly on the code under test. However, this often does not fully align with developers' expectations, as these tools frequently overlook broader aspects of the codebase, such as method relationships and existing test cases.

Key question: How can we develop an approach that not only maximizes coverage but also produces unit tests that are correct, highly understandable, and maintainable?

Challenges. Addressing these key questions faces several core challenges: *(1) Defining and Identifying Effective Reference Relationships:* How to precisely define and identify relationships between methods that make one method's test cases useful references for another, and spanning diverse scopes (intra-class and inter-class). *(2) Capturing and Integrating Contextual Dependencies:* How to effectively capture essential contextual information and setups (e.g., class members, fixtures) embedded in existing test cases, and intelligently integrate these retrieved references with static code context into an optimal LLM prompt. *(3) Prioritizing and Maximizing Utility of Retrieved References:* How to effectively rank retrieved reference relationships and associated test cases, and then fully leverage these prioritized references.

Our Approach. In contrast to conventional approaches that focus solely on the code under test, our method incorporates a broader perspective by considering both the static context of the code under test (e.g., variables, function references) and **existing test cases** in the repository. We extend the concept of context retrieval for LLMs by defining *reference relationships*. Specifically, unit tests are usually structured into three key phases: *Given*, *When*, and *Then* [61]. For each phase, we search for methods that can provide valuable insights during test generation, based on the characteristics of the corresponding phase.

Let us consider a simple example of a reference relationship. Suppose we are generating a unit test for the `update` method of a HashMap, and there already exists a test case for the `insert` method (to add a new element). Although their specific functionalities differ, the test case of `insert` can still provide valuable references at each phase of the test generation of `update`. Using the Given-When-Then (GWT) structure [61], we illustrate how each phase can benefit from such references:

- **Given Phase:** This phase involves initializing objects and setting up inputs. The `insert` test case provides guidance on how to initialize the HashMap instance and prepare elements/keys as inputs. This setup logic is highly reusable for the `update` method, as both operations require a pre-existing structure and data for manipulation.
- **When Phase:** During this phase, the method under test is invoked. The `insert` test case demonstrates the typical invocation pattern for adding an element to the HashMap. This pattern, including how input data is passed, offers valuable reference for calling `update`, as both involve interacting with the HashMap object.
- **Then Phase:** This phase focuses on verification and assertions. While the final state (assertion) for `update` will differ from `insert` (e.g., `insert` asserts the presence of a new element, `update` asserts the change of an existing element), the `insert` test case can provide a reference for how to assert on the HashMap's state, such as checking its size, or verifying the content of specific elements after the operation.

Our proposed method systematically identifies such references during test generation for a focal method by looking for task-specific contexts like these. By abstracting this process through the definition of *reference relationships*, our approach enables the retrieval of task-specific contexts, enhancing the effectiveness of test case generation. Based on this idea, we propose RefTest, which first preprocesses the project by parsing the code and converting it into a relational schema stored in a *Metainfo Database*. This supports both retrieval and test generation. Next, RefTest analyzes existing test cases, extracts *test bundles*—the core reference units for generating new tests—and maps them to focal methods. Additionally, RefTest introduces a *Static Context Retrieval* mechanism to streamline the retrieval and generation processes. Then, RefTest executes two main steps:

- **Retrieval**. After preprocessing, RefTest performs reference retrieval. For a given focal method, it uses *reference analysis* to find methods with reference relationships within the class. It then

extends to class inheritance and interface implementation to perform *reference deduction*, expanding the search to the entire repository. The relevant methods and their *test bundles* are retrieved.
- **Generation**. RefTest ranks the retrieved methods and *test bundles* by relevance and confidence, combining them with the static context of the focal method. The most relevant components are then provided to the LLM for test case generation. Additionally, RefTest applies an *Incremental Strategy*, where newly generated test cases guide the generation of subsequent tests, maximizing the number of correct unit tests.

In Summary, we propose a novel approach, *Reference-Based Retrieval-Augmented Unit Test Generation*. This method defines *reference relationships* between methods, and extends traditional vector-similarity-based retrieval by incorporating code-specific relationships such as behavioral similarities, structural similarities, dependencies, and inheritance and interface implementations among classes. By leveraging these reference relationships and contextual information, our approach significantly improves the **correctness**, **completeness**, and **maintainability** of the generated unit tests. We evaluate RefTest on 12 open-source projects and compare its performance with the previous state-of-the-art LLM-based tool ChatUniTest, TestPilot, LLM with default RAG, as well as the traditional tool EvoSuite. Our results show that RefTest generated 821 fully covered unit tests for 1,515 focal methods, outperforming all other tools in both correctness and completeness. Further, we show that RefTest excels in maintainability, surpassing other tools including GitHub Copilot, in terms of code style and mock usage. Ablation studies confirm the effectiveness of the reference relationship definition and its decomposition into the Given-When-Then (GWT) framework, with the *Given* phase proving to be the most important. Deeper analysis of the patterns within the reference relationships, along with the assertion qualities, further show the strengths of RefTest.

Our contributions are summarised as follows:

- We propose a *Reference-based retrieval* approach for unit test generation. By decomposing the test generation process into the *GWT* phases and defining reference relationship and reference retrieval, we leverage existing test cases to generate unit tests for focal methods. This extends the traditional Retrieval-Augmented Generation (RAG) paradigm to incorporate task-specific features.
- We propose RefTest, which integrates preprocessing, retrieval, and generation to produce high-quality unit tests. Its incremental strategy further enhances this by leveraging newly-generated test cases, thereby maximizing the number of focal methods that can be covered and continuously improving test quality. The tool is open-source and available at: https://github.com/PAGTest-project/PAGTest.
- We conduct comprehensive evaluations of RefTest on 12 open-source projects with 1515 methods. The results show that RefTest consistently outperforms existing methods in terms of **correctness**, **completeness**, and **maintainability** of the generated tests.

2 Motivating Example

In this section, we first introduce an example of *reference relationship* that exists at the Class-Level. We then extend this relationship to the Repo-Level.

2.1 Class-Level Reference

To illustrate the motivations behind our approach, we use the `reset` method from the `RedissonLongAdder` class in the `Redisson` repository [47] as an example. `Redisson` is a widely used Java client for Redis, boasting over 23k stars on GitHub. We start by generating unit tests for the `reset` method.

```java
public RFuture<Void> reset() {
    String id = getServiceManager().generateId();
    RSemaphore semaphore = getSemaphore(id);
    RFuture<Long> future = topic.publishAsync(CLEAR_MSG + ":" + id);
    CompletionStage<Void> f = future.thenCompose(r -> semaphore.acquireAsync(r.intValue()))
                                    .thenCompose(r -> semaphore.deleteAsync().thenApply(res -> null));
    return new CompletableFutureWrapper<>(f);
}
```

Fig. 1. The method to be tested from the RedissonLongAdder class: reset method

The reset method is intended to set a counter to zero in a distributed system, where multiple instances might share the same counter. This operation guarantees that the reset is applied to all instances. It begins by generating a unique identifier (line 2) to fetch a semaphore, which is a synchronization tool (line 3). The method then sends a clear message to a topic, a communication channel, which returns a future representing the asynchronous operation (line 4). Upon successful message delivery, it acquires and deletes the semaphore (lines 5-6), ensuring a controlled reset. The entire process is encapsulated in a CompletableFutureWrapper, facilitating safe and efficient asynchronous handling, crucial for managing concurrency in distributed systems.

When using existing unit test generation tools [10, 69], they strive to search for the general context (e.g., the definition of variables and the invoked methods that are defined outside the method) of the focal method to better understand what the method does, thus helping to generate test cases. For example, when generating unit tests using GitHub Copilot [16], it looks up the source code of functions getServiceManager, generateId, and getSemaphore, determining where topic is defined and its type to have a better understanding of the method being tested. As shown in Figure 2, the unit test generated by GitHub Copilot first sets up the environment (lines 1-5) by mocking RedissonClient and creating RedissonLongAdder with mocked dependencies. Then, the test directly calls reset() (line 10) and verifies that the result of reset() is not null (line 14). Moreover, the test verifies that acquireAsync and deleteAsync can be correctly invoked. The test primarily focuses on mocking external objects and verifying their interactions.

However, when compiling this unit test, it fails due to the heavy reliance on mocked external services. Additionally, the test fails to validate the actual core functionality of the reset() method. The core functionality of reset() is to reset the internal state of the LongAdder, ensuring that any accumulated values are reset to zero. However, the generated unit test is focused primarily on confirming that external methods are invoked, without verifying whether the internal state reset is correctly applied.

```java
@BeforeEach
public void setUp() {
    redissonMock = mock(RedissonClient.class);
    redissonLongAdder = new
    ↪  RedissonLongAdder(mock(CommandAsyncExecutor.class),
    ↪  "test", redissonMock);
}
@Test
public void testReset() {
    RSemaphore semaphoreMock = mock(RSemaphore.class);
    ......
    RFuture<Void> result = redissonLongAdder.reset();

    verify(semaphoreMock).acquireAsync(anyInt());
    verify(semaphoreMock).deleteAsync();
    Assert.assertThat(result).isNotNull();
}
```

Fig. 2. The Unit Test for reset generated by GitHub Copilot

This raises an important question: *Is understanding the* reset() *method and looking up relevant external references sufficient to construct meaningful and effective unit tests*? In this case, the unit test lacks a deeper understanding of the method's internal logic and only focuses on superficial features, which leads to incomplete and ineffective test coverage for the method's core functionality.

Our Insight. Our tool RefTest focuses on identifying related methods that have a *reference relationship* with the focal method. If these related methods already have test cases, they can provide valuable guidance and reference during the test generation process for the focal method. Specifically, when generating unit tests for the focal method `reset`, through *reference analysis*, RefTest identifies a *reference relationship* between `reset` method and `sum` method from class `RedissonLongAdder`. Although their functionalities are semantically different, RefTest observes significant similarities in the GWT stages of their unit tests. The unit test of both methods are shown in Figure 3, their similarities are summarized as follows:

(1) **Given**: Both `reset` and `sum` require initializing the `RedissonLongAdder` class and configuring it appropriately before performing any operations.
(2) **When**: For `sum`, the method is invoked to calculate the cumulative sum of values added to the adder, while `reset` is invoked to reset the value back to zero. Despite these differing outcomes, the structure of invoking the method follows a similar pattern in both cases.
(3) **Then**: In both cases, the assertions verify the final state of the adder. For `sum`, the assertion checks that the value returned matches the expected sum, whereas for `reset`, it should ensure that the value is reset to zero, confirming the correct functionality of both methods.

When RefTest identifies the `sum` method and observes that its unit test already exists, it uses `sum`'s tests as a reference to generate tests for the `reset` method. By leveraging the structure of `sum`'s test, RefTest simplifies and enhances the generation process for `reset`, reusing patterns from `sum`. This allows the creation of a unit test (as shown in Figure 3b) with similar initialization, invocation, and assertion logic, significantly reducing the complexity of test generation.

```java
@Test
public void testSum() {
    var adder1 = redisson.getLongAdder("test1");
    var adder2 = redisson.getLongAdder("test1");

    adder1.add(2);
    adder2.add(4);

    assertThat(adder1.sum()).isEqualTo(6);
    assertThat(adder2.sum()).isEqualTo(6);
    adder1.destroy();
    adder2.destroy();
}
```

(a) The Unit Test of `sum`

```java
@Test
public void testReset() {
    var adder1 = redisson.getLongAdder("test1");
    var adder2 = redisson.getLongAdder("test1");
    adder1.add(2);
    adder2.add(4);
    adder1.reset();

    assertThat(adder1.sum()).isZero();
    assertThat(adder2.sum()).isZero();
    adder1.destroy();
    adder2.destroy();
}
```

(b) The Unit Test of `reset`

Fig. 3. Two examples of Redisson LongAdder methods demonstrating the reuse of patterns from the `sum` method's unit test to generate a unit test for the `reset` method. The figure illustrates the similarities in the initialization, invocation, and assertion phases of the `sum` and `reset` methods.

2.2 Repo-Level Reference

Further, in object-oriented programming, when we broaden our focus from an individual class to the entire repository, we can identify more extensive reference relationships:

- **Inheritance**: In cases of both concrete and abstract inheritance, methods between subclasses, parent classes, and sibling classes can be referenced when generating unit tests. The relationships between different classes across the inheritance hierarchy can also be leveraged to enhance the unit test generation process.
- **Common Interface Implementations**: When multiple classes implement the same interface, the test generation process for these classes can refer to each other. For instance, the shared

abstract behavior of two classes implementing a common interface can guide the generation of consistent and comprehensive unit tests.

To demonstrate the Repo-Level reference relationships, Figure 4 shows an example, where both class `RedissonAtomicLong` and `RedissonAtomicDouble` inherit from the base class `RedissonExpirable`. The pre-existing test `testGetAndSet`, shown in Figure 5a, from class `RedissonAtomicDouble` not only provides guidance for generating tests for other methods in the same class, such as `getAndDelete`, but also extends to its sibling class, `RedissonAtomicLong`. Specifically, in Figure 5b, the `testGetAndSet` from `RedissonAtomicDoubleTest` helps generate unit tests for the `getAndSet` method in `RedissonAtomicLong` and can also assist in generating tests for `getAndDelete` in `RedissonAtomicLong`.

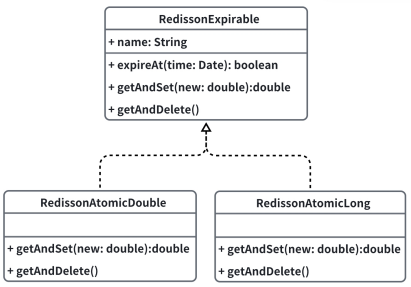

Fig. 4. An example of RedissonAtomicDouble and RedissonAtomicLong both implementing the abstract class RedissonExpirable.

```java
public class RedissonAtomicDoubleTest{
  @Test
  public void testGetAndSet() {
    RAtomicDouble al =
    ↪ redisson.getAtomicDouble("test");
    assertThat(al.getAndSet(12)).isEqualTo(0);
  }
  @Test
  public void testGetAndDelete() {
    RAtomicDouble al =
    ↪ redisson.getAtomicDouble("test");
    al.set(1);
    assertThat(al.getAndDelete()).isEqualTo(1);
    assertThat(al.isExists()).isFalse();
    RAtomicDouble ad2 =
    ↪ redisson.getAtomicDouble("test2");
    assertThat(ad2.getAndDelete()).isZero();
  }
}
```

(a) The Unit Test of `RedissonAtomicDouble`

```java
public class RedissonAtomicLongTest{
  @Test
  public void testGetAndSet() {
    RAtomicLong al =
    ↪ redisson.getAtomicLong("test");
    Assert.assertEquals(0, al.getAndSet(12));
  }
  @Test
  public void testGetAndDelete() {
    RAtomicLong al =
    ↪ redisson.getAtomicLong("test");
    al.set(10);
    assertThat(al.getAndDelete()).isEqualTo(10);
    assertThat(al.isExists()).isFalse();
    RAtomicLong ad2 =
    ↪ redisson.getAtomicLong("test2");
    assertThat(ad2.getAndDelete()).isZero();
  }
}
```

(b) The Unit Test of `RedissonAtomicLong`

Fig. 5. Unit tests for the `getAndSet` and `getAndDelete` methods in `RedissonAtomicDouble` and `RedissonAtomicLong`. The tests in `RedissonAtomicDoubleTest` can guide for generating tests in `RedissonAtomicLongTest` due to the structural and behavioral similarities of the methods.

Similarly, parent-child class relationships also exhibit the unit test reference effect. Specifically, shared methods between parent and child classes benefit from the mutual referencing of unit tests. Moreover, when two classes implement a common interface, methods shared by these classes can follow the same pattern of unit test reference as sibling classes. In this case, unit tests generated for methods in one class can serve as valuable references for generating unit tests for the corresponding methods in the other implementing class, just like the relationship between sibling classes.

In summary, these reference relationships extend across Class-Level and Repo-Level contexts. By considering these relationships, we can generate more accurate and effective unit tests for the focal method.

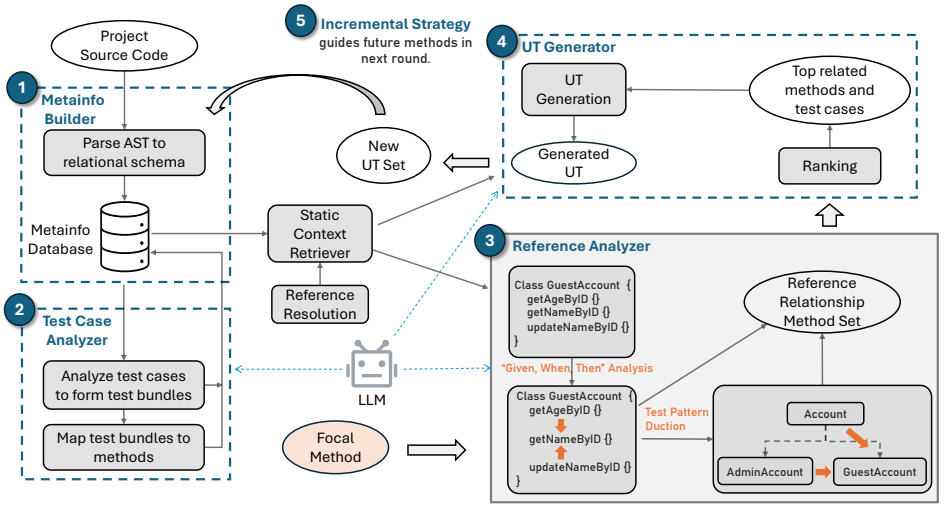

Fig. 6. The overall workflow of RefTest.

3 METHODOLOGY

We propose RefTest to address the problems and challenges previously outlined. The overall workflow of RefTest is illustrated in Figure 6.

First, the input project is processed by the `Metainfo Builder`, which parses the source code into an Abstract Syntax Tree (AST) and transforms it into a relational structure based on predefined schemas (e.g., Class, Method, Package). This structure is stored in the *Metainfo Database*, ensuring efficient data retrieval for the following stages. Next, existing test cases in the project are analyzed by the `Test Case Analyzer`. This process uses LLM to abstract the test cases into *Test Bundles*, which are then mapped back to their corresponding methods. Third, the `Reference Analyzer` takes the focal method and its class as input and performs reference resolution through static context retrieval. It conducts a "Given, When, Then" analysis using LLM to identify intra-class properties and uses reference deduction to retrieve related methods across the entire repository. These results form the *Reference Relationship Method Set*. Last, the `UT Generator` ranks the methods in the *Reference Relationship Method Set*, retrieves their corresponding test cases, and calls LLM to generate unit tests. The generated tests are added to the `UT Set`. After generating tests for all focal methods, the `Incremental Strategy` is applied. The newly generated tests are treated as historical tests for the next iteration to generate new unit tests.

3.1 Problem Formulation

At First, we formally define the problem of generating unit tests for focal methods in a code repository by leveraging existing test cases through reference relationships.

Let *Repo* be a code repository containing a set of classes $C = \{c_1, c_2, \ldots, c_k\}$. Methods in *Repo* are denoted as $M = \bigcup_{c \in C} M(c)$, where $M(c) = \{m_1, m_2, \ldots, m_p\}$ is the set of methods in class c. Some methods have associated test cases, while others do not. A set of existing test cases are given as $TC = \{tc_1, tc_2, \ldots\}$, each associated with some methods in M. A unit test UT_t for a method $m_t \in M$ consists of a set of test cases $TC_t = \{tc_1, tc_2, \ldots\}$, where each test case tc_i is designed to validate some aspect of the behavior of m_t. Given a focal method $m_t \in M$, our goal is to generate unit test(s) UT_t for m_t by:

(1) **Reference Definition:** Define a *reference relationship* $P \subseteq M \times M$, where $(m_t, m_r) \in P$ signifies that the method $m_r \in M$ is related to m_t for the purpose of unit test generation.
(2) **Reference Retrieval:** Define the operation $ReferenceRetrieval(m_t)$, which takes a focal method m_t as input and retrieves the set of related methods $R(m_t)$, representing the *reference relationship method set*, such that $(m_t, m_r) \in P$ for each $m_r \in R(m_t)$. $R(m_t)$ is composed of all methods that have a reference relationship with m_t, along with the specific description of the reference relationships (e.g., *Given*, *When*, *Then* phases).
(3) **Generation:** Utilize $R(m_t)$ and the existing test cases TC_r associated with methods in $R(m_t)$ to generate effective unit test(s) UT_t for the focal method m_t.

3.2 Preliminaries

In the context of reference relationship analysis and unit test generation using LLMs, three core problems must be addressed as they are foundational to effectively providing context to LLMs:

- **Where to find?** In reference relationship analysis and unit test generation, relevant code entities such as methods, classes, and interfaces must be efficiently located across large codebases, much like querying a relational database.
- **What to look for?** To let LLM understand a code snippet fully, especially for unit test generation, it's necessary to resolve references to external entities (e.g., methods, variables, or classes) that may not be defined locally in the code under analysis.
- **What information to provide?** In scenarios where the external information is too large to fit within the context window of LLM, we need to prioritize and compress the relevant code data while preserving essential context.

To tackle these challenges and provide LLMs with the essential code structure and context for accurate analysis and unit test generation, RefTest incorporates three core processes: *Metainfo Extraction*, *Lightweight Context-Aware Reference Resolution*, and *AST-Based Code Compression*.

3.2.1 Metainfo Extraction. The `Metainfo Builder` processes the input project by parsing the source code into an Abstract Syntax Tree (AST) and transforming it into a relational structure based on predefined schemas (e.g., Class, Method, Package). The extracted data is stored in the *Metainfo Database*, allowing for efficient retrieval of relevant information. Figure 7 illustrates structured metadata for the `MoveOperation` class, including details like its URI, methods, fields, superclasses, and additional documentation. Such structured organization of code data enables effective retrieval and supports tasks like reference relationship analysis and unit test generation. We also design schemas for other code entities, such as test classes, interfaces, abstract classes, and records.

```
{
  "uri": "src/main/java/org/redisson/transaction/
  operation/set/MoveOperation.java.MoveOperation",
  "name": "MoveOperation",
  "file_path": "src/main/java/org/redisson/transaction/
  operation/set/MoveOperation.java",
  "superclasses": "SetOperation",
  "methods": [
    "[void]commit(CommandAsyncExecutor)",
    "[void]rollback(CommandAsyncExecutor)",
  ],
  "class_docstring": "@author Nikita Koksharov",
  "original_string": "", // Omitted
  "super_interfaces": [],
  "fields": [
    {
      "docstring": "",
      "modifiers": "private",
      "marker_annotations": [],
      "type": "String",
      "name": "destinationName"
    }
  ]
}
```

Fig. 7. Simplified structured metadata for the `MoveOperation` class

3.2.2 Lightweight Context-Aware Reference Resolution. Accurate unit test generation by LLMs requires precise understanding of code snippets, which often include external references not defined locally. Traditional reference resolution, exemplified by language-specific tools like Jedi [23] (relied upon by RepoFuse [29] for Python autocompletion) or the Language Server Protocol (LSP) [38], are often either language-specific, or incur significant overhead due to their comprehensive program analysis. For efficient and scalable LLM-based code understanding across large codebases, a lightweight and context-aware reference resolution is essential. Our approach, named Lightweight Context-Aware Reference Resolution, addresses this challenge by extending the capabilities of Scope Graphs [39] and leveraging the Metainfo Database. Unlike Scope Graphs that typically focus on local variable definitions, our method identifies unresolved references across classes and files, completing lookups using the Metainfo Database. This allows us to provide LLMs with accurate, sufficient, and non-redundant context for arbitrary code snippets, which is crucial for subsequent reference analysis and robust code generation. This tailored design for LLM context management rather than a full program understanding.

Definition. A **Scope Graph** $G = (N, E)$ provides a structured representation of lexical scopes, declarations, and references in the program.

In this framework, nodes N represent various entities within the code. These include `LocalScope`, which corresponds to a lexical scope; `LocalDef`, representing a local definition such as a variable, function, or class; `LocalImport`, which refers to an import statement introducing external symbols; and `Reference`, denoting a usage of a symbol that must be resolved to its definition or import. Edges E, on the other hand, describe the relationships between these entities. These include `ScopeToScope`, which captures the parent-child relationship between scopes; `DefToScope`, linking a definition to the scope it belongs to; `ImportToScope`, connecting imports to the target scope; and `RefToDef` and `RefToImport`, which represent references to definitions or imports, respectively.

Reference Resolution via Scope Graphs and Metainfo Database. Given a code block cb, PAGTest performs reference resolution by constructing a graph and applying a multi-stage lookup process:

(1) **Scope Insertion:** Each lexical scope is inserted into the graph, with its parent-child relationships captured using `ScopeToScope` edges.
(2) **Declaration Insertion:** For every symbol declared (e.g., variable or function), a `LocalDef` node is added and linked to the appropriate scope.
(3) **Local Reference Resolution:** For each reference r in the cb, it is associated with its lexical scope s and inserted into the Scope Graph. A matching declaration is then searched for in the current scope s. If not found, the graph is traversed through `ScopeToScope` edges to parent scopes, recursively searching for the definition. This phase covers intra-file and local scope lookups.
(4) **Global Reference Resolution and Unresolved Detection:** If no declaration is found within the entire local scope chain, the reference r is marked as unresolved. For these unresolved references ($r \in UR$), RefTest extends the search by querying the *Metainfo Database* to retrieve its definition from other classes or files. All resolved references from this global lookup are aggregated into resolveRef(cb).

$$\text{resolveRef}(cb) = \{\text{QueryMetainfo}(r) \mid r \in UR\}$$

This hierarchical resolution process ensures that all necessary external contexts for any given code snippet are accurately identified and provided to the LLM.

3.2.3 AST-Based Code Compression. When retrieving relevant references for a code snippet, it is crucial to provide only the necessary information to fit within the LLM's context window. Hence, RefTest uses two mechanisms tailored for different scenarios: *Class Montage* and *Class Shrink*.

```java
public class RedisCommonBatchExecutor {
    private final Entry entry;
    private final BatchOptions options;
    public RedisCommonBatchExecutor() {
        // Constructor
    }
    private static int timeout(ConnectionManager
    ↪ connectionManager, BatchOptions options);
    private static int
    ↪ retryInterval(ConnectionManager
    ↪ connectionManager, BatchOptions options);
    private static int
    ↪ retryAttempts(ConnectionManager
    ↪ connectionManager, BatchOptions options);
    @Override
    protected void
    ↪ sendCommand(CompletableFuture<Void>
    ↪ attemptPromise, RedisConnection
    ↪ connection);
}
```

(a) RedisCommonBatchExecutor's Class Montage

```java
public class RedisCommonBatchExecutor {
    private final Entry entry;
    public RedisCommonBatchExecutor() {
        // Constructor
    }

    private static int
    ↪ retryInterval(ConnectionManager
    ↪ connectionManager, BatchOptions
    ↪ options) {
        // Implementation omitted
    }

    private static int
    ↪ retryAttempts(ConnectionManager
    ↪ connectionManager, BatchOptions
    ↪ options) {
        // Implementation omitted
    }
}
```

(b) RedisCommonBatchExecutor's Class Shrink

Fig. 8. Class Montage and Class Shrink of RedisCommonBatchExecutor: a comparative example illustrating two mechanisms for reducing class information. The left side shows Class Montage, which abstracts the class by including its signature and fields. The right side shows Class Shrink, which reduces the class by retaining only the methods and fields relevant to the focal method.

Class Montage. When analyzing test cases using LLM, it is necessary to map test methods to their corresponding target methods. Specifically, given a test method, it needs to trace through the call chain to find the final target function. In such cases, knowing just the method signatures of the class is sufficient. Additionally, when understanding code, it is often enough to know what a referenced class does without needing all the implementation details. Here, a simplified "montage" of the class is sufficient for certain scenarios. *Class Montage* abstracts implementation details by including only method signatures, fields, and inner classes while omitting method bodies. For example, in Figure 8a, the RedisCommonBatchExecutor class is reduced to its signatures and fields, offering a compact representation for LLM processing.

Class Shrink. After retrieving the *Reference Relationship Method Set* $R(m_t)$ through *reference-based retrieval*, providing the entire class to the LLM can be inefficient due to input length limitations and the risk of including irrelevant details. To address this, RefTest employs *Class Shrink*, which focuses on keeping only the methods in $R(m_t)$ and fields of the class, discarding everything unrelated. For example, as shown in Figure 8b, in the RedisCommonBatchExecutor class [1], only focal method retryAttempts and relevant method retryInterval are kept, while unrelated methods like sendCommand are discarded. This ensures that the LLM focuses on the relevant context without unnecessary information.

[1]https://github.com/redisson/redisson/blob/master/redisson/src/main/java/org/redisson/command/CommandBatchService.java

3.3 Test Case Analysis

When a relevant test case is identified and deemed suitable for reference, it is essential to provide not just the test case itself but also the full context in which it operates. This context includes the test case itself, its associated fixtures, any referenced variables (whether within the class or from external sources), mock objects, or other forms of dependencies.

Definition. Let c be a test class containing n test cases, m fixtures, as well as several imported modules, class variables, and functions. The test class c is formally defined as the combination of its test cases, fixtures, imports, and class members:

$$c = \{\text{tc}_1, \text{tc}_2, \ldots, \text{tc}_n\} + \{\text{Fixture}_1, \text{Fixture}_2, \ldots, \text{Fixture}_m\} + \text{Imports}(c) + \text{ClassMembers}(c)$$

Definition. Similarly, the Test Bundle set TB_r for a specific method m_r is defined as the collection of all test bundles related to m_r. Each test bundle $\text{tb}(c, i)$ for a test case tc_i in test class c is defined as the combination of the test case, its related fixtures, imports, and class members:

$$\begin{aligned}\text{tb}(c, i) = \text{tc}_i &+ \{\text{Fixture}_j \mid \text{Fixture}_j \text{ is used by } \text{tc}_i\} \\ &+ \{\text{Imports}(c) \mid \text{Imports}(c) \text{ is used by } \text{tc}_i\} \\ &+ \{\text{ClassMembers}(c) \mid \text{ClassMembers}(c) \text{ is used by } \text{tc}_i\}\end{aligned}$$

The set of test bundles is denoted as TB, including all $\text{tb}(c, i)$ for every test class c and test case tc_i.

The test case analysis process consists of two main steps. First, it performs test bundle abstraction, using static context retrieval and LLM-based analysis to generate the test bundles. Each test bundle encapsulates the relevant context for a given test case. Then, the test bundles are mapped to the corresponding methods based on the analysis results. Specifically, the input to the test bundle abstraction includes the test class and the *Class Montage* of the class being tested. The *Class Montage* helps the LLM identify the precise signature of the methods under test. This input is fed into the LLM, and the output is a structured JSON format, as shown in Figure 9. The "test_cases" field in the JSON stores the individual test bundles, each containing details of which method the test case targets, with its full context, including external dependencies, fixtures used, and project-specific resources.

This approach ensures that all relevant contexts, dependencies, and configurations influencing the test's behavior are fully accounted. By leveraging the *Test Bundle*, the use of test cases becomes more precise and context-aware.

```
{
    "file_path": "src/test/.../JCacheTest.java",
    "name": "JCacheTest",
    "dependencies": [
      "org.redisson.config.Config"
    ],
    "class_members": {
      "variables": [{ "name": "REDIS", "type":
      ↪  "GenericContainer<?>" }],
      "methods": [],
      "nested_classes": ["ExpiredListener"]
    },
    "fixtures": ["beforeEach"],
    "test_cases": [
      {
        "name": "testClose",
        "primary_tested": ["Cache.close()"],
        "external_dependencies": {
          "modules": ["Config"],
          "class_members": [{"variable": "REDIS"}]
        },
        "fixtures_used": ["beforeEach"],
        "project_specific_resources": [
            "TestUtil.logTestResult(String, int)"
        ]
      }
    ],
    ...
}
```

Fig. 9. Simplified test case analysis result for `JCacheTest`

3.4 Reference Retrieval

To address the *reference retrieval* challenge , we first define the reference relationship and then present the analysis of reference relationships within a class and across classes.

3.4.1 Reference Relationship Definition. A reference relationship between two methods m_A and m_B exists if m_A can provide reference information that aids in the unit test generation for m_B in any one or more of the three phases: *Given*, *When*, and *Then*. The reference relationship $P(m_A, m_B)$ is defined as:

$$P(m_A, m_B) \iff P_{Given}(m_A, m_B) \vee P_{When}(m_A, m_B) \vee P_{Then}(m_A, m_B)$$

The relations in ***Given*** phase involve similar preconditions, object initialization, setup, etc:

$$P_{Given}(m_A, m_B) \iff \text{The preconditions or setup of } m_A \text{ can be reused or referenced for } m_B$$

The relation in ***When*** phase refers to the similarities in terms of call sequences, parameters, dependencies, etc, which is defined as:

$$P_{When}(m_A, m_B) \iff \text{The method invocation patterns or dependencies of } m_A \text{ apply to } m_B$$

The relations in ***Then*** phase involves similar asserting the expected results:

$$P_{Then}(m_A, m_B) \iff \text{The assertions or exception handlings from } m_A \text{ are relevant for } m_B$$

In cases where all three conditions are met:

$$P_{Complete}(m_A, m_B) \iff P_{Given}(m_A, m_B) \wedge P_{When}(m_A, m_B) \wedge P_{Then}(m_A, m_B)$$

3.4.2 Class-Level Reference Analysis (Intra-Class). Class-Level Reference Analysis focuses on identifying reference relationships between methods within the same class, forming the foundation for extending the analysis to cross-class relationships. Given a focal method m_t in class c_t, the analysis examines other methods within the class $M(c_t) = \{m_1, m_2, \ldots, m_n\}$, including any inherited methods from parent classes, which are recursively incorporated via the *MetaInfo Database*. The goal is to find methods m_r with reference relationships to m_t across the *Given*, *When*, and *Then*.

Algorithm 1: Reference Relationship Identification for a Focal Method

Input: Focal method: m_t; Class of m_t: C_t; Static Context: SC; reference analysis prompt template: $P_{template}$;
Output: Reference relation method set: $R(m_t)$ (structured JSON)

1 $R(m_t) \leftarrow \emptyset$ `// Initialize empty reference relationship set`

2 /* Assemble LLM prompt for reference analysis */
3 $P_{input} \leftarrow m_t$ and its code context (including C_t and SC);
4 $M_{candidate} \leftarrow$ Methods in C_t (including inherited/implemented);
5 **foreach** $m_i \in M_{candidate}$ **do**
6 $\quad P_{input} \leftarrow P_{input} \cup \{m_i$ and its code context$\}$;
7 $Prompt \leftarrow P_{template} \cup P_{input}$;

8 /* Call LLM to identify reference relationships */
9 $LLM_{output} \leftarrow$ `CallLLM`$(Prompt, LLM_{prop})$;
10 $R(m_t) \leftarrow$ ParseJson(LLM_{output});
11 **return** $R(m_t)$;

As illustrated in Algorithm 1, the process of identifying reference relationships begins with assembling a comprehensive prompt for the LLM. This prompt (Line 7) integrates the focal method's code context (m_t, its class C_t, and static context SC), along with the code contexts of all candidate methods ($m_i \in M_{candidate}$, including inherited/implemented methods). This structured input guides the LLM to perform the reference analysis.

The core of the analysis, as detailed in Figure 10 (which depicts the formal prompt template), lies in leveraging *Rule-based Reasoning* [50] and *Chain of Thought (CoT) reasoning* [8]. Instead of directly identifying all reference relationships for a method m_t at once, which can confuse the LLM due to complex inter-method dependencies, RefTest decomposes the problem into smaller

manageable parts. Specifically, the LLM is instructed to first imagine a set of expected test cases $TC(m_t)$ for m_t by reasoning. Then, it traverses all candidate methods m_i, imagines their respective test cases $TC(m_i)$, and compares $TC(m_t)$ with $TC(m_i)$ to identify potential reference relationships in the *Given*, *When*, and *Then* phases. If a relationship is found in any phase, the method m_i and its corresponding description are added to the *gwt set*; if relationships are found in all three phases, they are added to the *complete set*. Finally, the LLM outputs the structured JSON representation of these sets, forming the *Reference Relationship Method Set* $R(m_t)$.

// Task Description
Given focal method m_t , its class c_t and definition of reference relationship $P(m_i, m_t)$, find a set of m_i and corresponding test

// Steps
1. Expect test cases for m_t : $TC(m_t)$
2. For m_i in $M(c_t)$:
  - Expect test cases for m_i: $TC(m_i)$
  - Check $TC(m_t)$ and $TC(m_i)$:
    $P_{Given}(m_i, m_t)$: Preconditions or setup of m_i can give insights to m_t
    $P_{When}(m_i, m_t)$: Invocation patterns similar to m_t
    $P_{Then}(m_i, m_t)$: Assertions or exception handling relevant to m_t
  - If $P_{Given}(m_i, m_t) \vee P_{When}(m_i, m_t) \vee P_{Then}(m_i, m_t)$:
      Add m_i and description of P to *gwt set*
  - If $P_{Given}(m_i, m_t) \wedge P_{When}(m_i, m_t) \wedge P_{Then}(m_i, m_t)$:
      Add m_i and description of P to *complete set*

// Input and Output
- Input: m_t, resolveRef(m_t) and its class c_t(with methods inherited from parents).
- Output: *gwt* set and *complete* set

Fig. 10. Formal version of the prompt for analyzing properties.

The generated JSON output provides a clear mapping of reference relationships between methods. As shown in Table 1, it highlights the methods that share reference relationships with `removeFirst()`, along with corresponding descriptions and confidence scores. Specifically, the methods `poll()`, `remove()`, and `getFirst()` offer valuable references to `removeFirst()` across all three phases *Given*, *When*, and *Then*—thus they are categorized under **Complete**. On the other hand, methods such as `initializeQueue()`, `validateQueueState()`, and `verifyRemoval()` provide specific references in the *Given*, *When*, and *Then* phases respectively.

Furthermore, Table 1 also shows their respective confidence scores with the focal method. These scores represent the confidence assigned by the LLM based on the degree of reference in each relationship, with higher scores (such as `poll()` at 0.90) indicating a closer functional similarity to `removeFirst()`. Lower scores, like the 0.70 for `validateQueueState()`, suggest a more indirect but still relevant relationship, such as ensuring the queue is not empty before calling `removeFirst()`. It is important to note that the relevance of reference relationships is not confined to methods with semantic similarity. For instance, even methods that perform semantically opposite tasks, such as *encode* and *decode*, can share a common test case structure across all three phases (*Given*, *When*, and *Then*) and thus have reference relationships that can enhance test case generation.

Table 1. Reference Relationships and Enhancements for `removeFirst()`

Phase	Method Name	Reason	Confidence	External
Complete	`poll()`	Similar control flow for removing the first element asynchronously.	0.90	No
	`remove()`	Internally calls `removeFirst()`, making both interchangeable.	0.85	No
	`getFirst()`	Accesses the first element but does not remove it.	0.80	No
Given	`initializeQueue()`	Queue initialization is needed to ensure proper configuration.	0.80	No
When	`validateQueueState()`	Ensures the queue is not empty before calling `removeFirst()`.	0.70	No
Then	`verifyRemoval()`	Confirm the first element was successfully removed from the queue.	0.85	No

In a word, this approach breaks down the task into three more manageable components, each focusing on one of the three phases of unit test generation. By guiding the LLM to explore each phase separately, RefTest enables it to incrementally uncover reference relationships.

3.4.3 Repo-Level Reference Analysis (Inter-Class). *Repo-Level Reference Analysis* extends reference analysis across class boundaries by considering relationships such as inheritance, sibling classes, and common interface implementations. These relationships introduce opportunities for cross-class reference relationships, where test cases for methods in one class can provide reference information for generating test cases for methods in another class.

General Reference Relationship Formulation. Let m_A, m_B be methods. Let C_1, C_2 be classes related by inheritance, sibling, or common interface implementation. Let M_{shared} be the set of methods shared between C_1 and C_2 (i.e., $M_{shared} = M(C_1) \cap M(C_2)$). If $P(m_A, m_B)$ denotes that method m_A provides reference information for method m_B's unit test generation, then the general condition for extending this reference relationship across classes is as follows:

$$(P(m_A, m_B) \land m_A \in M_{shared} \land m_A \in M(C_1) \land m_B \in M(C_1)) \implies P(m'_A, m_B)$$

Here, m'_A denotes the method m_A when its implementation is considered from the context of class C_2 (i.e., $m'_A \in M(C_2)$). This means if m_A enhances m_B when m_A is in C_1, then m'_A can also enhance m_B.

Inheritance Relationships. For a parent class C_p and its child class C_c, $M_{shared} = M(C_p) \cap M(C_c)$. If $P(m_A, m_B)$ holds where $m_A \in M_{shared}$ and $m_B \in M(C_c)$, then $P(m'_A, m_B)$ also holds where $m'_A \in M(C_p)$. The relationship is bidirectional, allowing m_A from C_p to enhance m_B in C_c, and vice versa, following the general formula.

Sibling Class Relationships. For sibling classes C_1 and C_2 (sharing a common parent), $M_{shared} = M(C_1) \cap M(C_2)$. If $P(m_A, m_B)$ holds where $m_A \in M_{shared}$ and $m_A, m_B \in M(C_1)$, then $P(m'_A, m_B)$ also holds where $m'_A \in M(C_2)$, following the general formula.

Common Interface Implementations. For classes C_1 and C_2 implementing the same interface I, $M_{shared} = M(C_1) \cap M(C_2) \cap M(I)$. If $P(m_A, m_B)$ holds where $m_A \in M_{shared}$ and $m_A, m_B \in M(C_1)$, then $P(m'_A, m_B)$ also holds where $m'_A \in M(C_2)$, following the general formula.

Consolidation. In all cases, cross-class reference relationships leverage shared methods (M_{shared}). If $P(m_A, m_B)$ holds for $m_A \in M_{shared}$ in one class, then $P(m'_A, m_B)$ (with $m'_A \in M(\text{related class})$) also holds, ensuring reference relationships extend across class boundaries. Additionally, the "External" field (Section 3.4.2) indicates when a related method stems from an external class, further facilitating inter-class reference analysis.

3.5 Unit Test Generation

In this subsection, we introduce the approach to generating effective unit tests for the focal method, solving *Generating Effective Unit Tests* challenge.

3.5.1 Reference-Based Retrieval and Ranking. Here, we explain the process of generating unit tests for the focal method m_t by leveraging reference relationships and a ranking strategy.

Algorithm 2: Unit Test Generation for a Focal Method

Input: Focal method: m_t; Reference relationship method set: $R(m_t)$; Existing test bundle set: TB; Reference-based retrieval prompt: *reference_prompt*; Fallback prompt: *fallback_prompt*;
Output: Generated unit test ut_t for m_t

1 /* *Initialization* */
2 $S_{origin} \leftarrow \emptyset$ `// Store related methods with test bundles and reference description to` m_t
3 $S_{ranked} \leftarrow \emptyset$ `// Store ranked methods with test bundles and reference relationship to` m_t
4 $D \leftarrow \{\}$ `// Dictionary (ranked method → test bundles)`
5 $SC \leftarrow \emptyset$ `// Static context`

6 /* *Rank* */
7 **foreach** $m_r \in R(m_t)$ **do**
8 **if** $TC_r \neq \emptyset$ **then**
9 $S_{origin} \leftarrow S_{origin} \cup \{m_r\}$;
10 $S_{ranked} \leftarrow$ `rank`(S_{origin}) // *intra-class complete > intra-class GWT > inter-class complete > inter-class GWT*

11 /* *Assembling prompt, input, and context* */
12 $SC \leftarrow SC \cup$ `ResolveRef`(m_t); `// Resolve reference for` m_t
13 **if** $S_{ranked} \neq \emptyset$ **then**
14 $S_{\text{intra}}, S_{\text{inter}} \leftarrow$ Split S_{ranked} into intra-class and inter-class methods;
15 **if** $S_{intra} \neq \emptyset$ **then**
16 $SC \leftarrow SC \cup$ `ClassShrink`(m_t, S_{intra});
17 **foreach** $m_r \in S_{inter}$ **do**
18 $SC \leftarrow SC \cup$ `QueryMetainfo`(m_r);
19 **foreach** $m_r \in S_{ranked}$ **do**
20 $D \leftarrow D \cup \{m_r \rightarrow TB_r\}$;
21 $D_m \leftarrow$ `MinimizeTestBundle`(D); `// Remove redundancy`
22 $I \leftarrow$ *reference_prompt* $+ m_t + SC + S_{\text{ranked}} + D_m$;
23 **else**
24 $I \leftarrow m_t + SC +$ *fallback_prompt*;

25 $ut_t \leftarrow$ `CallLLM`(I)

Algorithm 2 outlines the process for generating a unit test. The inputs are the focal method m_t, the reference relationship method set $R(m_t)$ (from the *reference analysis*), and the test bundle set TB. RefTest first obtains methods with existing test cases in $R(m_t)$ to create S_{origin} (line 7 - 9), then ranks them to form S_{ranked} (line10). It prioritizes methods at the class level over those at the repository level as inter-class relationships are weaker than intra-class due to variations in method implementations. Within each level, *complete* relationships are selected first, with up to N methods chosen. Afterward, for the *Given*, *When*, and *Then* phases, RefTest selects up to N methods per phase, ensuring at least one per phase if available. The final ranking order is: *intra-class complete* > *intra-class GWT* > *inter-class complete* > *inter-class GWT*, with a maximum of $4N$ methods selected for test generation. The default N is three, but it can be adjusted by the user.

After ranking, RefTest begins packing the context. First, based on *Reference Resolution Using Scope Graph and Metainfo Database*, RefTest call resolveRef(m_t) to resolve references, ensuring that all external references used by m_t are resolved (line 12). Next, S_{ranked} is split into intra-class methods S_{intra} and inter-class methods S_{inter} (line 14). For S_{intra}, *Class Shrink* is applied to remove redundancy, and the resulting context is added to SC (lines 15-16). For each method m_r in S_{inter}, RefTest queries the metainfo database (`QueryMetainfo`) to get the detailed information of m_r and add them to SC (lines 17-18). Simultaneously, RefTest gets all the test bundle of methods in S_{ranked}, storing them in D and minimizes D to D_m to reduce redundancy (lines 19-21). The final input to the LLM, I, consists of the reference prompt, m_t, SC, S_{ranked}, and D_m. If S_{ranked} is empty, a fallback

strategy is applied to generate unit tests directly for the focal method (lines 23-24). In this case, the final input to the LLM is m_t, SC, along with the fallback prompt. The `CallLLM` function is invoked with I, and the generated unit test ut_t is returned (line 25).

Following previous work [19, 31], RefTest incorporates a repair mechanism where test cases generated in the first round are corrected in subsequent iterations. The faulty test case, along with the focal method and its static context, is provided to the LLM for refinement in the next round.

3.5.2 Incremental Strategy. Having explained the approach to generating unit tests for a focal method, this subsection outlines the incremental strategy to optimize test generation across the entire project.

The strategy prioritizes focal methods whose relevant methods already have associated test cases. Focal methods without such references are temporarily skipped and revisited in subsequent rounds. Newly generated tests from earlier rounds serve as references for focal methods in later rounds. This ensures that each focal method benefits from the most relevant, existing unit tests whenever possible. For $m_A \in M$, if there exists a reference relationship $P(m_A, m_B)$ with a related method m_B, where m_B already has an existing test case tc_B, formalized as:

$$C(m_A, m_B) = P(m_A, m_B) \wedge TC_B \neq \emptyset$$

During each round, newly generated test cases tc_A are added to the overall test set TC. Once all applicable focal methods are processed in the current round, the newly generated test cases undergo *Metainfo Extraction* and *Test Case Analysis* to update the reference information. The process then proceeds to the next round, generating unit tests for additional focal methods that satisfy $C(m_A, m_B)$. This process continues, aiming to generate unit tests for as many focal methods as possible using the Reference-Based Retrieval Augmentation approach. Each round introduces new unit tests that expand the number of focal methods covered until no further unit tests can be generated for additional methods. For focal methods that still lack relevant unit tests to reference at last, a fallback prompt is used to generate the unit tests.

4 EVALUATION

We implement RefTest using 8,000 lines of Python code, and we evaluate RefTest and answer the following research questions:

- **RQ1 (Correctness and Completeness)**: How effective is RefTest in generating unit tests for a given focal method in terms of correctness and completeness?
- **RQ2 (Maintainability)**: How well is the maintainability of the generated unit tests?
- **RQ3 (Ablation Study)**: How does the *Given-When-Then (GWT) Strategy*, *Incremental Strategy* contribute to the performance of RefTest?
- **RQ4 (Patterns)**: What patterns exist in *reference relationships*, and what common characteristics can be identified among them?
- **RQ5 (Assertion Quality)**: How do the generated assertions perform in terms of complexity and usefulness beyond simple error rates?

4.1 Experiment Setup

4.1.1 Datasets. We build an experimental benchmark following the criteria outlined in ChatUnitest[10]. We utilize 4 projects from the datasets provided by HITS [59] and ChatUnitest[10], excluding all projects that have been previously archived. We additionally crawl 8 repositories from GitHub, bringing the total number of projects to 12. Projects were required to have at least 150 stars to ensure community interest and were updated within the last month to guarantee active maintenance. The dataset encompasses a variety of domains, including utilities, parsers (e.g., HTML or expression

parsers), and network protocols. For method selection, private methods and those containing only a single line of effective code are excluded. For methods involving nested or anonymous inner classes, only the outermost method is considered for testing. These criteria ensure the practicality of the dataset. Ultimately, unit tests are generated for 1,515 focal methods across 12 projects.

4.1.2 Baseline and Configuration. Previous SOTA tools like `ChatUniTest` [10], `TestPilot` [49] and `HITS` [59] are designed for unit test generation. However, during our experiments, the current open-source implementation of `HITS` encounters significant runtime issues, making it unusable for generating unit tests at the method level. `TestPilot` is for JavaScript that enhances prompts with usage examples from documentation and we re-implemented it's core strategy in Java for comparison. Additionally, we include a baseline LLM using traditional Retrieval-Augmented Generation (RAG). Evosuite is a prominent search-based software testing (SBST) tool, is included as another baseline. For the LLM, we use `DeepSeek-V2.5` [74], which offers an optimal balance between cost and performance. DeepSeek-V2.5 has been widely recognized and exceeds GPT-4 specifically in terms of coding tasks [6]. Additionally, we include EvoSuite as another baseline in our comparison. The specific experimental parameters are as follows:

- **Configuration of LLM-Based Tools:** For each LLM-based tool, a focal method is specified before execution. The tool first performs one generation round to produce an initial unit test. Subsequently, it is allowed up to two repair rounds to refine and improve the generated test. Each tool produces one unit test file per focal method, which may include up to N test cases. For `ChatUniTest`, we use its default configuration, which generates 5 test cases per focal method in a single file. The baseline LLM receives the source code of the containing class as context, utilizing the default Retrieval-Augmented Generation (RAG) strategy. Additionally, the prompt is adapted from ChatTester [69], a proven effective approach for Java unit test generation. We don't directly use ChatTester in our comparison due to its source code being unavailable. For RefTest, when generating unit tests for the focal method, it relies solely on reference retrieval to identify methods and test cases with reference relationships to the focal method. The existing test cases of focal method are ensured to be hidden during the generation process, maintaining the integrity and fairness of the evaluation.
- **Configuration of EvoSuite:** For EvoSuite [15], we use its latest runtime version 1.0.6, which supports only Java 8 and JUnit 4. Consequently, out of the 12 selected projects, only 3 are compatible with EvoSuite, resulting in tests being generated for 389 methods out of the total 1515. EvoSuite's search budget (time limit) is set to 180 seconds, with 60 seconds allocated for the initial test generation and 60 seconds for each of the two repair rounds. This allocation mirrors the two repair rounds allowed for LLM-based tools. All other parameters remain at their default settings.

4.2 Correctness and Completeness

To evaluate the correctness and completeness of the generated unit tests, we compare RefTest with state-of-the-art (SOTA) solutions.

4.2.1 Metrics. Previous approaches typically generate all test cases in a single batch and report overall project coverage[10, 59]. However, this does not ensure that the number of unit tests generated by each tool is consistent, leading to potential unfairness in comparisons. In practice, developers generally prefer to generate a unit test for a specific focal method each time they use the tool[68]. To ensure a fair comparison, our strategy involves providing one focal method as input at a time and generating a corresponding test file specifically for that focal method. This test file,

Table 2. Unit Test Generation Results Across Tools

Metric	RefTest	ChatUniTest	TestPilot	LLM (default RAG)	EvoSuite
Compilation and Run Errors (↓)	**393 (25.9%)**	923 (60.9%)	840 (55.4%)	1155 (76.2%)	232 (59.6%)
Assert Errors (↓)	210 (13.9%)	228 (15.0%)	216 (14.3%)	**127 (8.4%)**	53 (13.6%)
Successful Execution (↑)	**912 (60.2%)**	364 (24.0%)	459 (30.3%)	233 (15.4%)	104 (26.7%)
Fully Covered (↑)	**821 (54.2%)**	323 (21.3%)	431 (28.4%)	200 (13.2%)	99 (25.4%)
Total Methods Tested	1515	1515	1515	1515	389

*Note: **Bold** values indicate the best performance. The arrows (↑/↓) denote whether a higher or lower value is better.*

treated as a unit test, contains multiple test cases. We then execute only this test file to measure the code coverage of the focal method. We use `JaCoCo` [2] to calculate the coverage for the focal method.

The execution results are categorized as follows:

- **Compilation and Runtime Errors**: Compilation errors may arise from issues like undeclared variables, while runtime errors can include execution problems like uncaught exceptions. These indicate that at least one test case in the generated unit test for the focal method encounters the corresponding issue.
- **Assertion Errors**: These refer to assertion errors where the expected value does not match the actual value. To measure the effectiveness of the test generation tools, we assume the focal method is correctly implemented. Like compilation and runtime errors, this indicates that at least one test case encounters the issue.
- **Successful Execution**: This means all test cases in the generated unit test run successfully. Additionally, **Fully Covered** indicates that the unit test runs successfully and achieves 100% line and branch coverage for the focal method, demonstrating both the correctness and completeness of the generated unit tests.

4.2.2 Results. Table 2 shows the results of evaluated tools. In terms of **correctness**, RefTest demonstrates significantly fewer compilation and runtime errors (**25.9%**) compared to ChatUnitTest (60.9%), TestPilot (55.4%), LLM with default RAG (76.2%), and EvoSuite (59.6%). It also has the highest rate of successful executions at 60.2%, outperforming ChatUnitTest (24.0%), TestPilot (30.3%), the LLM with default RAG (15.4%), and EvoSuite (26.7%). These results highlight the higher reliability of test cases generated by RefTest. In terms of **completeness**, RefTest achieved a Fully Covered rate in **54.2%** of cases, outperforming ChatUnitTest (21.3%), TestPilot (28.4%), the LLM with default RAG (13.2%), and EvoSuite (25.4%). This demonstrates RefTest 's superior ability to generate comprehensive tests that provide broader coverage of the code under test.

Figure 11 compares the rates of the fully covered method for each project. RefTest consistently achieves higher rates across nearly all projects. For example, in `binance-connector-java`, RefTest achieves 53.3%, significantly outperforming ChatUniTest (18.1%), TestPilot (25.2%), and LLM with default RAG (11.7%). While TestPilot often showed stronger performance than ChatUniTest and LLM (default RAG) on projects with rich documentation (e.g., `commons-collections` and `jsoup`), RefTest consistently achieved even higher rates. This superiority stems from RefTest's ability to leverage existing tests beyond documentation-mined usage examples, providing a more comprehensive context for LLM-based test generation. An interesting observation is that projects with abstract or interface-heavy code, such as `datafaker`, posed significant challenges for all tools, leading to lower fully covered rates primarily due to mistaken attempts to instantiate abstract classes. Despite these general difficulties, in the `ice4j` project with its complex network protocols,

[2]https://github.com/jacoco/jacoco

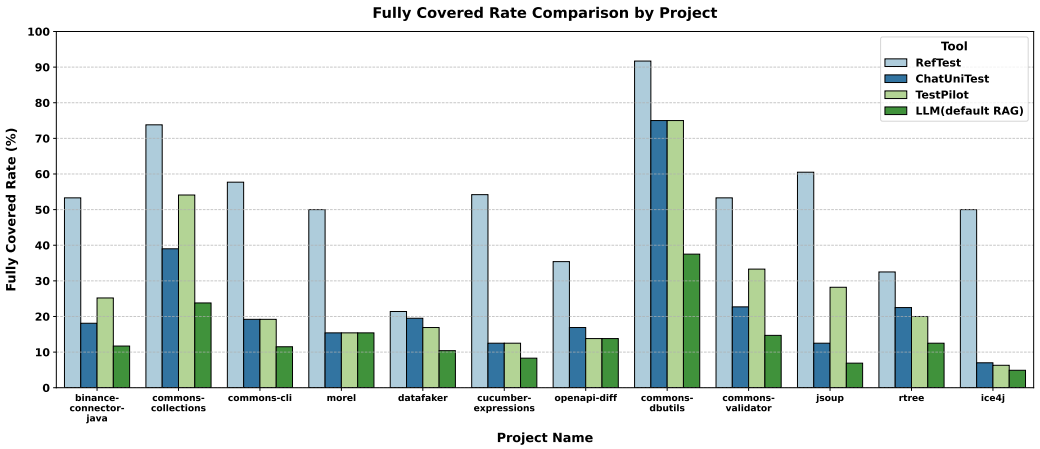

Fig. 11. Project-wise Fully Covered Rate Across Tools

Table 3. Fully Covered and Successful Execution Results Compared with Existing Tests

Tool	RefTest	ChatUniTest	TestPilot	LLM (default RAG)	EvoSuite	Existing Tests
Fully Covered	**821**	323	431	200	99	398
Successful Execution	**912**	364	459	233	104	453

Note: RefTest does not refer to existing tests of the focal method when generating unit tests.

RefTest still achieved a 50.0% Fully Covered rate, significantly outperforming all other baselines. This highlights the value of incorporating existing tests when generating new tests in projects with intricate dependencies and protocols.

To compare with existing unit tests, we evaluate their performance based on the Completeness metric, focusing on the number of tests that fully cover the focal method among those that were successfully executed. Among the 1,515 focal methods, 453 were identified as having existing unit tests. The results are shown in Table 3. The table highlights that RefTest successfully generates **821 fully covered unit tests** (out of 912 successful executions), demonstrating superior completeness. This significantly outperforms ChatUniTest (323/364), TestPilot (431/459), LLM with default RAG (200/233), and EvoSuite (99/104). This shows that RefTest produces a higher number of complete tests compared to both compared tools and the existing unit tests.

Furthermore, we analyze the overlap of fully covered methods across the evaluated tools, which is shown in Table 4. RefTest uniquely covered **374** methods. RefTest also covered 318 methods shared with ChatUniTest, 423 methods with TestPilot, 199 methods with LLM with default RAG, and 97 methods with EvoSuite. Other tools showed limited unique contributions: ChatUniTest covered 3 unique methods, TestPilot 4, LLM with default RAG 0, and EvoSuite 2. This indicates that RefTest subsumes the coverage capabilities of the baseline tools while providing broader overall coverage.

Table 4. Overlap Analysis of Fully Covered Methods

Category	Count
Overlap with RefTest	
vs. ChatUniTest	318
vs. TestPilot	423
vs. LLM (default RAG)	199
vs. EvoSuite	97
Unique Coverage	
RefTest (Unique)	**374**
ChatUniTest (Unique)	3
TestPilot (Unique)	4
LLM (default RAG) (Unique)	0
EvoSuite (Unique)	2

To further evaluate the efficiency of all tools, we analyze the average time taken per focal method and the number of test cases generated. As shown in Table 5, RefTest demonstrates the fastest **per-focal method generation time**, with an average of 42 seconds, significantly

outperforming EvoSuite (60 seconds), TestPilot (45 seconds), ChatUniTest (70 seconds), and LLM with default RAG (52 seconds). In terms of the average number of test cases generated, EvoSuite produces the most with 14.4 test cases per method. ChatUniTest generates an average of 5.0 test cases per method, followed by TestPilot (4.3), LLM with default RAG (3.8), and RefTest (3.7). RefTest's ability to generate a smaller yet highly effective set of test cases in significantly less time demonstrates its efficiency.

It is important to clarify that the **Generation Time** in Table 5 represents the time taken to generate tests for an individual focal method, after the initial project setup. Beyond this per-method cost, RefTest involves an initial preprocessing cost per project for building its Metainfo Database. This setup primarily comprises Metainfo Extraction (averaging 10 seconds per project using tree-sitter) and Test Case Analysis. While Test Case Analysis involves LLM calls averaging 6 seconds per test bundle, this process is highly parallelizable based on LLM concurrency settings. To accommodate evolving source code, this database can be efficiently updated by re-running the analysis, which adds minimal overhead. For the 12 open-source projects in our dataset, the average total initial preprocessing time is approximately **30 seconds per project**. This upfront and subsequent maintenance cost is efficiently managed and amortized across the entire project's test generation lifecycle.

Table 5. Average Per-Focal Method Generation Time and Test Cases Across Tools

Tool	Generation Time	Test Cases
RefTest	**42s**	**3.7**
ChatUniTest	70s	5.0
TestPilot	45s	4.3
LLM (default RAG)	52s	3.8
EvoSuite	60s	14.4

Overall, the results confirm that RefTest not only produces test cases with higher correctness by reducing errors and increasing successful executions but also ensures greater completeness by achieving Fully Covered Rate in a significantly larger proportion of cases. These findings underscore the effectiveness of RefTest in generating reliable, high-coverage unit tests.

4.3 Maintainability

To address this research question, we evaluate the maintainability of unit tests that passed and fully covered using two primary metrics: **Code Style Violations** and **Mock Density**.

4.3.1 Settings. We opt not to evaluate the tests produced by EvoSuite due to its strong reliance on custom mock libraries and scaffolding inheritance. Additionally, the test cases generated by EvoSuite often suffer from non-standard naming conventions (e.g., "test0", "test1"), which further detracts from their usability. Instead, we choose GitHub Copilot for our comparison, noting that Copilot was not included in RQ1 due to its lack of support for automated invocation, and manually generating tests for 1,515 methods would be impractical. We use 183 unit tests generated for methods that belong to the intersection of those fully covered methods by all four LLM-based tools. These tests are integrated back into their original test suites for evaluation.

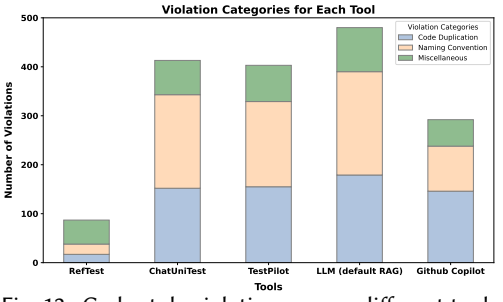

Fig. 12. Code style violations across different tools.

4.3.2 Code Style Violations. **Code Style Violations** assess how well the generated tests adhere to coding standards. Based on Tang et al. [54], these violations are categorized as follows:

- **Code Duplication**: This refers to redundant code elements, such as redefined variables, repeated imports, or duplicate logic.
- **Naming Convention**: This covers inconsistencies in variable, method, or class names, such as the use of non-descriptive or inconsistent names.
- **Miscellaneous**: This includes other style violations such as improper indentation, missing comments, or unorganized imports.

Such violations diminish the maintainability of the generated tests, making them harder to manage and less likely to be accepted by developers. Following the methodology outlined by Tang et al. [54], we used CheckStyle [7] for code style checking. The analysis of code violations is shown in Figure 12, and the detailed breakdown is as follows:

- **Code Duplication**: RefTest introduces **17** instances, while ChatUniTest, TestPilot, LLM with default RAG, and Github Copilot result in significantly higher duplication, with 152, 155, 179 and 146 violations, respectively.
- **Naming Convention**: RefTest has **21** violations, which is substantially lower than ChatUniTest's 191, TestPilot's 174, LLM with default RAG's 211, and Github Copilot's 92 violations.
- **Miscellaneous**: RefTest again performs better, with **49** violations compared to ChatUniTest's 70, TestPilot's 74, LLM with default RAG's 90 and Github Copilot's 54.

These results show that RefTest generates maintainable and readable tests, with significantly fewer code style violations, especially in key features like code duplication and naming conventions.

4.3.3 Mock Density. **Mock Density** evaluates the use of mock objects, as excessive mocking can lead to lower test readability, higher maintenance difficulty, and overly fine-grained test cases [51]. To further analyze the usability of the generated unit tests, we use PMD [45] used by Tang et al. [54] for mock detection. Table 6 compares the additional mocks detected by each tool. The results clearly show that RefTest introduces only 8 additional mocks, significantly fewer than ChatUniTest (61 mocks), TestPilot (56 mocks) and LLM (default RAG) (69 mocks). Github Copilot, while widely used by developers in reality, still adds 42 unnecessary mocks—far more than RefTest. This highlights that RefTest produces the fewest redundant mocks, resulting in cleaner and more maintainable tests.

Table 6. Mock Detection Analysis Across Tools

Tool	Additional Mocks
RefTest	8
ChatUniTest	61
TestPilot	56
LLM (default RAG)	69
Github Copilot	42

4.4 Ablation Study

In this RQ, we aim to evaluate the effectiveness of the reference relationship definition and its decomposition into the Given-When-Then (GWT) phases. Additionally, we assess the contribution of the incremental strategy to the performance of our approach.

4.4.1 Impact of Given-When-Then (GWT) Strategy. We investigate the impact of the GWT strategy in the Reference Analysis phase by comparing test generation results with and without splitting the process into Given, When, and Then phases.

Settings. "GWT" refers to splitting the test generation process into Given, When, and Then phases, as implemented in RefTest. In contrast, "Naive-Way" allows the LLM to find relevant methods without phase-based splitting. Additionally, *No-Given*, *No-When*, and *No-Then* remove the relevant methods identified by their respective phases to evaluate the contribution of each phase. We validate the effectiveness of these strategies on the 1,515 focal methods used in RQ1.

Table 7. Impact of GWT Strategy on Test Generation

Metric	GWT	Naive-Way	No-Given	No-When	No-Then
Compilation and Run Errors	**393 (25.9%)**	597 (39.4%)	787 (51.9%)	657 (43.4%)	609 (40.2%)
Assert Errors	**210 (13.9%)**	316 (20.9%)	353 (23.3%)	420 (27.7%)	499 (32.9%)
Successful Executions	**912 (60.2%)**	602 (39.7%)	375 (24.8%)	438 (28.9%)	407 (26.9%)
Fully Covered	**821 (54.2%)**	487 (32.1%)	298 (19.7%)	358 (23.6%)	331 (21.8%)

Results and Analysis. Table 7 presents the impact of the GWT strategy on test generation across multiple metrics. The results demonstrate that **GWT** consistently outperforms all alternative strategies, achieving the lowest rate of Compilation and Runtime Errors (25.9%) and Assertion Errors (13.9%), while also delivering the highest successful execution rate (60.2%) and Fully Covered rate (54.2%). In comparison, the **Naive-Way**, which does not decompose the test generation process into Given, When, and Then phases, exhibits significantly higher error rates (Compilation and Runtime Errors: 39.4%, Assertion Errors: 20.9%) and lower performance in Successful Executions rate (39.7%) and Fully Covered rate (32.1%). This highlights the effectiveness of phase-based decomposition in guiding the LLM to generate more accurate and comprehensive tests.

The *No-Given*, *No-When*, and *No-Then* variants further emphasize the importance of each phase. Among these, removing the **Given** phase (*No-Given*) results in the highest compilation and runtime error rates (51.9%), suggesting that proper input setup is critical for test generation. Similarly, removing the **When** phase (*No-When*) leads to a noticeable decline in successful executions (28.9%), indicating the importance of accurately capturing method execution. Lastly, removing the **Then** phase (*No-Then*) results in the highest assertion error rate (32.9%), highlighting its essential role in verifying test correctness. Notably, the **Given** phase appears to have the most significant impact, as its removal leads to the largest overall decline in performance metrics. This highlights the foundational importance of input setup in the test generation process. Furthermore, the **Naive-Way** performs better than these three variants because, while it does not explicitly guide the LLM to follow the GWT phases during reference analysis, it does not exclude any phase entirely. This allows it to retain a larger set of methods with reference relationships.

These findings validate the effectiveness of the GWT strategy, with all three phases contributing significantly to improving the correctness, completeness, and reliability of generated unit tests. The decomposition into Given, When, and Then phases enables a structured retrieval process, ensuring that relevant methods are effectively utilized to support test generation.

Table 8. Comparison of Performance Metrics with and without Incremental Strategy (IS)

Project Name	Compilation and Run Error		Assertion Error		Successful Executions		Fully Covered	
	Without IS	With IS	Without IS	With IS	Without IS	With IS	Without IS	With IS
binance-connector-java	22.6%	12.3%	35.8%	34.1%	41.6%	53.6%	38.4%	53.3%
commons-collections	31.7%	17.9%	6.6%	4.8%	61.7%	77.2%	60.3%	73.8%
commons-cli	34.6%	23.1%	11.5%	9.6%	53.9%	67.3%	50.0%	57.7%
morel	42.3%	32.7%	11.5%	9.6%	46.2%	57.7%	42.3%	50.0%
datafaker	70.8%	69.5%	9.7%	8.4%	19.5%	22.1%	18.8%	21.4%
cucumber-expressions	50.0%	37.5%	0.0%	0.0%	50.0%	62.5%	45.8%	54.2%
openapi-diff	61.5%	58.5%	7.7%	4.6%	30.8%	36.9%	29.2%	35.4%
commons-dbutils	4.2%	0.0%	4.2%	0.0%	91.7%	100.0%	91.7%	91.7%
commons-validator	32.0%	20.0%	16.0%	14.7%	52.0%	65.3%	46.7%	53.3%
jsoup	26.2%	11.3%	15.3%	12.5%	58.5%	76.2%	51.6%	60.5%
rtree	60.0%	57.5%	5.0%	2.5%	35.0%	40.0%	27.5%	32.5%
ice4j	50.7%	34.5%	7.0%	5.6%	42.3%	59.9%	34.5%	50.0%

4.4.2 Impact of Incremental Strategy (IS). We further conduct an ablation study to demonstrate the impact of the Incremental Strategy (IS). That means that we generated unit tests for all focal methods $m_A \in M$, regardless of whether the condition $C(m_A, m_B)$ was met. With the Incremental Strategy, we prioritize generating unit tests for methods whose relevant methods already have unit tests. These newly generated unit tests then serve as references for subsequent rounds of test generation. This creates a "propagation" effect, where as many focal methods as possible benefit from existing test cases. In contrast, without the incremental strategy, when a focal method's relevant methods (identified via reference analysis) do not have existing unit tests, the test generation process solely rely on the fallback strategy.

As shown in Table 8, IS significantly reduces the **Compilation and Run Errors** rate across most projects. For example, in project `binance-connector-java`, with IS, the compilation and run error reduces from 22.6% to 12.3%. Similarly, IS also helps in generating more syntactically and semantically correct tests. The **Assertion Error Rate** is also slightly decreased, with `commons-validator` reducing from 16.0% to 14.7% and `rtree` from 5.0% to 2.5%. For the **Successful Execution** rate, IS demonstrates notable improvements. Projects like `commons-collections` and `jsoup` experience increases from 61.7% to 77.2% and from 58.5% to 76.2%, respectively. The **Fully Covered** rate also reflects positive impact, with `binance-connector-java` improving from 38.4% to 53.3% and `commons-cli` from 50.0% to 57.7%. Overall, the results confirm that the Iterative Strategy improves test quality by reducing errors and enhancing both coverage success and overall coverage.

4.5 Exploring Patterns

To understand the reason why RefTest is effective, we further analyze the patterns in *reference relationships*. To do so, we manually analyze the paired methods with reference relationships and summarize the patterns.

4.5.1 Study Design. To minimize bias and ensure reliability, we employ several rigorous strategies in our study design, by referencing methodologies from prior work [46, 63]:

- **Participant Selection and Training**: A team of 5 graduate and doctoral researchers, all majoring in software engineering with relevant expertise, are selected as participants. Comprehensive training sessions are conducted, covering the definition of reference relationships, the review process, and the criteria for validation. Example cases were used to illustrate expected outcomes.
- **Cross-Validation**: Each method relationship was independently classified by two researchers. We measured the inter-rater reliability using Cohen's Kappa, achieving a score of 0.85, which indicates "almost perfect" agreement [26]. A third senior researcher resolved any disagreements through discussion to reach a final consensus.
- **Random Sampling and Multiple Rounds**: We randomly select representative 300 methods. The analysis is conducted in two rounds, with different participants reviewing the same relationships in the second round to verify consistency and identify any discrepancies.

4.5.2 Analysis and Implications. As shown in Table 9, we select the top 6 patterns in reference relationships that demonstrate how test logic can be enhanced across different phases. The first three patterns, Structural Similarity, Behavioral Similarity, and Substitutability, allow for enhancements across all three test phases: *Given*, *When*, and *Then*. These patterns highlight the potential for reusing test logic comprehensively. In contrast, the last three patterns, Exception Handling Similarity, Resource Access Similarity, and Dependency, enhance specific phases of testing. These targeted enhancements focus on key areas like exception handling and resource management, ensuring that testing remains both efficient and contextually appropriate for each method's unique characteristics.

Table 9. Patterns in Reference Relationships

Pattern	Description	Examples	Phases	Count
Structural Similarity	Methods share internal structures or control flows, even if operations differ.	*readFile()* vs. *writeFile()*: Both handle files with similar control structures but perform opposite tasks.	Given, When, Then	320
Behavioral Similarity	Methods achieve similar outcomes but operate on different data or contexts.	*fetchUserName()* vs. *fetchUserEmail()*: Both retrieve user information but return different data types.	Given, When, Then	617
Substitutability	Methods serve the same purpose with different implementations.	*mergeSort()* vs. *quickSort()*: Both sort data but use different algorithms.	Given, When, Then	124
Exception Handling Similarity	Methods handle exceptions in similar ways but under different contexts.	*connectToServer()* vs. *downloadFile()*: Both handle TimeoutException, though one is establishing a server connection and the other is downloading a file.	Then	357
Resource Access Similarity	Methods access the same resource but perform entirely different operations on it.	*lockFile()* vs. *deleteFile()*: Both methods access a file, but one locks it for editing, while the other deletes it.	Given, When	400
Dependency	Methods rely on similar initialization or preconditions for operation.	*initializeIterator()* vs. *asMultipleUseIterable()*: Both need iterator initialization.	Given	351

4.6 Assertion Quality Analysis

While our quantitative metrics (e.g., Assertion Errors rate) demonstrate the correctness of RefTest's generated assertions, we conducted a comprehensive qualitative analysis. We randomly sampled 60 focal methods for each LLM-based tool from the intersection of methods fully covered by all four LLM-based tools. For each selected method, we manually classified all non-trivial assertions generated by RefTest, ChatUniTest, LLM (default RAG), and TestPilot into two levels based on their complexity and practical utility. This classification process was conducted with great care to ensure consistency and accuracy, adhering to the same rigorous standards outlined in Section 4.5. Specifically, each assertion was independently classified into Level 1 (Basic) or Level 2 (Complex) by two researchers. We then measured the inter-rater reliability using Cohen's Kappa coefficient, achieving a value of 0.87, which indicates "almost perfect" agreement. Any initial disagreements were resolved through discussion with a third senior researcher to reach a final consensus.

Table 10. Assertion Quality Levels Across Tools (Number of Assertions)

Tool	Level 1	Level 2
RefTest	164	115
ChatUniTest	97	64
LLM (default RAG)	90	58
TestPilot	112	77

Inspired by TestPilot's definition of "non-trivial assertions" [49] and adapted for Java scenarios, we define "non-trivial assertions" as follows:

- **Level 1 (Basic)**: Verifies the direct input-output relationship or simple, explicit state changes. E.g., assertEquals(5, result) for add(2,3).
- **Level 2 (Complex)**: Verifies specific boundary conditions, precise exception, or the exact content/invariants or specific field values of complex data structures. E.g., assertThat(map.get("key")). isEqualTo("value").

For instance, consider the assertions generated by RefTest for the `OpenApiCompare.fromFiles()` method, shown in Figure 13. This set of assertions is categorized as **Level 2 (Complex)**. It goes beyond basic null checks by verifying multiple critical aspects of the method's core business logic: the identification of both new and missing endpoints, which are fundamental types of API changes.

This demonstrates an understanding of the complex `ChangedOpenApi` object's internal structure and its key attributes. Crucially, while a simple call to `fromFiles()` already achieve full code coverage, RefTest generates these detailed assertions to ensure comprehensive functional verification beyond mere coverage metrics. This level of precision is vital for practical use, highlighting RefTest's strength in generating assertions that capture deep functional verification.

```
// Method: OpenApiCompare.fromFiles(oldFile, newFile,
↪ filters)
// Context: Comparing two OpenAPI specification files.

ChangedOpenApi changedOpenApi = OpenApiCompare.fromFiles(
    oldFile, newFile, Collections.emptyList());

// Assertions generated by RefTest for
↪ testFromFilesWithValidFiles
assertThat(changedOpenApi).isNotNull();
assertThat(changedOpenApi.getNewEndpoints()).isNotEmpty();
assertThat(changedOpenApi.getMissingEndpoints()).isNotEmpty();
```

Fig. 13. Example of a Level 2 (Complex) Assertion by RefTest

As for comparison with other tools, the results shown in Table 10 highlighting RefTest's strength. It produces a substantially larger number of both Basic (Level 1) and Complex (Level 2) assertions compared to all baselines. This qualitative superiority stems from RefTest's unique ability to effectively leverage reference relationships, particularly those identified during the "Then" phase, which often involve sophisticated validation logic. By focusing on meaningful functional verification rather than mere syntactic correctness or general coverage, RefTest demonstrates a significant advancement in generating truly useful unit tests.

5 DISCUSSION

This section first evaluates RefTest by comparing its generated tests with existing reference tests, then examines its performance in cold start scenarios and the utility of pre-existing tests, followed by discussions on extensibility and validity threats.

5.1 Case Study

To demonstrate the effectiveness of RefTest, we analyze the similarity between generated tests and tests for methods sharing a "reference relationship". We use a representative example from the jsoup project to highlight how RefTest leverages reference relationships to generate correct and contextually relevant tests.

```
public class Element extends Node {
    ...
    public Node traverse(NodeVisitor nodeVisitor) {
        Validate.notNull(nodeVisitor);
        NodeTraversor.traverse(nodeVisitor, this);
        return this;
    }

    public Node forEachNode(Consumer<? super Node>
    ↪ action) {
        Validate.notNull(action);
        nodeStream().forEach(action);
        return this;
    }
}
```

Fig. 14. Simplified Example from the jsoup Project

As shown in Figure 14, the `Element` class in the jsoup represents an HTML element in the DOM tree. It provides methods `traverse` and `forEachNode` to traverse and manipulate the DOM. The `traverse` method uses a `NodeVisitor` to handle both pre-visit and post-visit actions, making it suitable for complex tasks. In contrast, `forEachNode` employs a simpler `Consumer` to perform direct actions on each visited node, offering a lightweight alternative. These two methods, `traverse` and `forEachNode`, share the same purpose of traversing the node tree but differ in their implementation.

When generating a test for `forEachNode`, RefTest effectively identified its relationship with `traverse` and retrieved the existing test [3] for `traverse` (Figure 15a). RefTest's generated test for `forEachNode`(Figure 15b) referenced elements from `testTraverseLambda` in both the *Given* phase (object construction) and the *Then* phase (assertion logic). Notably, in the *When* phase (method invocation), RefTest introduced an `AtomicInteger` with its `incrementAndGet` method to handle counting. This addition was specifically designed to address potential **multi-threaded environments**, ensuring the robustness of the generated test. In contrast, although tests generated by other tools may execute successfully, they fail to account for multi-threading scenarios. This limitation stems from their reliance solely on the source code of `forEachNode`, without utilizing the contextual information provided by related methods like `traverse`. This underscores RefTest's strength in performing effective reference analysis, allowing it to generate tests that are not only functional but also contextually aware and robust.

```java
@Test
void testTraverseLambda() {
    Document doc =
    ↪ Jsoup.parse("<div><p>One<p>Two<p>Three");
    Element div = doc.selectFirst("div");
    final AtomicInteger counter = new
    ↪ AtomicInteger(0);

    Element div2 = div.traverse((node, depth) ->
    ↪ counter.incrementAndGet());

    assertEquals(7, counter.get());
    assertEquals(div2, div);
}
```

(a) Existing Test for `traverse`

```java
@Test
public void testForEachNode() {
    Document doc =
    ↪ Jsoup.parse("<div><p>1<p>2<p>3</div>");
    element = doc.body().child(0);
    AtomicInteger seenCount = new
    ↪ AtomicInteger();
    Consumer<Node> action = node -> {
        if (node instanceof Element && ((Element)
        ↪ node).tagName().equals("p")) {
            seenCount.incrementAndGet();
        }
    };
    element.forEachNode(action);
    assertEquals(3, seenCount.get());
}
```

(b) Generated Test for `forEachNode`

Fig. 15. Comparison of Existing and Generated Tests. The left test validates `traverse`, while the right test demonstrates how RefTest adapts the structure to generate a consistent test for `forEachNode`.

5.2 Cold Start Scenarios and Value of Existing Tests

Our proposed RefTest approach is primarily designed to leverage existing test cases through defined reference relationships to generate high-quality, maintainable unit tests. This paradigm offers significant advantages by drawing upon developers' prior investment and domain knowledge embedded in pre-existing tests, which tend to be more practical, understandable, and aligned with real-world requirements. However, in scenarios where no such pre-existing tests are available (i.e., cold start scenarios), RefTest can rely on fallback strategies or integrate with complementary tools to generate an initial set of test cases. These initial test cases serve as a foundation for subsequent iterations. It is important to emphasize that cold start scenarios are not the optimal use case for RefTest, as they lack the rich contextual information provided by high-quality, pre-existing tests. The fallback mechanisms are primarily intended to ensure functionality in the absence of such tests, rather than to fully substitute the advantages offered by well-crafted, domain-specific test cases.

To evaluate RefTest's capability without existing tests, we conducted an ablation study on three projects from our dataset by deliberately removing all pre-existing tests. As shown in Table 11, cold start results vary with project complexity. For instance, jsoup achieved 85 fully covered tests

[3] https://github.com/jhy/jsoup/blob/master/src/test/java/org/jsoup/nodes/ElementTest.java

(compared to 150 normally), and commons-collections generated 180 (compared to 214 normally), demonstrating reasonable performance for projects of moderate complexity. ice4j, a highly complex project, saw a more significant drop (12 vs 71 normally), highlighting that the value of existing test context increases with code complexity. Notably, RefTest's cold start performance is comparable to or exceeds the best results achieved by other baseline tools. For instance, its performance on jsoup (85 fully covered tests) and commons-collections (180 tests) surpassed the best results from TestPilot (70 and 157, respectively). In the case of the complex ice4j project, RefTest's result of 12 tests was also superior to the best baseline performance of 10 from ChatUniTest. These findings confirm RefTest's robustness in cold start while underscoring that its optimal power is unleashed when existing test contexts are available.

Table 11. Performance of RefTest in Cold Start vs. Normal Scenarios

Project	Fully Covered (Normal)	Fully Covered (Cold Start)
jsoup	150	85
ice4j	71	12
commons-collections	214	180

5.3 Extensibility

Function-Level Enhancement. While our paper demonstrates class-level and repository-level enhancement relationships, for a focal method, its own existing test cases can also provide valuable insights for generating new tests. Although this does not depend on the defined *reference relationships*, it leverages task-specific context effectively.

Extend to Other Code-Related Tasks. The approach extends beyond unit test generation and can be applied to other code-related tasks. For a given code snippet C and a task T that produces an output G, If the output G_i can guide the generation of G, the C and C_i have reference relationships. These relationships enable the identification of relevant contextual references, ensuring more accurate and relevant outputs. For example, in the context of API documentation generation , the reference relationships may differ from the GWT paradigm used in unit test generation. Instead, the task may involve stages like *Define*, *Explain*, and *Example*, which help identify and retrieve references to related code and documentation. By aligning the task stages with the reference's characteristics, the approach ensures that the generated API documentation is both precise and contextually enriched. This extensibility demonstrates reference relationship's generalizability as a framework for improving various code generation tasks through the systematic use of reference relationships.

5.4 Threats to Validity

The threats to the validity of RefTest primarily involve external validity (generalization to other LLMs and project diversity) and internal validity (potential data contamination and specific experimental design choices). Regarding generalizability to other LLMs, our evaluation utilized DeepSeek-V2.5, and performance may vary with different underlying Large Language Models. The diversity of projects evaluated could introduce bias, as these projects vary in complexity and existing test coverage. For data contamination, while challenging to eliminate completely, we minimized this risk by selecting projects updated after DeepSeek-V2.5's release date. All LLM-based tools compared utilize the same underlying LLM, ensuring fair relative performance comparisons. In the TestPilot comparison, its core ideas were reimplemented in Java for our evaluation. However, subtle differences in implementation or language-specific nuances could affect comparability. Future work will involve expanding the dataset, including more diverse projects, and testing with other LLMs to better assess the generalizability and robustness of RefTest.

6 RELATED WORK

In this section, we review relevant research in two key areas: automated unit test generation and the application of Retrieval-Augmented Generation (RAG) in the code domain.

6.1 Unit Test Generation

Unit testing focuses on testing individual hardware or software units, or groups of related units [40]. Several test case generation approaches have been proposed, including random-based methods [11, 60] and constraint-driven techniques [22]. Approaches such as DART [17] and KLEE [35], which leverage symbolic execution, often face challenges like path explosion [4]. Search-based software testing (SBST) techniques [9, 14, 27, 30, 52, 64, 73], such as EvoSuite [2] and evolutionary testing [55], mitigate some of these issues but struggle with broad search spaces and high computational costs [36]. In contrast, deep learning-based methods [5, 14, 57, 67, 72], like AthenaTest [56], leverage neural models to generate diverse test inputs and better capture functional program intent.

Recently, large language models (LLM)-based techniques have gained popularity in testing [1, 12, 20, 31, 42, 58]. TestPilot[49] generates test for function using its signature and implementation, along with usage examples extracted from documentation. ChatUniTest[10] uses ChatGPT to generate Java-compliant test cases after static analysis. Yuan et al. [69] proposed ChatTester, which refines initial test cases with ChatGPT, sometimes outperforming traditional SBST methods. Hybrid approaches like TELPA [65], uses program analysis with LLM by integrating refined counter examples into prompts, guiding the LLM to generate diverse tests for hard-to-cover branches, and HITS [59] decomposes focal methods into slices and ask the LLM to generate tests slice by slice, improving coverage for complex methods. Fuzz4All leverages LLMs for diverse input generation, while ChatAFL applies them to protocol fuzzing [37, 62]. CovRL-Fuzz integrates LLMs with RL for JavaScript interpreter testing [13]. While TestPilot primarily enhances prompts with function-specific usage examples extracted from documentation, the difference is that RefTest leverages inter-method relationships for test reference. These tools have demonstrated promising results in specific scenarios, but they often fail to account for existing test cases. This leads to generated tests lacking practical usability and alignment with existing testing infrastructure. In contrast, our tool not only comprehends the focal method in-depth, but also leverages existing test cases to increase the correctness and practical applicability of the generated tests.

6.2 Retrieval-Augmented Generation (RAG) for Code Tasks

Retrieval-Augmented Generation (RAG) was first introduced in the context of NLP tasks to enhance the knowledge retrieval and generation capabilities of language models, especially for tasks requiring extensive domain knowledge [28]. RAG systems, particularly Retrieval-Augmented Code Generation (RACG), enhance large language models (LLMs) by retrieving relevant code snippets or structures from repositories [24, 33, 44, 53]. Existing approaches include similarity-based methods [21, 48], such as using the BM25 algorithm to perform similarity retrieval [25], as well as vector similarity-based retrieval [43]. Additionally, AST-based retrieval approaches, such as AutoCodeRover [71], leverage AST structures to retrieve relevant code contexts. Some methods design specific tools to perform more targeted retrieval, as in the case of MASAI [3], while CODEXGRAPH [32] uses code graph databases to allow for flexible and powerful retrieval to get more code structure information. Further approaches involve retrieving based on “repo-specific semantic graphs” [29]. Our tool extends traditional retrieval methods by considering task-specific contexts of code and defining property relationships to guide the retrieval process, offering a more tailored approach to enhancing unit test generation.

7 CONCLUSION

We presented *Reference-Based Retrieval Augmentation for Unit Test Generation*, a novel approach that enhances the correctness, completeness, and maintainability of unit tests by leveraging reference relationships between methods. By extending LLM-based Retrieval-Augmented Generation (RAG) with code-specific relationships such as behavioral similarities and structural dependencies, our method improved test generation through the *Given*, *When*, and *Then* phases. RefTest further employed an iterative strategy, using newly generated tests to guide future ones. Our evaluation across 12 open-source projects demonstrated that RefTest significantly outperformed existing tools, offering a promising direction for more reliable and context-aware test generation.

Moreover, our work introduced a novel code-context-aware retrieval mechanism for LLMs, offering valuable insights and potential applications for other code-related tasks.

Acknowledgments

This work was supported by the National Key R&D Program of China No 2024YFB4506200, Aviation Science Foundation of China under Grant No 20240058051002, and National Natural Science Foundation of China under Grant No 62202026.

References

[1] Saranya Alagarsamy, Chakkrit Tantithamthavorn, Chetan Arora, and Aldeida Aleti. 2024. Enhancing Large Language Models for Text-to-Testcase Generation. *arXiv preprint arXiv:2402.11910* (2024).

[2] J. H. Andrews, T. Menzies, and F. C. H. Li. 2011. Genetic algorithms for randomized unit testing. *IEEE Transactions on Software Engineering* 37, 1 (2011), 80–94.

[3] Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhav Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. 2024. MASAI: Modular Architecture for Software-engineering AI Agents. *arXiv preprint arXiv:2406.11638* (2024).

[4] Riccardo Baldoni, Enrico Coppa, Domenico C D'elia, et al. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[5] Arianna Blasi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. 2022. Call me maybe: Using nlp to automatically generate unit test cases respecting temporal constraints. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–11.

[6] ChatBotArena. 2024. "Chatbot Arena LLM Leaderboard: Community-driven Evaluation for Best LLM and AI chatbots". https://lmarena.ai/ Accessed: 2024.

[7] Checkstyle. 1999. "Code conventions for the java programming language". https://checkstyle.sourceforge.io/ Accessed: 2024.

[8] Jiuhai Chen, Lichang Chen, Heng Huang, and Tianyi Zhou. 2023. When do you need chain-of-thought prompting for chatgpt? *arXiv preprint arXiv:2304.03262* (2023).

[9] Junjie Chen, Chenyao Suo, Jiajun Jiang, Peiqi Chen, and Xingjian Li. 2023. Compiler test-program generation via memoized configuration search. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2035–2047.

[10] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 572–576.

[11] Matthew C Davis, Sangheon Choi, Sam Estep, Brad A Myers, and Joshua Sunshine. 2023. NaNofuzz: A Usable Tool for Automatic Test Generation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1114–1126.

[12] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435.

[13] Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. 2024. Covrl: Fuzzing javascript engines with coverage-guided reinforcement learning for llm-based mutation. *arXiv preprint arXiv:2402.12222* (2024).

[14] Patric Feldmeier and Gordon Fraser. 2022. Neuroevolution-based generation of tests and oracles for games. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[15] Gordon Fraser and Andrea Arcuri. 2011. 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*. 416–419.

[16] Github. 2024. "The world's most widely adopted AI developer tool.". https://github.com/features/copilot Accessed: 2024.

[17] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. ACM, 213–223.

[18] Giovanni Grano, Simone Scalabrino, Harald C Gall, and Rocco Oliveto. 2018. An empirical investigation on the readability of manual and generated test cases. In *Proceedings of the 26th Conference on Program Comprehension*. 348–351.

[19] Siqi Gu, Chunrong Fang, Quanjun Zhang, Fangyuan Tian, and Zhenyu Chen. 2024. TestART: Improving LLM-based Unit Test via Co-evolution of Automated Generation and Repair Iteration. *arXiv preprint arXiv:2408.03095* (2024).

[20] Vitor Guilherme and Auri Vincenzi. 2023. An initial investigation of ChatGPT unit test generation capability. In *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing*. 15–24.

[21] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).

[22] Sungjae Hwang, Sungho Lee, Jihoon Kim, and Sukyoung Ryu. 2021. Justgen: Effective test generation for unspecified JNI behaviors on jvms. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1708–1718.

[23] jedi. 2024. "Awesome autocompletion, static analysis and refactoring library for python". https://github.com/davidhalter/jedi Accessed: 2024.

[24] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. *arXiv preprint arXiv:2406.00515* (2024).

[25] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770* (2023).

[26] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174.

[27] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 919–931.

[28] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[29] Ming Liang, Xiaoheng Xie, Gehao Zhang, Xunjin Zheng, Peng Di, wei jiang, Hongwei Chen, Chengpeng Wang, and Gang Fan. 2024. RepoFuse: Repository-Level Code Completion with Fused Dual Context. arXiv:2402.14323 [cs.SE] https://arxiv.org/abs/2402.14323

[30] Jun-Wei Lin, Navid Salehnamadi, and Sam Malek. 2023. Route: Roads not taken in ui testing. *ACM Transactions on Software Engineering and Methodology* 32, 3 (2023), 1–25.

[31] Kaibo Liu, Yiyang Liu, Zhenpeng Chen, Jie M Zhang, Yudong Han, Yun Ma, Ge Li, and Gang Huang. 2024. LLM-Powered Test Case Generation for Detecting Tricky Bugs. *arXiv preprint arXiv:2404.10304* (2024).

[32] Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Wenmeng Zhou, Fei Wang, and Michael Shieh. 2024. CodexGraph: Bridging Large Language Models and Code Repositories via Code Graph Databases. *arXiv preprint arXiv:2408.03910* (2024).

[33] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. *arXiv preprint arXiv:2203.07722* (2022).

[34] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 168–172.

[35] Lin Ma, Cyril Artho, Chao Zhang, et al. 2015. Grt: Program-analysis-guided random testing. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 212–223.

[36] Phil McMinn. 2011. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 153–163.

[37] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, Vol. 2024.

[38] microsoft. 2024. "Language Server Protocol.". https://microsoft.github.io/language-server-protocol Accessed: 2024.

[39] Pierre Néron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A theory of name resolution. In *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 24*. Springer, 205–231.

[40] Michael Olan. 2003. Unit testing: test early, test often. *Journal of Computing Sciences in Colleges* 19, 2 (2003), 319–328.
[41] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.
[42] Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. 2024. Multi-language unit test generation using llms. *arXiv preprint arXiv:2409.03093* (2024).
[43] Zhiyuan Pan, Xing Hu, Xin Xia, and Xiaohu Yang. 2024. Enhancing Repository-Level Code Generation with Integrated Contextual Information. *arXiv preprint arXiv:2406.03283* (2024).
[44] Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. *arXiv preprint arXiv:2108.11601* (2021).
[45] Pmd. 2023. "An extensible cross-language static code analyzer.". https://pmd.github.io/ Accessed: 2023.
[46] Dezhi Ran, Hao Wang, Zihe Song, Mengzhou Wu, Yuan Cao, Ying Zhang, Wei Yang, and Tao Xie. 2024. Guardian: A Runtime Framework for LLM-Based UI Exploration. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 958–970.
[47] redisson. 2024. "Easy Redis Java client and Real-Time Data Platform". https://github.com/redisson/redisson Accessed: 2024.
[48] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.
[49] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* 50, 1 (2023), 85–105.
[50] Sergio Servantez, Joe Barrow, Kristian Hammond, and Rajiv Jain. 2024. Chain of Logic: Rule-Based Reasoning with Large Language Models. *arXiv preprint arXiv:2402.10400* (2024).
[51] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. 2017. To mock or not to mock? an empirical study on mocking practices. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 402–412.
[52] Baicai Sun, Dunwei Gong, Feng Pan, Xiangjuan Yao, and Tian Tian. 2023. Evolutionary generation of test suites for multi-path coverage of MPI programs with non-determinism. *IEEE Transactions on Software Engineering* 49, 6 (2023), 3504–3523.
[53] Hanzhuo Tan, Qi Luo, Ling Jiang, Zizheng Zhan, Jing Li, Haotian Zhang, and Yuqun Zhang. 2024. Prompt-based Code Completion via Multi-Retrieval Augmented Generation. *arXiv preprint arXiv:2405.07530* (2024).
[54] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. 2024. Chatgpt vs sbst: A comparative assessment of unit test suite generation. *IEEE Transactions on Software Engineering* (2024).
[55] Paolo Tonella. 2004. Evolutionary testing of classes. *ACM SIGSOFT Software Engineering Notes* 29, 4 (2004), 119–128.
[56] M. Tufano, D. Drain, A. Svyatkovskiy, and et al. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020).
[57] Chunhui Wang, Fabrizio Pastore, Arda Goknil, and Lionel C Briand. 2020. Automatic generation of acceptance test cases from use case specifications: an nlp-based approach. *IEEE Transactions on Software Engineering* 48, 2 (2020), 585–616.
[58] Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2024. TESTEVAL: Benchmarking Large Language Models for Test Case Generation. *arXiv preprint arXiv:2406.04531* (2024).
[59] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1258–1268.
[60] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering*. 995–1007.
[61] wikipedia. 2024. "Given-When-Then.". https://en.wikipedia.org/wiki/Given-When-Then Accessed: 2024.
[62] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[63] Congying Xu, Songqiang Chen, Jiarong Wu, Shing-Chi Cheung, Valerio Terragni, Hengcheng Zhu, and Jialun Cao. 2024. MR-Adopt: Automatic Deduction of Input Transformation Function for Metamorphic Testing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 557–569.
[64] Rahul Krishna Yandrapally and Ali Mesbah. 2022. Fragment-based test generation for web apps. *IEEE Transactions on Software Engineering* 49, 3 (2022), 1086–1101.
[65] Chen Yang, Junjie Chen, Bin Lin, Jianyi Zhou, and Ziqi Wang. 2024. Enhancing LLM-based Test Generation for Hard-to-Cover Branches via Program Analysis. *arXiv preprint arXiv:2404.04966* (2024).
[66] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, et al. 2024. On the evaluation of large language models in unit test generation. In *Proceedings of the*

39th IEEE/ACM International Conference on Automated Software Engineering. 1607–1619.

[67] Guixin Ye, Tianmin Hu, Zhanyong Tang, Zhenye Fan, Shin Hwei Tan, Bo Zhang, Wenxiang Qian, and Zheng Wang. 2023. A Generative and Mutational Approach for Synthesizing Bug-Exposing Test Cases to Guide Compiler Fuzzing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1127–1139.

[68] Xiao Yu, Lei Liu, Xing Hu, Jacky Keung, Xin Xia, and David Lo. 2024. Practitioners' Expectations on Automated Test Generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1618–1630.

[69] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1703–1726.

[70] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570* (2023).

[71] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1592–1604.

[72] Yixue Zhao, Saghar Talebipour, Kesina Baral, Hyojae Park, Leon Yee, Safwat Ali Khan, Yuriy Brun, Nenad Medvidović, and Kevin Moran. 2022. Avgust: automating usage-based test generation from videos of app executions. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 421–433.

[73] Zhichao Zhou, Yuming Zhou, Chunrong Fang, Zhenyu Chen, and Yutian Tang. 2022. Selectively combining multiple coverage goals in search-based unit test generation. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[74] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence.