

LLM-based Unit Test Generation via Property Retrieval

ZHE ZHANG, Beihang University, China

XINGYU LIU, Beihang University, China

YUANZHANG LIN, Beihang University, China

XIANG GAO*, Beihang University, China

HAILONG SUN, Beihang University, China

YUAN YUAN, 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 on vast text and code data, excel in various code-related tasks, including unit test generation. However, existing LLM-based tools often narrow their focus to general code context, overlooking broader task-specific context, such as leveraging existing tests in unit test generation tasks. 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 *Property-Based Retrieval Augmentation*, a novel mechanism that extends LLM-based Retrieval-Augmented Generation (RAG) beyond basic vector, text similarity, and graph-based methods. Our approach considers task-specific context in the code domain and introduces a tailored property retrieval mechanism that leverages the unique characteristics of code. Specifically, in the unit test generation task, we account for the unique structure of unit tests by dividing the test generation process into *Given*, *When*, and *Then* phases. When generating tests for a focal method, we not only retrieve general context for the code under test but also consider task-specific context such as pre-existing tests of other methods, which can provide valuable insights for any of the *Given*, *When*, and *Then* phases. This forms *property relationships* between focal method and other methods, thereby expanding the scope of retrieval beyond traditional RAG. We implement this approach in a tool called *APT*, which sequentially performs preprocessing, property retrieval, and unit test generation, using an iterative strategy where newly generated tests guide the creation of subsequent ones. We evaluated APT on 12 open-source projects with 1515 methods, and the results demonstrate that APT consistently outperforms existing tools in terms of correctness, completeness, and maintainability of the generated tests. Moreover, we introduce a novel code-context-aware retrieval mechanism for LLMs beyond general context, offering valuable insights and potential applications for other code-related tasks.

1 Introduction

Automated unit test generation has been extensively studied, resulting in a variety of approaches. Prominent methods include random-based strategies [9, 34, 48] and constraint-driven techniques, such as DART [14] and KLEE [29]. Search-Based Software Testing (SBST) techniques [8, 11, 23, 25, 28, 42, 52, 60], exemplified by tools like EvoSuite [12], aim to improve the effectiveness of test generation but still struggle with large search spaces and high computational costs [30]. Recently, deep learning-based approaches, such as AthenaTest [45], have emerged, utilizing neural models to generate more diverse test inputs and better capture the functional intent of code [4, 11, 46, 54, 59].

Additionally, Large Language Models (LLM)-based techniques are becoming increasingly popular in automated testing [10, 17, 23]. Tools like ChatUniTest [50] and ChatTester [56] 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 [47, 53] combine the strengths of LLMs and traditional techniques to improve test quality. For instance,

*Corresponding author.

TELPA [53], 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 [47] decomposes focal methods into slices and asks the LLM to generate tests slice by slice, improving coverage for complex methods.

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 integrating relevant code snippets or structures from code repositories. However, these approaches often concentrate solely on contextualizing the original code using various retrieval techniques, such as similarity-based retrieval (e.g., BM25 [22]), AST-based retrieval [58], graph-based retrieval [27], iterative file-based retrieval [57], and vector similarity-based retrieval [35]. Despite their effectiveness, these methods do not fully account for the specific context of the code 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: Practical Problem in Unit Test Generation. Despite advancements in automated test generation, many tools remain focused on maximizing coverage metrics. However, a recent study by Yu et al. [55] 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) [15] 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. For instance, ChatUniTest [50] incorporates an adaptive focal context mechanism that captures useful context within prompts. 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?

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 *property relationships*. Specifically, unit tests are usually structured into three key phases: *Given*, *When*, and *Then* [49]. 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 property relationship. Suppose we are generating a unit test for decode method (Base64 decoding), and there already exists a test case for encode method

(Base64 encoding). Although encode and decode are inverse operations, the test case of encode can still provide useful references at each phase of the test generation of decode:

- **Given Phase:** The encode test case provides object initialization and input setup, which can give reference for decode, as both methods process similar data structures (e.g., handling strings and converting them to/from Base64 format).
- **When Phase:** Although encode and decode perform inverse actions, the method invocation in encode, including how input data is handled, offers valuable insights for calling decode, particularly in the way the data is prepared before processing.
- **Then Phase:** The assertions and exception handling in encode guide the verification process for decode, such as output validation’s format, boundary checking (e.g., verifying the behavior when the input is an empty), and exception handling (e.g., reusing the same logic for catching and handling invalid Base64 input).

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 *property relationships*, our approach enables the retrieval of task-specific contexts, enhancing the effectiveness of test case generation.

Based on our approach, we propose APT, which 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, APT analyzes existing test cases, extracts *test bundles*—the core reference units for generating new tests—and maps them to focal methods. Additionally, APT introduces a *Static Context Retrieval* mechanism to streamline the retrieval and generation processes. Then, APT executes two main steps:

- **Retrieval.** After preprocessing, APT performs property retrieval. For a given focal method, it uses *property analysis* to find methods with property relationships within the class. It then extends to class inheritance and interface implementation to perform *property deduction*, expanding the search to the entire repository. The relevant methods and their *test bundles* are retrieved.
- **Generation.** APT 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, APT applies an *Iterative Strategy*, where newly generated test cases guide the generation of subsequent tests, maximizing the number of correct test cases.

In Summary, we propose a novel approach, *Property-Based Retrieval Augmentation for Unit Test Generation*. This method defines *property relationships* between methods, APT 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 property relationships and contextual information, our approach significantly improves the **correctness**, **completeness**, and **maintainability** of the generated unit tests. This paper offers several key contributions:

- We are the first to propose a *property-based retrieval* approach for unit test generation. By decomposing the test generation process into the GWT phases and defining properties between methods and property retrieval, we leverage existing test cases to generate unit tests for focal methods. This extends the traditional Retrieval-Augmented Generation (RAG) paradigm to incorporate code-specific property features.
- We propose APT, which integrates preprocessing, retrieval, and generation to produce high-quality unit tests. Additionally, our iterative strategy maximizes the number of focal methods that can be covered by leveraging existing test cases with property relationships.

- We conduct comprehensive evaluations of APT on 12 open-source projects with 1515 methods. The results show that APT 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 *property relationship* that exists at the Class-Level. We then extend this relationship to the Repo-Level.

2.1 Class-Level Enhancement

To illustrate motivations behind our approach, we take as an example the `reset` method from the `RedissonLongAdder` class in the `Redisson` repository [38], a widely used Java client for Redis with over 23k stars on GitHub. We start by generating unit tests for the `reset` method.

```

1 public RFuture<Void> reset() {
2     String id = getServiceManager().generateId();
3     RSemaphore semaphore = getSemaphore(id);
4     RFuture<Long> future = topic.publishAsync(CLEAR_MSG + ":" + id);
5     CompletionStage<Void> f = future.thenCompose(r -> semaphore.acquireAsync(r.intValue()))
6                                     .thenCompose(r -> semaphore.deleteAsync().thenApply(res -> null));
7     return new CompletableFutureWrapper<>(f);
8 }

```

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 5). Upon successful message delivery, it acquires and deletes the semaphore (lines 6-7), 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 [50, 56], 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 code under test in

order to better understand what the method does, thus helping to generate test cases. For example, When generating tests using GitHub Copilot [13], 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 test generated by GitHub Copilot first

```

1 @BeforeEach
2 public void setUp() {
3     redissonMock = mock(RedissonClient.class);
4     redissonLongAdder = new RedissonLongAdder(mock(
5         ↪ CommandAsyncExecutor.class), "test",
6         ↪ redissonMock);
7 }
8 @Test
9 public void testReset() {
10     RSemaphore semaphoreMock = mock(RSemaphore.class);
11     .....
12     RFuture<Void> result = redissonLongAdder.reset();
13
14     verify(semaphoreMock).acquireAsync(anyInt());
15     verify(semaphoreMock).deleteAsync();
16     Assert.assertNotNull(result);
17 }

```

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

sets up the environment (lines 1-5) by mocking `RedissonClient` and creating `RedissonLongAdder` with mocked dependencies. Then, the test directly calls `reset()` (lines 11) and verifies that the result of `reset()` is not null (lines 15). Moreover, the test also 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 test case, it failed 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 test is focused primarily on confirming that external methods were invoked, without verifying whether the internal state reset was correctly applied.

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 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 APT focuses on identifying methods that have a *property 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 *property analysis*, APT identifies a *property relationship* between `reset` method and `sum` method from class `RedissonLongAdder`. Although their functionalities are semantically different, APT observes significant similarities in the GWT stages of their unit tests. The test cases 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.

```
@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

```
@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.

When APT 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, APT 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.

2.2 Repo-Level Enhancement

Further, in object-oriented programming, when we broaden our focus from an individual class to the entire repository, we can identify more extensive attribute enhancement 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 benefit from mutual enhancement. For instance, the shared abstract behavior of two classes implementing a common interface can guide the generation of consistent and comprehensive unit tests.

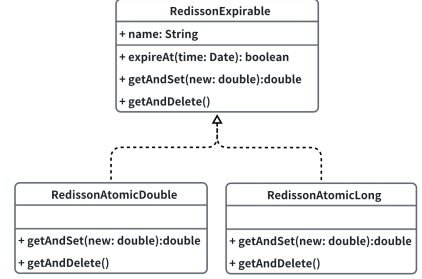


Fig. 4. RedissonAtomicDouble and RedissonAtomicLong both implementing the abstract class RedissonExpirable

To demonstrate the Repo-Level test enhancement 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.

Similarly, parent-child class relationships also exhibit similar test enhancement effects, where shared methods between parent and child classes benefit from the same mutual enhancement of unit tests. And when two classes implement a common interface, methods shared by these classes can follow the same pattern of test enhancement as sibling classes. In this case, 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 property-based enhancement relationships extend across Class-Level and Repo-Level contexts. By taking these relationships into account, we can generate more accurate and effective unit tests for the focal method.

3 PROBLEM FORMULATION

In this section, we formally define the problem of generating test cases for focal methods in a code repository by leveraging existing test cases through property relationships.

3.1 Formal Problem Definition

Let *Repo* be a code repository containing a set of classes $C = \{c_1, c_2, \dots, c_k\}$. Methods in *Repo* are denoted as $M = \bigcup_{c \in C} M(c)$, where $M(c) = \{m_1, m_2, \dots, m_p\}$ 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 donated as

```

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.exists()).isFalse();
        RAtomicDouble ad2 = redisson.
            ↪ getAtomicDouble("test2");
        assertThat(ad2.getAndDelete()).isZero();
    }
}

```

(a) The Unit Test of RedissonAtomicDouble

```

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.exists()).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 provide guidance for generating tests in `RedissonAtomicLongTest` due to the structural and behavioral similarities of the methods.

$TC = \{tc_1, tc_2, \dots\}$, each associated with some methods in M . Given a focal method $m_t \in M$, our goal is to generate unit test(s) UT_t for m_t by:

- (1) **Property Definition:** Defining a *property relation* $P \subseteq M \times M$, where $(m_t, m_r) \in P$ signifies that the method $m_r \in M$ is related to m_t for the purpose of test case generation.
- (2) **Property Retrieval:** Define the operation $PropertyRetrieval(m_t)$, which takes a focal method m_t as input and retrieves the set of related methods M_r such that $(m_t, m_r) \in P$. The result of this operation is denoted as $R(m_t)$, representing the *property relation method set*. This set contains all methods that have a property relationship with m_t , along with the specific description of the property relationships (e.g., *Given*, *When*, *Then* phases).
- (3) **Generation:** Utilize the property relation method set $R(m_t)$ and the existing test cases TC_r associated with methods in $R(m_t)$ to generate effective unit tests TC_t for the focal method m_t .

3.2 Challenges

Several challenges must be addressed to solve this problem effectively:

3.2.1 Identifying Complex and Broader Property Relationships.

- **Complexity of Relationships:** Relationships across the *Given*, *When*, and *Then* phases vary and are often complex, involving different types of connections such as structural, behavioral, and functional.
- **Broad Scope of Relationships:** These relationships can exist both within a class (intra-class) and across multiple classes (inter-class), making it crucial to consider relationships throughout the entire code repository.
- **Proximity and Relevance:** Not all relationships are equally useful. Some are more immediate, while others are more distant or indirect. Identifying the most relevant relationships that will contribute to effective test case generation is essential.

3.2.2 Generating Effective Unit Tests.

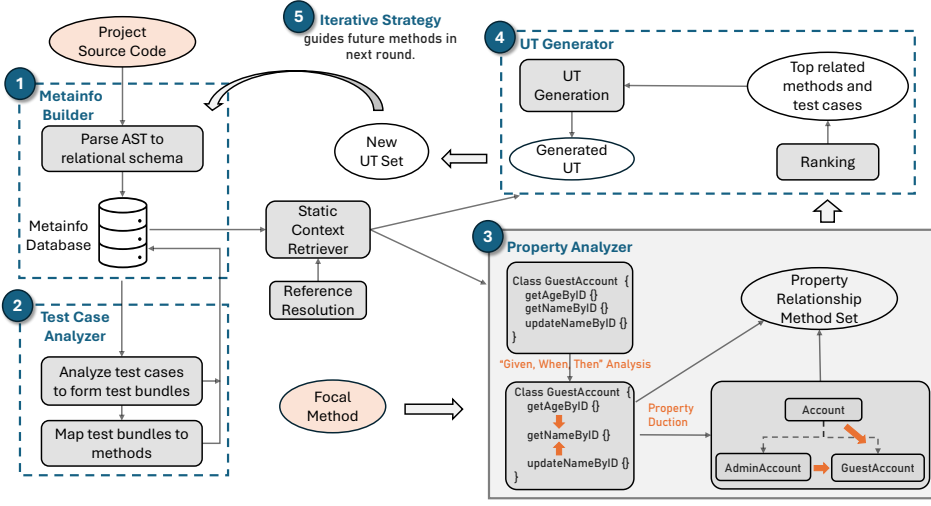


Fig. 6. The overall workflow of APT.

- **Capturing Contextual Dependencies from Existing Test Cases:** Test cases are not standalone but rely on specific setups (e.g., class members, fixtures, or external modules). Capturing these dependencies as part of the context is essential for generating correct and effective tests.
- **Integrating Property-Related Methods and Test Cases:** Once the relevant methods in $R(m_t)$ are retrieved, the challenge lies in determining the priority of these methods and their associated test cases. This involves how to rank the methods based on their relevance and how to better integrate the existing test cases with the retrieved ones to provide the most effective context for the LLM during test case generation.
- **Maximizing Test Case Generation:** The goal is to generate unit tests for as many methods as possible by leveraging property relationships throughout the codebase. The challenge is to ensure that the $R(m_t)$ relationships are effectively utilized not only for m_t but also to benefit as many methods as possible, allowing broader application of *Property-Based Retrieval Augmentation for Unit Test Generation* across the entire project.

4 METHODOLOGY

We propose APT to address the problems and challenges previously outlined. The overall workflow of APT 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 abstracts the test cases into *Test Bundles*, which are then mapped back to their corresponding methods. Third, the *Property 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 to identify intra-class properties and uses property deduction to retrieve related methods across the entire repository. These results form the *Property Relationship Method Set*. Last, the *UT Generator* ranks the methods in the *Property Relationship Method Set*, retrieves their corresponding test cases, and generates unit tests. The generated tests are added to the *UT Set*. After generating tests for all focal methods, the *Iterative Strategy* is

applied. The newly generated tests are treated as historical tests for the next iteration to generate new unit tests.

4.1 Preliminaries

In the context of property relationship analysis and unit test generation, there are three critical challenges to address:

- **Where to find?** In property 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 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 the language model (LLM), we need to prioritize and compress the relevant code data while preserving essential context.

To address these challenges, our methodology incorporates two key processes: *Metainfo Extraction* and *Static Context Retrieval*, both of which are essential for ensuring that the LLM has the necessary code structure and context for accurate analysis and test case generation.

4.1.1 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 property relationship analysis and test case generation. We also design schemas for other code entities, such as test classes, interfaces, abstract classes, and records.

```

1 {
2   "uri": "src/main/java/org/redisson/transaction/
3   operation/set/MoveOperation.java.MoveOperation",
4   "name": "MoveOperation",
5   "file_path": "src/main/java/org/redisson/
6   transaction/
7   operation/set/MoveOperation.java",
8   "superclasses": "SetOperation",
9   "methods": [
10    "[void]commit(CommandAsyncExecutor)",
11    "[void]rollback(CommandAsyncExecutor)"
12  ],
13   "class_docstring": "@author Nikita Koksharov",
14   "original_string": "", // Omitted
15   "super_interfaces": [],
16   "fields": [
17     {
18       "docstring": "",
19       "modifiers": "private",
20       "marker_annotations": [],
21       "type": "String",
22       "name": "destinationName"
23     }
24  ]
25 }
```

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

4.1.2 Reference Resolution Based on

Scope Graph. In addition to storing the code structure, resolving references to external code entities is essential for understanding and generating accurate test cases. Our approach, *Reference Resolution Based on Scope Graph*, addresses the challenge of determining “What to look for?” by identifying external variables, functions, or classes required to fully comprehend a code snippet. This process enhances the LLM’s ability to analyze code by completing the context necessary for code understanding.

Traditional reference resolution often depends on language-specific libraries, such as RepoFuse [24], which relies on tools like Jedi [20] for Python auto-completion. These approaches, while effective, are limited in scope, supporting only specific languages. The Language Server Protocol (LSP) [31] improves on this by providing cross-language support for features like go-to-definition and find-references, but it introduces significant overhead due to its reliance on language parsers, making it slow and resource-intensive. Given these constraints, a lightweight and language-agnostic solution is essential for supporting large-scale codebases.

To solve this problem, APT extends the capabilities of Scope Graphs [32], which originally focus on resolving variable definitions within a single scope. While traditional scope graphs struggle with cross-class references, APT enhances this by identifying unresolved references across classes and files, completing lookups using the *Metainfo Database*.

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

- Nodes N represent entities such as:
 - `LocalScope`: A lexical scope within the code.
 - `LocalDef`: A local definition (e.g., variable, function, or class).
 - `LocalImport`: An import statement introducing external symbols.
 - `Reference`: A usage of a symbol that must be resolved to its definition or import.
- Edges E describe relationships between these entities:
 - `ScopeToScope`: A parent-child relationship between scopes.
 - `DefToScope`: Links a definition to the scope it belongs to.
 - `ImportToScope`: Connects imports to the target scope.
 - `RefToDef`, `RefToImport`: Capture references to definitions or imports.

Graph Construction and Reference Resolution. APT construct graphs by the following steps:

- (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) **Reference Resolution:** When a symbol is referenced, the graph is traversed to find its corresponding definition or import via `RefToDef` or `RefToImport` edges. Unresolved references are flagged if no match is found in the current or parent scopes.

Algorithm for Reference Resolution. APT resolve references Ref within a code block cb via:

- (1) **Reference Insertion:** For each reference r in the cb , associate it with its lexical scope s and insert it into the Scope Graph:

`insertRef(r, s)`

- (2) **Declaration Lookup:** For each reference r , search for a matching declaration in the current scope s . If none is found, traverse the graph through `ScopeToScope` edges to the parent scopes, recursively searching for the definition:

`findDecl(r, s)` (traverse through parent scopes)

If no declaration is found in the entire scope chain, extend the search via `RefToImport` edges to check for imports:

`findDecl(r, s)` (via imports)

- (3) **Unresolved Reference Detection:** If no matching declaration is found, mark the reference as unresolved and collect all such unresolved references:

$$UR = \{r \in Ref \mid resolve(r) = \emptyset\}$$

- (4) **Unresolved Reference Retrieval from MetaInfo Database:** For each unresolved reference $r \in UR$, APT queries the *MetaInfo Database* to retrieve its definition from other classes or files and aggregates the results in `resolveRef(cb)`:

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

4.1.3 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, APT uses two mechanisms tailored for different scenarios: *Class Montage* and *Class Shrink*.

Class Montage. When analyzing test cases, 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 *Property Relationship Method Set* $R(m_t)$ through *property-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, APT 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` class¹, 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.

4.2 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:

$$c = \{tc_1, tc_2, \dots, tc_n\} + \{Fixture_1, Fixture_2, \dots, Fixture_m\} + Imports(c) + ClassMembers(c)$$

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 $tb(c, i)$ for a test case tc_i in test class c is defined

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

<pre> 1 public class RedisCommonBatchExecutor { 2 private final Entry entry; 3 private final BatchOptions options; 4 private static int timeout(5 ↪ ConnectionManager 6 ↪ connectionManager, BatchOptions 7 ↪ options); 8 private static int retryInterval(9 ↪ ConnectionManager 10 ↪ connectionManager, BatchOptions 11 ↪ options); 12 private static int retryAttempts(13 ↪ ConnectionManager 14 ↪ connectionManager, BatchOptions 15 ↪ options); 16 @Override 17 protected void sendCommand(18 ↪ CompletableFuture<Void> 19 ↪ attemptPromise, RedisConnection 20 ↪ connection); 21 } </pre>	<pre> 1 public class RedisCommonBatchExecutor { 2 private final Entry entry; 3 public RedisCommonBatchExecutor() { 4 // Constructor 5 } 6 private static int retryInterval(7 ↪ ConnectionManager 8 ↪ connectionManager, BatchOptions 9 ↪ options) { 10 // Implementation omitted 11 } 12 private static int retryAttempts(13 ↪ ConnectionManager 14 ↪ connectionManager, BatchOptions 15 ↪ options) { 16 // Implementation omitted 17 } 18 } </pre>
---	---

(a) RedisCommonBatchExecutor's Class Montage

(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.

as the combination of the test case, its related fixtures, imports, and class members:

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

The set of test bundles is denoted as TB, including all $tb(c, i)$ for every test class c and test case 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

```

{
  "file_path": "src/test/.../JCacheTest.java",
  "name": "JCacheTest",
  "dependencies": [...],
  "class_members": {
    "variables": [...],
    "methods": [],
    "nested_classes": ["ExpiredListener"]
  },
  "fixtures": ["beforeEach"],
  "test_cases": [{
    "name": "testClose",
    "primary_tested": ["Cache.close()"],
    "external_dependencies": {...},
    "fixtures_used": ["beforeEach"],
    "project_specific_resources": [
      "TestUtil.logTestResult(String, int)"
    ]
  }],
  ...
}

```

Fig. 9. Simplified test case analysis result for JCacheTest

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.

4.3 Property Retrieval

To address the *property retrieval* challenge in section 3.2.1, we first define the property and then present the analysis of property relationships within a class and across classes.

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

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

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

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

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

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

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

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

In cases where all three conditions are met:

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

4.3.2 Class-Level Property Analysis (Intra-Class). *Class-Level Property Analysis* focuses on identifying property 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 $M(c_t) = \{m_1, m_2, \dots, m_n\}$, including any inherited methods from parent classes, which are recursively incorporated via the *MetaInfo Database*. The goal is to find methods m_r with property relationships to m_t across the *Given*, *When*, and *Then*.

As illustrated in Figure 10, the formal version of the prompt for property analysis is provided. First, in the “Task Description” section, we define the task for the LLM, which includes the property definition and instructing the LLM to find the set of methods that have property relationships with the focal method m_t . The core of the prompt lies in leveraging *Rule-based Reasoning* [40] and *Chain of Thought (CoT) reasoning* [7], as shown in the “Steps” section of Figure 10. Instead of directly identifying all property relationships for a method m_t , which can overwhelm LLM due to complex inter-method dependencies and relationships, causing it to lose focus, we decompose the problem into smaller manageable parts and let LLM solve it step by step.

// Task Description Given focal method m_t , its class c_t and definition of property: $P(m_r, m_t)$, find a set of m_r and corresponding property description
// Steps 1. Expect test cases for m_t : TC(m_t) 2. For m_r in $M(c_t)$: - Expect test cases for m_r : TC(m_r) - Compare TC(m_r) and TC(m_t): $P_{\text{Given}}(m_r, m_t)$: Preconditions or setup of m_r can give insights to m_t $P_{\text{When}}(m_r, m_t)$: Invocation patterns are similar to m_t $P_{\text{Then}}(m_r, m_t)$: Assertions or exception handling are relevant to m_t - If $P_{\text{Given}}(m_r, m_t) \vee P_{\text{When}}(m_r, m_t) \vee P_{\text{Then}}(m_r, m_t)$: Add m_r and description of P to gwt set - If $P_{\text{Given}}(m_r, m_t) \wedge P_{\text{When}}(m_r, m_t) \wedge P_{\text{Then}}(m_r, m_t)$: Add m_r and description of P to complete set
// Input and Output - Input: m_r , resolveRef(m_r) 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.

Specifically, the LLM is first instructed to imagine a set of expected test cases $TC(m_t)$ for the focal method m_t by reasoning. Then, it traverses all methods m_i in the class c_t . For each method m_i , the LLM imagines the test cases $TC(m_i)$, compares $TC(m_t)$ with $TC(m_i)$, and identifies potential property relationships in the *Given*, *When*, and *Then* phases. If a relationship is found in any phase, the method m_i and its corresponding relationship description are added to the *gwt set*. If relationships are found in all three phases, they are added to the *complete set*. Finally, the input to the LLM consists of the focal method m_t , the external reference information retrieved through $resolveRef(m_t)$, and the source code of class c_t , including any inherited methods from parents. The LLM is tasked with outputting the *gwt set* and *complete set* as structured JSON.

The generated JSON output provides a clear mapping of property relationships between methods. As shown in Table 1, it highlights the methods that share property 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 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()`.

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

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

It is important to note that the relevance of property 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 property relationships that can enhance test case generation.

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

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

General Property Relationship Formula. For any two classes c_1 and c_2 related by inheritance, sibling class, or common interface implementation, and for two methods $m_A \in M(c_1)$ and $m_B \in M(c_1)$, the general condition for extending property relationships across classes is as follows:

$$P(m_A \in c_1, m_B \in c_1) \wedge m_A \in M_{\text{shared}} \implies P(m_A \in c_2, m_B \in c_1)$$

This formula means that if a method $m_A \in c_1$ enhances method $m_B \in c_1$, and m_A belongs to the shared methods M_{shared} between c_1 and c_2 , then m_A from c_2 can also enhance m_B in c_1 . This provides a unified way to express how shared methods across related classes contribute to test generation.

Inheritance Relationships. For a parent class c_p and its child class c_c ,

- $M_{\text{shared}} = M(c_p) \cap M(c_c)$ represent shared method set between the parent and child classes.

Shared methods between parent and child classes can enhance methods in either class. If a shared method $m_A \in M_{\text{shared}}$ enhances $m_B \in c_c$, then the corresponding method in c_p can similarly enhance m_B , and vice versa. This bidirectional relationship follows from the general property relationship formula and enables mutual test case enhancement across the inheritance hierarchy.

Sibling Class Relationships. For sibling classes c_1 and c_2 , which share a common parent class,

- $M_{\text{shared}} = M(c_1) \cap M(c_2)$ represent shared method set between these sibling classes.

If $P(m_A \in c_1, m_B \in c_1)$ holds for a shared method $m_A \in M_{\text{shared}}$, then $m_A \in c_2$ can enhance $m_B \in c_1$, following the general formula.

Common Interface Implementations. For classes c_1 and c_2 that implement the same interface I , let:

- $M_{\text{shared}} = M(c_1) \cap M(c_2) \cap M(I)$ represent method set shared by both classes and the interface.

If $P(m_A \in c_1, m_B \in c_1)$ holds for a shared method m_A , then the corresponding method $m_A \in c_2$ can also enhance $m_B \in c_1$, as per the general formula.

Consolidation. In all cases, the enhancement of property relationships depends on shared methods M_{shared} . If $m_A \in M_{\text{shared}}$ enhances m_B in one class, the same method m_A from a related class can enhance m_B as well, ensuring property relationships extend across class boundaries and support more robust test generation.

Additionally, in section 4.3.2, when the LLM identifies methods from external classes that can be directly reused (e.g., when the focal method calls an external function in a substitutive manner), the ‘‘External’’ field in JSON is included to indicate whether the related method comes from an external class, and this also facilitates inter-class property analysis.

4.4 Iterative Test Case Generation

In this subsection, we introduce the approach to generating effective test cases for the focal method, solving *Generating Effective Unit Tests* challenge in section 3.2.2.

4.4.1 Property-Based Retrieval and Ranking. Here, we explain the process of generating test cases for the focal method m_t by leveraging property relationships and a ranking strategy.

Algorithm 1 outlines the process for generating a unit test. The inputs are the focal method m_t , the property relation method set $R(m_t)$ (from the *property analysis*), and the test bundle set TB .

APT first obtains methods with test cases in $R(m_t)$ to create S_{origin} (line 7 - 9), then ranks them to form S_{ranked} (line 10). It prioritizes methods at the class level over those at the repository level. Within each level, *complete* relationships are selected first, with up to N methods chosen. Afterward, for the *Given*, *When*, and *Then* phases, APT selects up to N methods per phase, ensuring at least one per phase if available. The final ranking order is: *class-level complete* > *class-level GWT* > *repo-level*

Algorithm 1: Test Case Generation for a Focal Method

Input: Focal method: m_t ; Property relation method set: $R(m_t)$; Existing test bundle set: TB ; Property-based retrieval prompt: *property_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 their property relationship to  $m_t$ 
3  $S_{ranked} \leftarrow \emptyset$  // Store ranked methods with test bundles and their property relationship to  $m_t$ 
4  $D \leftarrow \{\}$  // Dictionary (ranked method  $\rightarrow$  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  $S_{ranked} \leftarrow \text{rank}(S_{origin})$  // intra-class complete > intra-class GWT > inter-class complete > inter-class GWT

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

25  $ut_t \leftarrow \text{CallLLM}(I)$ 

```

complete > *repo-level* 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, APT begins packing the context. First, based on *Reference Resolution Using Scope Graph*, APT call $\text{resolveRef}(m_t)$ 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} , APT queries the metaInfo database (*QueryMetaInfo*) to get the detailed information of m_r and add them to SC (lines 17-18). Simultaneously, APT 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 property prompt, m_t , SC , S_{ranked} , and D_m . If S_{ranked} is empty, a fallback strategy is applied to generate test cases 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 [16, 26], APT 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.

4.4.2 Iterative Strategy. Having explained the approach to generating unit tests for a focal method, this section outlines the iterative strategy to optimize test generation across the entire project.

Given the set of all focal methods M in a project, the *Iterative Test Case Generation Strategy* begins by generating test cases for methods $m_A \in M$, if there exists a property 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 iteration, the newly generated test cases tc_A are added to the overall test suite TC . Once all applicable focal methods are processed in the current iteration, the newly generated test cases undergo *Metainfo Extraction* and *Test Case Analysis* to update the reference information. The process then continues to the next iteration, generating test cases for additional focal methods that satisfy $C(m_A, m_B)$. This iterative process continues, aiming to generate test cases for as many focal methods as possible using the Property-Based Retrieval Augmentation approach. Each round introduces new test cases that expand the number of focal methods covered until no further test cases can be generated for additional methods.

5 EVALUATION

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

- **(RQ1):** How effective is our approach in generating unit tests for a given focal method in terms of correctness and completeness?
- **(RQ2):** How well is the maintainability of the generated unit tests?
- **(RQ3):** How does the *iterative strategy* contribute to the performance of our approach?
- **(RQ4):** What patterns exist in *property relationships*, and what common characteristics can be identified among them?

5.1 Overall Effectiveness

To demonstrate the overall effectiveness of our tool, we compare it against state-of-the-art (SOTA) solutions. Previous approaches tend to generate large batches of test cases for an entire project, but in real-world development, users are more likely to need unit tests generated for individual focal methods. Therefore, in our experiments, we adopt a strategy where tests are generated for each focal method independently, allowing us to better measure the effectiveness of the tool in more practical, user-centric scenarios.

Dataset We utilize 4 moderately complex projects with >10,000 lines of code from the datasets provided by HITS [47] and ChatUnitest[50]. We additionally crawl 8 repositories from GitHub, bringing the total number of projects to 12. The selection criteria followed those outlined in HITS [47]. 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. Finally, we generate unit tests for 1,515 focal methods across 12 projects.

Metrics. Previous works typically generate all test cases in a single batch and then report the overall coverage for the entire project. Differently, our strategy is to input one focal method at a time and generate a test file specifically for the focal method. We run the generated tests separately and use JaCoCo² to calculate the coverage for the focal method.

The results of the execution are categorized as follows:

²<https://github.com/jacoco/jacoco>

Table 2. Test Generation Results Across Tools

Metric	APT (Our Tool)	ChatUnitTest	LLM (with default RAG)	Evosuite
Compilation and Run Errors	393 (25.9%)	923 (60.9%)	1155 (76.2%)	232 (59.6%)
Assert Errors	210 (13.9%)	228 (15.0%)	127 (8.4%)	53 (13.6%)
Successful Executions	912 (60.2%)	364 (24.0%)	233 (15.4%)	104 (26.7%)
Full Coverage	821 (54.2%)	323 (21.3%)	200 (13.2%)	99 (25.4%)
Total Methods Tested	1515	1515	1515	389

- **Compilation and Runtime Errors:** Compilation errors may arise from issues like undeclared variables, while runtime errors can include execution issues like uncaught exceptions.
- **Assertion Errors:** These refer to assertion errors where the expected value does not match the actual value. Note that, to measure the effectiveness of test generation tools, we assume the focal method is correctly implemented.
- **Successful Execution:** This means the generated test runs successfully. More specifically, **Full Coverage:** that both line and branch coverage are 100%.

Baseline and Configuration Previous SOTA tools like ChatUnitTest [50] and HITS [47] have set benchmarks in 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. Hence, we opt to compare our tool primarily with ChatUnitTest and the baseline large language model (LLM) using traditional Retrieval-Augmented Generation (RAG). For the LLM, we use DeepSeek-V2.5 [61], which offers an optimal balance between cost and performance. It is widely recognized and exceeds GPT-4 specifically in terms of coding capabilities [5]. Additionally, we include EvoSuite as another baseline in our comparison. The specific experimental parameters are as follows:

- For each LLM-based tool, before each run, we specify the focal method for which the tool generates unit tests, with a maximum of two repair rounds allowed. This process is repeated multiple times. Private methods and methods with only one line of effective code are filtered out. For methods with nested or anonymous inner classes, only the outer method is tested.
- To ensure fairness, APT generates a test file for each focal method with N test methods, similar to the baseline LLM. For ChatUnitTest, we use its default configuration to generate 5 test methods per focal method. The baseline LLM also receives the source code of the containing class as context (as the default RAG strategy), with the prompt adapted from ChatTester [56], effective for Java unit test generation.
- For EvoSuite [12], we use its latest runtime version 1.0.6, which supports only Java 8 and JUnit 4. As a result, the number of methods testable by EvoSuite is smaller compared to other tools. Specifically, EvoSuite is used to generate tests for 389 methods. Moreover, EvoSuite’s search budget (time limit) is set to 120 seconds, with all other parameters left at their default settings.

5.1.1 Test Generation Results. Table 2 shows the results of evaluated tools. In terms of **correctness**, APT demonstrated significantly fewer compilation and run errors (25.9%) compared to ChatUnitTest (60.9%) and the LLM with default RAG (76.2%). It also has the highest rate of successful executions at 60.2%, outperforming ChatUnitTest (24.0%), the LLM with default RAG (15.4%), and EvoSuite (26.7%). These results highlight the higher reliability of test cases generated by APT. In terms of **completeness**, APT achieved full coverage in 54.2% of cases, outperforming ChatUnitTest (21.3%), the LLM with default RAG (13.2%), and EvoSuite (25.4%). This demonstrates APT’s superior ability to generate comprehensive tests that provide broader coverage of the code under test.

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

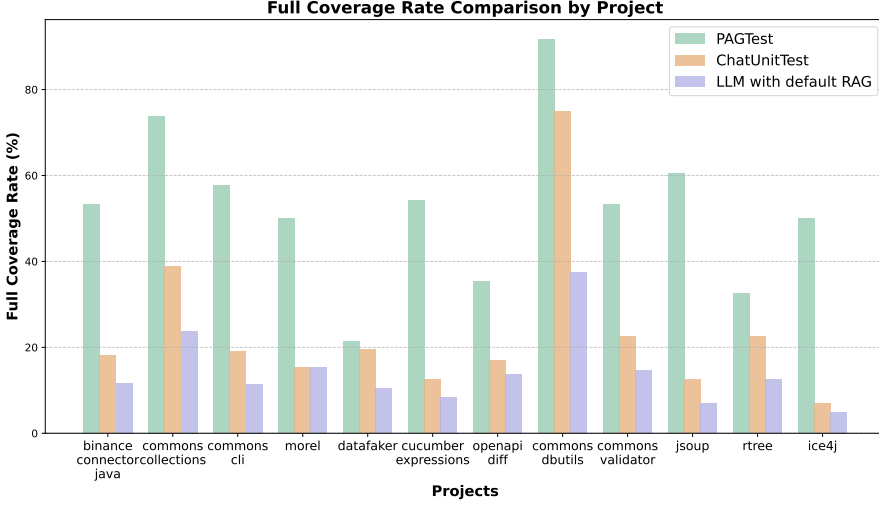


Fig. 11. Project-wise Full Coverage Rate of APT, ChatUnitTest, and LLM with default RAG

Figure 11 compares the full coverage rates across three LLM-based tools. APT consistently achieves higher full coverage rates across nearly all projects. For example, in `binance-connector-java`, APT achieves 53.3%, outperforming ChatUnitTest at 18.1% and the LLM with default RAG at 11.7%. Similarly, in `commons-collections`, APT reached 73.8%, significantly exceeding ChatUnitTest’s 38.9%, and the LLM with default RAG’s 23.8%.

An interesting observation is that projects with abstract or interface-heavy code pose significant challenges for all tools. For instance, in the `datafaker` project, which contains numerous abstract classes and interfaces, all tools experience higher failure rates. This occurs because the tools mistakenly attempted to instantiate abstract classes, which is not permissible in Java. In the `ice4j` project, the complexity of network protocols makes test generation particularly challenging. However, APT achieves a 50.0% full coverage rate, significantly outperforming ChatUnitTest’s 7.0%. This underscores the value of incorporating existing tests when generating new tests in projects with intricate dependencies and protocols.

5.2 Maintainability of Generated Unit Tests

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

Code Style Violations assess how well the generated tests adhere to coding standards. Based on Tang et al. [43], 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. [43], we used CheckStyle [6] for code style checking. We opt not to evaluate the tests produced by EvoSuite due to its strong reliance on custom auxiliary 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. We select 167 unit tests generated for methods that were fully covered by all three tools. These tests are integrated back into their original test suites for evaluation.

The analysis of code violations is shown in Figure 12, and the detailed breakdown is as follows:

- **Code Duplication:** APT introduces 17 instances, while ChatUnitTest and Github Copilot result in significantly higher duplication, with 152 and 146 violations, respectively.
- **Naming Convention:** APT has 21 violations, which is substantially lower than ChatUnitTest’s 191 and Github Copilot’s 92 violations.
- **Miscellaneous:** APT again performs better, with 49 violations compared to ChatUnitTest’s 70 and Github Copilot’s 54.

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

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 [41]. To further analyze the usability of the generated unit tests, we use PMD [36] used by Tang et al. [43] for mock detection. Table 3 compares the additional mocks detected by each tool. The results clearly show that APT introduces only 8 additional mocks, significantly fewer than ChatUnitTest, which detects 61 unnecessary mocks. Copilot, while more conservative than ChatUnitTest, still adds 42 unnecessary mocks—far more than APT. This highlights that APT produces the fewest redundant mocks, resulting in cleaner and more maintainable tests.

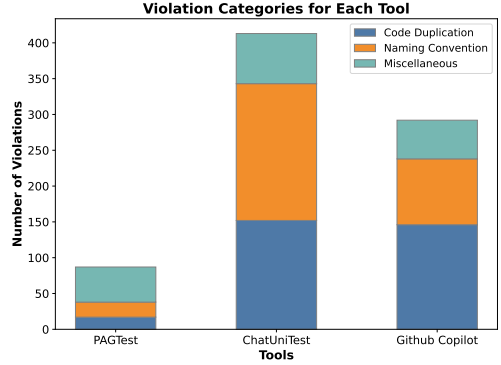


Fig. 12. Code style violations across different tools.

Table 3. Mock Detection Analysis Across Tools

Tool	Additional Mocks
APT	8
ChatUnitTest	61
Github Copilot	42

5.3 Impact of Iterative Strategy

To demonstrate the impact of the Iterative Incremental Strategy (IS), we conduct an ablation study on the projects from RQ1 (Section 5.1).

As shown in Table 4, IS significantly reduces the **Compilation and Run Error 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%, and jsoup drops from 26.2% to 11.3%. Similarly, showing that IS 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%, indicating a reduction in logic errors within test assertions. For the **Successful Execution Rate**, IS demonstrates notable improvements. Projects like commons-collections and

Table 4. Comparison of Performance Metrics with and without Iterative Strategy (IS)

Project Name	Compilation and Run Error		Assertion Error		Successful Executions		Full Coverage	
	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%

jsoup experience increases from 61.7% to 77.2% and from 58.5% to 76.2%, respectively. This shows that IS enhances the ability to generate tests that successfully execute, covering more code paths without encountering runtime issues. The **Full Coverage 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%. These results confirm that IS not only helps generate executable tests but also boosts their ability to provide full coverage of the tested code.

Overall, the results confirm that the Iterative Incremental Strategy improves test quality by reducing errors and enhancing both coverage success and overall coverage.

5.4 Exploring Patterns in Property Relationships

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

5.4.1 Study Design. To minimize bias and ensure reliability, we employ several rigorous strategies in our study design, referencing methodologies from prior work [37, 51]:

- **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 property relationships, the review process, and the criteria for validation. Example cases were used to illustrate expected outcomes.
- **Cross-Validation:** Each method relationship generated by the LLM is reviewed independently by two participants. In cases where their assessments differed, a third researcher is involved in resolving the disagreement through discussion. This ensured a high level of agreement.
- **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.

5.4.2 Analysis and Implications. As shown in Table 5, we select the top 6 patterns in property 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 5. Patterns in Property Relationships

Pattern	Description	Examples	Phases	Count
Structural Similarity	Methods share internal structures or control flows, even if operations differ.	<code>readFile()</code> vs. <code>writeFile()</code> : 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.	<code>fetchUserName()</code> vs. <code>fetchUserEmail()</code> : Both retrieve user information but return different data types.	Given, When, Then	617
Substitutability	Methods serve the same purpose with different implementations.	<code>mergeSort()</code> vs. <code>quickSort()</code> : Both sort data but use different algorithms.	Given, When, Then	124
Exception Handling Similarity	Methods handle exceptions in similar ways but under different contexts.	<code>connectToServer()</code> vs. <code>downloadFile()</code> : Both handle <code>TimeoutException</code> , though one is for establishing a server connection and the other is for downloading a file.	Then	357
Resource Access Similarity	Methods access the same resource but perform entirely different operations on it.	<code>lockFile()</code> vs. <code>deleteFile()</code> : 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.	<code>initializeIterator()</code> vs. <code>asMultipleUserIterable()</code> : Both require iterator initialization.	Given	351

6 DISCUSSION

6.1 Threats to Validity

The threats to the validity of APT primarily involve its generalization to other LLMs and the diversity of the projects used in the evaluation. Due to budget constraints, we utilize DeepSeek V2.5, which offers a balance between performance and cost. However, using different LLMs may yield varying results and could impact the overall effectiveness of the tool. Additionally, the range of projects may introduce bias, as they vary in complexity and existing unit test coverage. The presence of existing unit tests can affect APT’s performance, as it relies on these tests to establish property relationships and guide the generation process. Future work will involve expanding the dataset and testing with other LLMs to better assess the generalizability of APT.

6.2 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 *property relationships*, it leverages task-specific context effectively.

Extend to Other Code-Related Tasks. At a higher level of abstraction, our approach extends beyond unit test generation. By defining *property relationships* between methods within code-specific tasks, the framework is applicable to other code-related tasks, not limited to test generation.

7 RELATED WORK

7.1 Unit Test Generation

Unit testing focuses on testing individual hardware or software units, or groups of related units [33]. Several test case generation approaches have been proposed, including random-based methods [9, 48] and constraint-driven techniques [19]. Approaches such as DART [14] and KLEE [29], which leverage symbolic execution, often face challenges like path explosion [3]. Search-based software

testing (SBST) techniques [8, 11, 23, 25, 42, 52, 60], such as EvoSuite [1] and evolutionary testing [44], mitigate some of these issues but struggle with broad search spaces and high computational costs [30]. In contrast, deep learning-based methods [4, 11, 46, 54, 59], like AthenaTest [45], leverage neural models to generate diverse test inputs and better capture functional program intent.

Recently, large language model (LLM)-based techniques have gained popularity in testing [10, 17]. Xie et al. [50] introduced ChatUniTest, which uses ChatGPT to generate Java-compliant test cases after static analysis. Yuan et al. [56] proposed ChatTester, which refines initial test cases with ChatGPT, sometimes outperforming traditional SBST methods. Hybrid approaches like TELPA [53], uses program analysis with LLM by integrating refined counterexamples into prompts, guiding the LLM to generate diverse tests for hard-to-cover branches, and HITS [47] decomposes focal methods into slices and ask the LLM to generate tests slice by slice, improving coverage for complex methods. While these tools have demonstrated promising results in specific scenarios, they often struggle with deeply understanding the focal method and 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.

7.2 Retrieval-Augmented Generation (RAG) for Code Tasks

RAG systems, particularly Retrieval-Augmented Code Generation (RACG), enhance large language models (LLMs) by retrieving relevant code snippets or structures from repositories [21]. Existing approaches include similarity-based methods [18, 39], such as using the BM25 algorithm to perform similarity retrieval [22], as well as vector similarity-based retrieval [35]. Additionally, AST-based retrieval approaches, such as AutoCodeRover [58], leverage AST structures to retrieve relevant code contexts. Some methods design specific tools to perform more targeted retrieval, as in the case of MASAI [2], while CODEXGRAPH [27] 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” [24]. 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 test generation.

8 CONCLUSION

We presented *Property-Based Retrieval Augmentation for Unit Test Generation*, a novel approach that enhances the correctness, completeness, and maintainability of unit tests by leveraging property 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. APT further employed an iterative strategy, using newly generated tests to guide future ones. Our evaluation across 12 open-source projects demonstrated that APT 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.

References

- [1] 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.
- [2] 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).

- [3] 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.
- [4] 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.
- [5] ChatBotArena. 2024. "Chatbot Arena LLM Leaderboard: Community-driven Evaluation for Best LLM and AI chatbots". <https://lmarena.ai/> Accessed: 2024.
- [6] Checkstyle. 1999. "Code conventions for the java programming language". <https://checkstyle.sourceforge.io/> Accessed: 2024.
- [7] Jiu hai 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).
- [8] 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.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] Github. 2024. "The world's most widely adopted AI developer tool". <https://github.com/features/copilot> Accessed: 2024.
- [14] 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.
- [15] 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.
- [16] 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).
- [17] 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.
- [18] 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).
- [19] 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.
- [20] jedi. 2024. "Awesome autocompletion, static analysis and refactoring library for python". <https://github.com/davidhalter/jedi> Accessed: 2024.
- [21] 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).
- [22] 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).
- [23] 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.
- [24] 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>
- [25] 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.
- [26] 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).

- [27] 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).
- [28] Stephan Lukaszcyk 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.
- [29] 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.
- [30] 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.
- [31] microsoft. 2024. "Language Server Protocol". <https://microsoft.github.io/language-server-protocol> Accessed: 2024.
- [32] Pierre Neron, 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.
- [33] Michael Olan. 2003. Unit testing: test early, test often. *Journal of Computing Sciences in Colleges* 19, 2 (2003), 319–328.
- [34] 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.
- [35] 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).
- [36] Pmd. 2023. "An extensible cross-language static code analyzer". <https://pmd.github.io/> Accessed: 2023.
- [37] Dezhi Ran, Hao Wang, Zihong 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.
- [38] redisson. 2024. "Easy Redis Java client and Real-Time Data Platform". <https://github.com/redisson/redisson> Accessed: 2024.
- [39] 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.
- [40] 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).
- [41] Davide Spadini, Mauricio 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.
- [42] 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.
- [43] 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).
- [44] Paolo Tonella. 2004. Evolutionary testing of classes. *ACM SIGSOFT Software Engineering Notes* 29, 4 (2004), 119–128.
- [45] 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).
- [46] 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.
- [47] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing.
- [48] 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.
- [49] wikipedia. 2024. "Given-When-Then". <https://en.wikipedia.org/wiki/Given-When-Then> Accessed: 2024.
- [50] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. ChatUniTest: a ChatGPT-based automated unit test generation tool.
- [51] 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. *arXiv preprint arXiv:2408.15815* (2024).
- [52] Rahul Krishna Yandrapally and Ali Mesbah. 2022. Fragment-based test generation for web apps. *IEEE Transactions on Software Engineering* 49, 3 (2022), 1086–1101.
- [53] C. Yang, J. Chen, B. Lin, and et al. 2024. Enhancing LLM-based Test Generation for Hard-to-Cover Branches via Program Analysis. *arXiv preprint arXiv:2404.04966* (2024).

- [54] 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.
- [55] 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.
- [56] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No more manual tests? evaluating and improving chatgpt for unit test generation.
- [57] 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).
- [58] 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.
- [59] 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.
- [60] 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.
- [61] 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.