Knowledge Matters: Injecting Project and Testing Knowledge into LLM-based Unit Test Generation

Anji Li
School of Software Engineering, Sun Yat-sen University
Zhuhai, Guangdong, China
lianj8@mail2.sysu.edu.cn

Mingwei Liu*
School of Software Engineering, Sun Yat-sen University
Zhuhai, Guangdong, China
liumw26@mail.sysu.edu.cn

Zhenxi Chen
School of Software Engineering, Sun Yat-sen University
Zhuhai, Guangdong, China
chenzhx236@mail2.sysu.edu.cn

Zheng Pei
School of Software Engineering, Sun Yat-sen University
Zhuhai, Guangdong, China
peizh3@mail2.sysu.edu.cn

Zike Li
School of Software Engineering, Sun Yat-sen University
Zhuhai, Guangdong, China
lizk8@mail2.sysu.edu.cn

Dekun Dai
School of Software Engineering, Sun Yat-sen University
Zhuhai, Guangdong, China
daidk@mail2.sysu.edu.cn

Yanlin Wang
School of Software Engineering, Sun Yat-sen University
Zhuhai, Guangdong, China
wangylin36@mail.sysu.edu.cn

Zibin Zheng
School of Software Engineering, Sun Yat-sen University
Zhuhai, Guangdong, China
zhzibin@mail.sysu.edu.cn

Abstract

Automated unit test generation using large language models (LLMs) holds great promise but often struggles with generating tests that are both correct and maintainable in real-world projects. This paper presents KTester, a novel framework that integrates project-specific knowledge and testing domain knowledge to enhance LLM-based test generation. Our approach first extracts project structure and usage knowledge through static analysis, which provides rich context for the model. It then employs a testing-domain-knowledge-guided separation of test case design and test method generation, combined with a multi-perspective prompting strategy that guides the LLM to consider diverse testing heuristics. The generated tests follow structured templates, improving clarity and maintainability. We evaluate KTester on multiple open-source projects, comparing it against state-of-the-art LLM-based baselines using automatic correctness and coverage metrics, as well as a human study assessing readability and maintainability. Results demonstrate that KTester significantly outperforms existing methods across six key metrics, improving execution pass rate by 5.03% and line coverage by 11.67% over the strongest baseline, while requiring less time and generating fewer test cases. Human evaluators also rate the tests produced by KTester significantly higher in terms of correctness, readability, and maintainability, confirming the practical advantages of our knowledge-driven framework.

CCS Concepts

• **Software and its engineering → Software testing and debugging**; *Software development process management*.

Keywords

Unit Test, Large Language Models, Domain Knowledge

*corresponding author.

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3787769

ACM Reference Format:
Anji Li, Mingwei Liu, Zhenxi Chen, Zheng Pei, Zike Li, Dekun Dai, Yanlin Wang, and Zibin Zheng. 2026. Knowledge Matters: Injecting Project and Testing Knowledge into LLM-based Unit Test Generation. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3787769

1 Introduction

Unit testing is a fundamental practice in software development that verifies whether a method behaves as expected, playing a key role in ensuring software correctness, maintainability, and enabling regression testing [39]. As the first line of defense in the development lifecycle, it helps detect and localize bugs early, preventing their propagation [47]. However, writing high-quality unit tests manually is often time-consuming and error-prone, as it requires developers to deeply understand the method's behavior, construct valid input states, and define precise assertions [22].

For a method under test (*i.e.,* often called the focal method), a well-constructed unit test usually has two parts: the test prefix, which prepares the objects and environment needed to bring the system into a testable state, and the test oracle, which verifies that the actual output matches the expected behavior [16]. High-quality unit tests improve software reliability and also serve as living documentation, facilitating code comprehension and maintenance [31].

To reduce manual effort, automated test generation techniques have been proposed to generate a suite of unit tests with the main goal of maximizing the coverage in the software under test. Traditional unit test generation techniques include search-based [17, 24, 35], constraint-based methods [21, 28, 57], and random-based strategies [44, 59]. While effective in achieving code coverage, the generated tests are often hard to read and maintain, limiting practical adoption [12].

Recently, large language models (LLMs) have shown great promise in generating more human-like test code. These LLM-based methods demonstrate promising capabilities in understanding code semantics and synthesizing test code without explicit test specifications. Despite this potential, current LLM-based methods still suffer from key limitations that hinder their practical adoption due to a fundamental lack of essential knowledge. In particular, these models often lack access to **project-specific knowledge**, such as how to correctly instantiate and use classes, or how utility methods

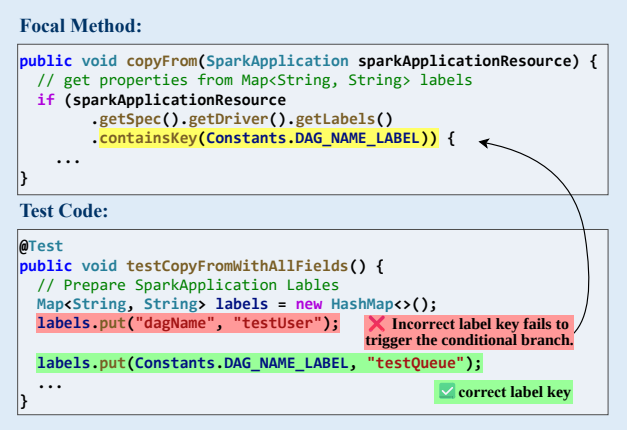

Figure 1: Motivational Examples(a): Setting Tested Object Incorrectly

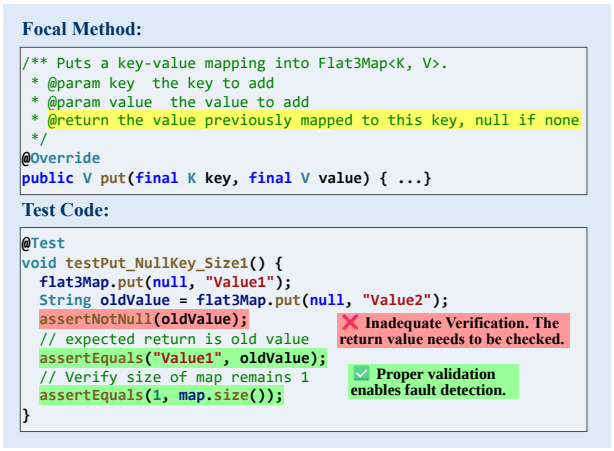

Figure 2: Motivational Examples(b): Insufficient Assert Statements

and APIs interact across modules. They also overlook **testing domain knowledge**, including core principles like boundary value analysis, exception handling, or the separation between test design and implementation. **This lack of both project-specific and testing-domain knowledge often results in test code that is unreliable and difficult to maintain.**

As illustrated in Figure 1-3, these issues commonly stem from the lack of project-specific knowledge and test domain expertise in existing LLM-based test generation methods. For instance, Figure 1 shows an incorrect construction of the *labels* Map used by *SparkApplication*, inserting the key "dragName" instead of the correct constant *Constants.DAG_NAME_LABE*. This error arises from a lack of awareness of constructor dependencies and configuration requirements. Figure 2 depicts a test with insufficient assert statements, weakening its effectiveness. Meanwhile, Figure 3 shows an inconsistent test structure with hard-coded values and missing essential setup logic, which results in fragile and difficult-to-maintain code.

To overcome limitations in existing LLM-based unit test generation, we propose a novel framework that integrates project-specific knowledge with software testing domain knowledge to guide the test generation process. Our approach consists of two main phases: an offline knowledge extraction phase and an online test generation pipeline. In the offline phase, we perform static analysis on the target project to extract two key types of knowledge: project structure knowledge, which includes class definitions, method signatures, and field declarations; and project usage knowledge, which covers invocation patterns, dependencies, and related functions. This comprehensive project knowledge is used to provide rich contextual information that improves the relevance and accuracy of the generated tests. The online test generation pipeline involves five sequential steps: (1) test class framework generation, which sets up reusable scaffolding such as setup and teardown methods; (2) multi-perspective test case design, where test scenarios are created from complementary testing heuristics; (3) test method transformation, which converts structured test cases into executable test methods; (4) test class integration, assembling all generated methods into a coherent test class; and (5) test class refinement, which improves code quality and maintainability. Throughout these steps, the pipeline leverages both the extracted project knowledge and guidance from established software testing principles.

Our approach's core innovations are twofold. First, **the explicit incorporation of project-aware knowledge provides the LLM with rich, contextualized information about the codebase**, leading to more accurate and meaningful tests. Second, we introduce a **testing domain knowledge–guided separation of test design and test generation and multi-perspective test case design prompting strategy**. By decoupling the "what to test" from "how to test", we make the testing intent explicit and improve the semantic clarity and maintainability of the generated test code.

We validate our approach on several real-world open-source projects, comparing it against state-of-the-art LLM-based baselines using both automated metrics (e.g., execution pass rate, line coverage) and a human study assessing readability, maintainability, and test intent clarity. Our findings show that (1) our method consistently outperforms baselines across six key metrics, improving execution pass rate by 5.03% and line coverage by 11.67% compared to the strongest baseline, while takes shorter time and generate less test cases; (2) the modular test case transformation component, which embodies the testing-principle-guided design, has the largest individual impact, with removal causing a drop of 11.45% in execution pass rate and 13.39% in line coverage; and (3) human evaluators rate our generated tests significantly higher in correctness, readability, and maintainability, confirming the practical advantages of our knowledge-driven framework.

Our main contributions are:

- We propose a novel test generation framework that integrates project-specific knowledge with testing domain knowledge. By leveraging comprehensive project knowledge and applying a separation of test design and test code generation guided by testing domain knowledge, our approach produces more accurate and maintainable test cases.
- We introduce a multi-perspective prompting strategy along with testing-domain-knowledge-guided separation of test case design and test method implementation, enabling the

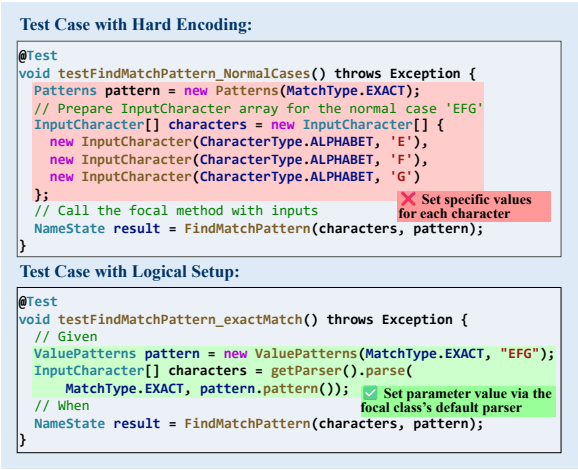

Figure 3: Motivational Examples(c): Hard Encoding Values

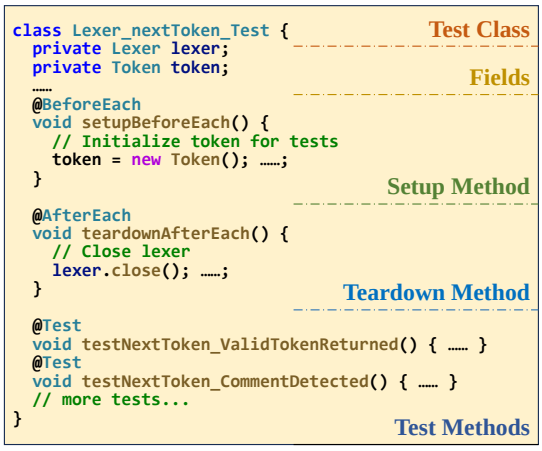

Figure 4: Test Class Example

LLM to generate semantically rich, logically structured, and purposeful tests.

- We perform extensive evaluation, including automatic metrics and human studies, demonstrating that our method consistently outperforms existing LLM-based approaches in correctness, readability, maintainability, and overall quality.

All data/code used in this study is provided in the package [9].

2 Approach

We present KTester, a knowledge-aware unit test generation framework that leverages both project-specific knowledge and testing domain knowledge to guide LLMs in generating high-quality, semantically meaningful, and maintainable unit tests.

Unlike prior LLM-based approaches that rely solely on the focal method as input and often generate brittle or unclear test cases, KTester decouples test design from test code generation, and injects knowledge at multiple stages to address key deficiencies of prior work (see Figure 1, 2 and 3). In particular, KTester is designed around the following core principles:

Project Knowledge Awareness. We statically analyze the entire codebase to extract structured information about classes, methods, usage patterns, dependencies, and documentation. This knowledge is later used to resolve object instantiation, locate related APIs, and reduce hallucinations during code generation.

Testing Knowledge Awareness. We explicitly guide the LLM to design test cases from multiple testing perspectives (e.g., control flow, boundary, exception handling), instead of treating the task as a black-box code-to-code translation.

Modular Generation Pipeline. We separate test class framework construction, scenario-specific test design, and final test case synthesis into distinct stages, improving modularity, reusability, and interpretability of generated tests.

The two-stages process of KTester generating the test class is showed in Figure 5. First, during the offline knowledge extraction stage (Section 2.1), we perform static analysis of the target project to build a structured knowledge base capturing essential project-specific information such as class hierarchies, method signatures, field access patterns, call relationships, and document comments. This knowledge base provides a foundation for accurate, context-aware test generation. Second, in the online test generation stage (Section 2.2), given a focal method, KTester retrieves relevant project knowledge and testing heuristics to construct a knowledge-rich prompt for the LLM. The LLM then produces a structured test class framework along with diverse test cases, which are validated and optionally repaired to ensure correctness and completeness. Figure 4 illustrates an example generated test class, including field declarations, setup and teardown methods, and multiple test methods.

2.1 Project Knowledge Extraction

To mitigate the limitations of LLMs in understanding project-specific semantics, KTester performs a comprehensive offline analysis to extract structured *project knowledge* from the codebase. This process is based on the assumption that unit tests are to be generated for methods within a project, and thus, knowledge beyond the focal method—such as class definitions, field usage, method invocations, and usage examples—is essential for meaningful test generation.

Specifically, KTester conducts two key analyses to construct a reusable project knowledge base: project structure knowledge mining, which captures the static architecture of the codebase—including class hierarchies, method signatures, and inter-method dependencies—and project usage knowledge mining, which extracts realistic invocation contexts that reveal how focal methods are constructed, initialized, and invoked in real code scenarios.

2.1.1 Project Structure Knowledge Mining. To enable knowledge-aware unit test generation, KTester statically analyzes the source code to mine structured *project structure knowledge*. This includes the architectural, semantic, and dependency-level information necessary for understanding the focal method's context and generating correct, maintainable test code. The analysis is based on abstract syntax tree (AST) parsing and code graph construction, allowing us to systematically extract and index key properties of classes and methods across the project.

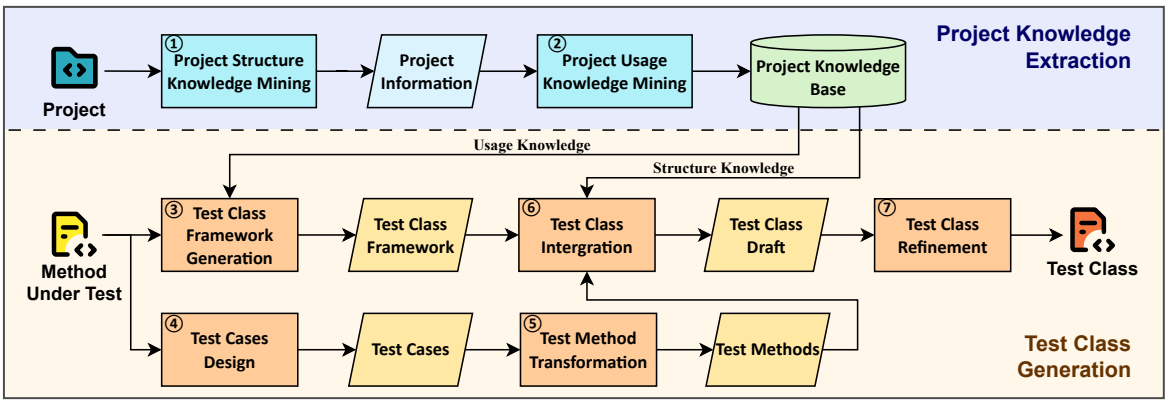

Figure 5: The Framework of KTester

Specifically, for each class and its methods or constructors, we extract the following information:

Structural Metadata. We collect the class name, package path, field declarations, constructor signatures, and method signatures (including parameter types and return types). This information enables the model to understand how to instantiate the target class or construct inputs for the method under test. For example, if a method requires a parameter of type `UserConfig`, this metadata reveals how `UserConfig` can be constructed via its fields and constructors.

Document Comments. We extract Javadoc-style comments associated with classes, constructors, and methods, as they often contain high-level semantic descriptions of behavior and usage constraints. These comments help LLMs infer testing intent and expected behavior. For instance, a comment like `/* Returns null if no user is found */` suggests that null should be included as a valid return value in test oracles.

Dependency Relations. For each method or constructor, we extract the list of invoked methods and accessed fields. These relationships are fundamental for understanding how the method interacts with its internal state or other classes. For example, if method `updateBalance()` internally calls `validateAccount()` and accesses `this.balance`, such dependencies indicate that tests should ensure the account is in a valid state and that balance updates are properly asserted. These dependency relations also support the identification of functionally related methods (used later in Usage Context Construction) and enable accurate call path tracing for generating invocation examples (Section 2.1.2).

2.1.2 Project Usage Knowledge Mining. To mitigate the limitations of LLMs in understanding project-specific usage patterns, we extract *Project Usage Knowledge* from the source code. This knowledge captures realistic invocation scenarios of the focal method, including (1) how its input parameters are constructed and (2) how the target object is initialized—both of which are essential for generating executable and semantically meaningful unit tests.

Such usage knowledge provides concrete, in-project examples that reflect actual calling contexts, enabling the LLM to synthesize inputs that align with real usage scenarios. For example, it reveals how the focal class is typically constructed in the project, which may involve complex initialization logic or dependency injection. It also uncovers how input parameters are derived—such as being composed from helper methods, shared resources, or intermediate computations—rather than being hardcoded literals. By grounding the generation process in real-world usage, this approach improves both the realism and maintainability of the generated test cases. As the code snippet illustrated in Figure 6, the focal method *findMatchPattern(InputCharactor[], Patterns)* is invoked inside *findMatchPattern(ValuePatterns)* after checking whether the variable *pattern* is an instance of *ValuePatterns* class, and called a Parser to parse this pattern into an *InputCharactor* array. Such code would be extracted as representative usage knowledge.

To mine project usage knowledge, we first identify the relevant caller context and then extract semantically meaningful execution traces showing how the focal method is invoked in practice.

Caller Method Discovery. For each focal method, we identify its *caller methods* using the static call graph built during the offline analysis. When the focal method is private or not directly reachable from public APIs, we trace the shortest call chain from an externally accessible method to the focal method. This ensures that the usage knowledge reflects a realistic and complete setup process, capturing how the focal method is constructed and triggered in actual usage.

Path-sensitive Usage Trace Extraction. For each discovered caller method, we construct a Control Flow Graph (CFG) [11] and perform intra-procedural slicing to extract execution paths from the method's entry point to the focal method invocation. By pruning unrelated branches and removing irrelevant operations, we isolate minimal yet semantically rich usage traces. These traces, which demonstrate how the focal class is instantiated and how arguments are prepared, are incorporated into the generation prompt as part of the Project Usage Knowledge. This provides the LLM with grounded, context-aware examples that help it synthesize test inputs aligned with the project's actual usage conventions.

2.1.3 Function-Level Index Construction. To enable scalable and consistent test generation, each method or constructor is transformed into a *knowledge unit* with its signature, documentation, and dependencies (invoked methods and accessed fields). These units are stored in an indexable format, forming a function-level knowledge base that enables efficient retrieval of related methods, usage contexts, and class-level information during prompt construction.

Invocation example of `findPattern(IputCharactor[],Patterns)`:

```
NameState findPattern(final Patterns pattern) {
  switch (pattern.type()) {
    case EXACT:
    case WILDCARD:
        assert pattern instanceof ValuePatterns;
        return findMatchPattern((ValuePatterns) pattern);
}

private NameState findMatchPattern(ValuePatterns pattern) {
  return findMatchPattern(
    getParser().parse(pattern.type(), pattern.pattern()),
    pattern);
}
```

Figure 6: Illustration of extracting invocation examples from a caller method via control flow pruning.

Unlike ad hoc, per-method extraction, our analysis is conducted *once per project* and reused across focal methods. The knowledge base is also *incrementally updatable*, requiring reprocessing only changed entities, which makes it practical for evolving codebases. While our method is language-agnostic, we focus on Java due to its popularity and mature analysis tooling[4, 7]. We use Spoon [7] to efficiently extract ASTs, control flow, and dependency graphs.

2.2 Test Class Generation

As illustrated in Figure 5, KTester generates a complete and executable test class for a given focal method by leveraging offline-extracted project knowledge and domain-specific testing expertise. It constructs a rich generation context that integrates structural program information, usage patterns, and unit testing heuristics, guiding the LLM to produce high-quality, realistic unit tests. The generation pipeline consists of five steps: test class framework generation, test case design, test method transformation, test class integration, and test class refinement.

Inspired by how developers and testers write unit tests in practice, KTester mimics the typical testing workflow—first planning test cases based on expected behaviors and coverage goals, then gradually transforming them into executable test code—rather than relying on a single-step code generation process.

2.2.1 Test Class Framework Generation. The first step prompts the LLM to generate a test class framework that establishes the necessary environment for unit testing. Unlike prior methods [20, 52] that directly prompt the LLM with a focal method and expect complete test cases in one shot, KTester decouples the generation process by first constructing a reusable, project-aware test class framework. This decision is grounded in a key insight: test generation is not merely a method-level translation task, but a context-sensitive activity requiring awareness of surrounding project structure, usage conventions, and domain-specific testing patterns. As illustrated in Figure 7, we design a prompt template grounded in project knowledge and test domain knowledge to guide this process. The prompt is composed of several key components, including core task and instructions, focal method and class, relevant project knowledge, test class template, and output specification.

Prompt For Test Class Framework Generation

You are tasked with **creating a framework for a test class in Java**. Now analyze the following information, and **complete the test class template** in the following requirements:

1. Provide the code wrapped in a Markdown code block "```java" at the beginning and "```" at the end.
2. Initialize necessary fields and dependencies within *setupBeforeAll()* and *setupBeforeEach()*.
3. Handle any required cleanup in *teardownAfterEach()* and *teardownAfterAll()*.
4. ...

Task Description

@input{focal method and target class}
Here is the code of focal method, *<method sigature>*, with the focal class *<class name>*:

```
Class <class_name> {
    dependent field declaration;
    dependent method signature;
    <method signature> {<code>} }
```

Code Under Test

@input{related context}
Document comment of focal class & focal method: *<javadoc>*
Constructors of focal class: *<constructor code>*
Constructors of param *<param class>*: *<constructor code>*
Usage examples of focal method: *<usage example code>*
Existing Test class: *<example test code>*

Relevant Project Knowledge

@input{test class template}

```
package <package name>;
import ...;
// Add other imports if necessary
Class <class_name> {
    @BeforeAll
    static void setupBeforeAll() {}
    @BeforeEach
    void setupBeforeEach() {}
    @AfterEach
    void teardownAfterEach() {}
    @AfterAll
    static void teardownAfterAll() {} }
```

Test Class Template

Figure 7: Prompt for test class framework generation.

The **core task and instructions** section provides high-level guidance to the LLM on setting up proper initialization and cleanup routines, enforcing access restrictions to target class internals, and ensuring syntactic correctness, thereby enabling a robust and maintainable test framework. The **focal method and class** section specifies the method under test and its containing class, providing the primary scope for framework generation. The **output specification** section defines the expected format and content of the generated framework, ensuring consistency and completeness.

To enrich the context, the **relevant project knowledge** section of the prompt draws from an offline-constructed knowledge base that captures both semantic and structural details of the codebase. This includes document comments for the focal class and method, which describe functionality, preconditions, and side effects to inform meaningful setup and teardown procedures. It also comprises constructors and parameter details, assisting the LLM in object instantiation and input preparation, particularly in complex scenarios. Additionally, usage knowledge mined from the codebase provides real-world usage patterns of the focal method and related classes, grounding the test framework in practical contexts. Finally, existing test classes associated with the focal class are incorporated as references for mocking strategies, resource management, and testing conventions, with only setup logic and class-level configurations extracted to avoid redundancy.

The **test class template** section outlines the structural skeleton, including annotations, lifecycle methods (e.g., setup and teardown), and placeholders for field declarations. This template, informed by

Prompt For Test Cases Design

As a professional Java software testing expert, your task is to **generate formatted test cases for a focal method**.
Following the guidance: *<conditional branch> or <functional behavoir> or <exception behavior>*

Conditional Branch Guidance	Functional Behavior Guidance	Exception-oriented Guidance
Create test cases to cover all **conditional branches** in the target method, including *if, else if, else, switch* and *case* statements.	Analyze the **method's functionality** and identify possible input combinations including **normal cases** and **boundary conditions**.	Analyze method behavior and identify ALL **possible exception scenarios**, then design test cases that trigger each identified exception.

@input{focal method and target class}
Here is the code of focal method, *<method sigature>*, with the focal class *<class name>*: *<code under test>*

@output{test cases}
Follow the JSON format provided here for the output:

```
[{
  "group": "test name",
  "cases": [{
    "input": [
      {"parameter": "param name", "value": "param value"},
      {...}],
    "expected": "expected exception or behavior",
    "description": "test scenario description"
  }, {...}]
},{...}]
```

Figure 8: Prompts for multi-view test case design.

practical experience in writing test code, serves as a starting point for the LLM to fill in with project-specific logic and configurations.

By integrating these components, KTester creates context-aware prompts that enable the LLM to generate reusable, well-structured test class frameworks aligned with project conventions. This modular approach enhances consistency and reduces errors common in one-shot generation. Separating framework from test logic also enables reuse across methods, minimizing redundant effort and reflecting practical testing workflows.

2.2.2 Test Cases Design. After generating the test class framework, the next step is to design high-quality test cases that cover different functional and non-functional behaviors of the focal method. In this phase, KTester focuses on the planning of test cases, rather than directly generating test code. Unlike the previous step, test case design solely leverages domain-specific testing knowledge, without relying on project-specific context. The only input is the focal method itself.

KTester adopts a multi-view guidance strategy grounded in established software testing principles. The LLM is explicitly instructed to design test cases from multiple perspectives—such as functional behavior, boundary conditions, and exception handling—and to organize them into logical groups based on shared testing intent. Each test case is represented internally in a machine-readable intermediate format that records structured natural-language descriptions of the scenario. In our implementation, this intermediate representation is realized using a lightweight JSON structure, and an illustrative example is provided in our replication package [9]. This design intentionally decouples test intent from concrete executable code, improving clarity and maintainability. The intermediate representations are subsequently translated into framework-specific test code.

To facilitate structured generation, the LLM is required to output grouped test cases, where each group targets a distinct aspect of the focal method. For example, one group may cover valid inputs for core functionality, another may focus on boundary values such as empty lists or null, and a third may address failure or exception-triggering scenarios. This grouped organization improves modularity and controllability during subsequent transformation steps and mirrors human testing practices, where related scenarios are often clustered for clarity and reuse.

To guide the LLM, we design three types of prompts, each corresponding to a classical testing perspective, as shown in Figure 8. These prompts share a common structure, with variations only in the design guidance section.

- **Condition branch-driven prompt**: This prompt encourages the LLM to explore different control-flow paths, such as conditional statements and loops, improving branch and path coverage.
- **Functionality-driven prompt**: This prompt focuses on the intended semantics of the method, helping the LLM generate test inputs that exercise typical behaviors and edge cases derived from parameter types and return values.
- **Exception-oriented prompt**: This prompt drive the generation of test cases for robustness and failure handling, including invalid or unexpected inputs that should trigger exceptions or error states.

This multi-view prompt design is modular and extensible—new testing views (e.g., performance, concurrency) can be incorporated by adding corresponding prompt templates. By separating test planning from execution and grounding the design in testing theory, this step enhances the quality, coverage, and interpretability of generated tests.

2.2.3 Test Method Transformation. After designing structured test cases, KTester transforms each test case group into an executable test method, aligning with how developers typically write tests—first decide what to test, then implement the logic.

Prompt Construction. We design a usage-aware prompt with several semantically distinct sections to guide the LLM in generating executable test methods, as shown in Figure 9. First, test case groups contain structured cases sharing a common testing intent (e.g., valid inputs, edge cases, exceptions), each with a scenario, inputs, and expected output. The LLM generates one test method per group to ensure modularity and clarity. Second, the usage context, retrieved from the project knowledge base (Section 2.1.2), provides focal method dependencies and related methods to guide realistic invocation and assertion generation. Lastly, the test class framework offers reusable setup/teardown code and field declarations, promoting consistency and eliminating redundancy.

Usage Context Retrieval. To enhance generation quality, we retrieve methods that are functionally related to the focal method and its dependencies, offering a broader context of real-world usage patterns. Based on our observations, related methods are identified based on shared usage of functions and fields. We define the similarity between two functions a and b using a Jaccard-based metric [36] by combining with method usage similarity $Sim_m(a, b)$ and field usage similarity $Sim_f(a, b)$:

$$Sim(a, b) = Sim_m(a, b) + Sim_f(a, b) \quad (1)$$

$$Sim_m(a,b) = \frac{|M(a) \cap M(b)|}{|M(a) \cup M(b)|} \quad (2)$$

$$Sim_f(a,b) = \frac{|F(a) \cap F(b)|}{|F(a) \cup F(b)|} \quad (3)$$

Here, $M(a)$ and $F(a)$ denote the sets of methods and fields used by function a, respectively. We select the top-N most similar functions to construct the context.

For each related class, we include: (1) the class declaration, optionally with Javadoc comments; (2) relevant field declarations accessed by dependent or related methods; (3) signatures of dependent methods with comments; and (4) signatures of related methods, annotated with markers indicating shared usage. This context provides the LLM with concise, relevant information to support realistic test generation.

2.2.4 Test Class Integration. After transforming individual test cases into test methods, the next step is to integrate them into a coherent and executable test class. This step ensures that all test methods, including newly generated ones and any accompanying setup or teardown logic, are correctly incorporated into the test class framework without introducing redundancy or conflicts.

To achieve this, KTester performs static analysis on the generated methods to identify their roles and resolve duplicates. The integration process proceeds as follows:

Setup/Teardown Merging. If a newly generated method is identified (via static analysis) as a setup or teardown method (e.g., annotated with `@Before`, `@BeforeEach`, etc.), and an existing method of the same type already exists, we merge their bodies. Specifically, any code in the new method that is not present in the original is appended, ensuring initialization routines are preserved and extended without duplication.

Test Method Deduplication. For test methods and general helper methods, we identify duplicates by method name. When two methods share the same name, we further compare their bodies and retain the longer implementation as a proxy for logical richness. This deduplication addresses a common pattern where the LLM first generates simple placeholder tests and later generates more complete versions with the same or similar names, ensuring that the final output preserves the more comprehensive implementation.

Non-conflicting Insertion. For all other non-conflicting methods, we append them directly to the test class. This modular addition supports test diversity and extensibility without disrupting the original structure. The integration process leverages standard AST-based static analysis tools to parse and manipulate the Java code structure safely and consistently. This automation ensures that the final test class is executable, maintainable, and free of duplicate or conflicting code.

By decoupling method generation from class integration, our approach maintains modularity while aligning with practical software engineering workflows, where tests are often extended iteratively and merged across sessions or contributors.

2.2.5 Test Class Refinement. In this stage, KTester validates the generated test class by checking compilation and execution errors. Despite the rich contextual guidance, LLM outputs may still contain issues such as hallucinated import statements, improper mocking, incorrect usage of private methods, or runtime errors during test execution. To address these problems, we adopt an iterative refinement process with static validation and dynamic repair.

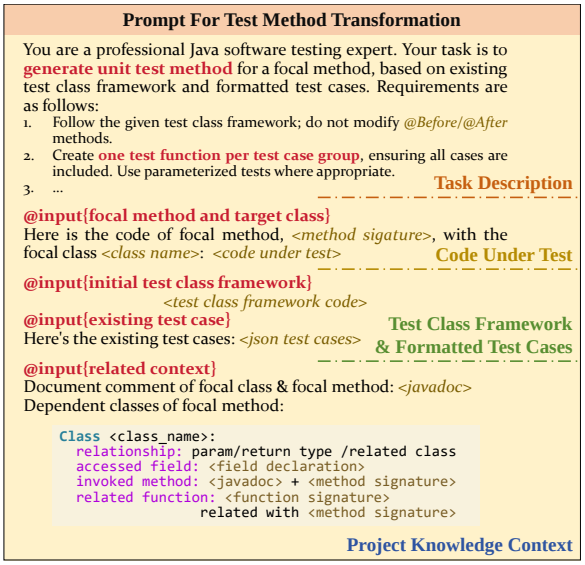

Figure 9: Prompt for test method trasformation.

Rule-based Repair. Rule-based repair focuses on correcting straightforward and common errors detected by static analysis, such as hallucinated or missing import statements and unresolved symbols. Leveraging the previously constructed project-level test knowledge base, KTester can automatically resolve dependencies by appending missing imports or correcting incorrect ones based on simple class names. The comprehensive indexing of classes, methods, and fields from the static analysis phase facilitates accurate dependency resolution without human intervention.

LLM-based Repair. For errors arising from semantic misunderstandings or subtle logic issues that cannot be fixed by static rules, KTester applies LLM-based repair using compile-time errors, runtime logs, and failing test feedback. This information, combined with the focal method context and relevant function signatures, is incorporated into a repair prompt that is fed back to the LLM. The model then performs targeted correction of issues such as incorrect assertions, missing exception handling, or logic bugs causing runtime failures.

Importantly, this refinement process extends beyond compilation to include execution feedback. If runtime errors occur or tests fail due to assertion mismatches, these failure details are fed into the repair pipeline to enable iterative correction and improvement of the generated test code. This dynamic feedback loop helps ensure that the final test class is not only syntactically correct but also functionally robust and reliable.

3 Evaluation

In this section, we evaluate the effectiveness of KTester by answering the following research questions (RQs):

RQ1 (Effectiveness Comparison): Does KTester outperform state-of-the-art baselines in terms of coverage scores and execution pass rate when testing complex methods?

RQ2 (Ablation Study): How do the key components of KTester contribute to its overall performance?

RQ3 (User Study): Are the tests generated by KTester more readable and maintainable than those produced by baseline methods?

3.1 Experimental Setup

3.1.1 Dataset. To evaluate the effectiveness of KTester, we adopt the HITS dataset [52], which is constructed from 10 popular open-source Java projects spanning various domains (e.g., microservices, command-line tools, event engines). This dataset specifically targets complex methods—defined as those with cyclomatic complexity greater than 10—making it well-suited for assessing how effectively LLMs handle methods with intricate control flow and dependencies. The dataset comprises 110 tasks, each consisting of a focal method and its containing class.

3.1.2 Baselines. We compare KTester with 4 state-of-the-art test generation methods: three are purely LLM-based, and one integrates search-based software testing (SBST) with LLMs. The LLM-based baslines adopt distinct strategies for context construction and test generation:

ChatUnitTest [20] provides the focal method and handcrafted context to the LLM, and applies iterative repair using error feedback from failed executions.

ChatTester [58] is similar to ChatUnitTest but differs in how it constructs the focal method's context, building it incrementally as needed.

HITS [52] decomposes the focal method into slices and prompts the LLM to generate test cases for each slice, which are then merged into a complete test suite. This fine-grained approach helps mitigate the difficulty LLMs face when reasoning about complex logic holistically.

For fair comparison, we adapt official or open-source implementations of these methods [5]. ChatUnitTest and ChatTester generate one test class per focal method, while HITS's slice-level tests are merged into a single class to ensure consistent execution and setup. All baselines and KTester use the same LLM backend—`gpt-4o-mini` [6]—with temperature=0.5, ensuring consistency in generation quality, efficiency, and cost.

The SBST–LLM hybrid method, **UTGen** [25], employs a traditional SBST tool to generate initial test cases, then leverages an LLM to rename functions and variables in order to improve test readability. Since UTGen rely on an LLM's ability on code understanding instead of enhancing code coverage, we follow the initial configuration provided in its replication package [10] (EvoSuite [30] as the generation tool and CodeLlama-7b [2] used for refinement) when conducting our experiments to evaluate its performance.

3.2 RQ1: Effectiveness Comparison

3.2.1 Design. We use the HITS dataset (Section 3.1.1), which contains 110 test generation tasks from 10 open-source Java projects. Our method builds the project knowledge base by analyzing the corresponding project source code versions in the dataset. For these tasks, both KTester and baseline methods generate unit tests using GPT-4o-mini as the backbone model. For fair comparison, KTester and all baselines generate exactly one test class per focal method, with no restriction on the number of test cases or generation time within each class. This allows complex methods to achieve high coverage through diverse test cases. We limited automatic repair iterations to 5 in KTester. To mitigate the effect of randomness introduced by LLMs, we repeated each experiment five times and report the average results.

Table 1: Detailed Information of HITS Dataset

Project	Domain	Version	#MUT
Commons-CLI	Cmd-line Interface	1.7.0-SNAPSHOT	2
Commons-CSV	Data processing	1.10.0	6
Gson	Serilization	2.10.1	20
Commons-codec	Encoding	3a6873e	18
Commons-collections4	Utility	4.5.0-M1	14
JDom2	Text Processing(XML)	2.0.6	21
Datafaker	Data Generation	1.9.0	6
Event-ruler	Event Engine	1.4.0	15
windward	Micoservices	1.5.1-SNAPSHOT	2
batch-processing-gateway	Cloud Computing	1.1	6

To systematically evaluate the effectiveness of generated unit tests, we adopt eight widely used metrics for unit test quality, capturing correctness, sufficiency and efficiency [20, 32, 52, 58].

- **Compile Pass Rate (CPR)**: percentage of test classes that compile successfully.
- **Execution Pass Rate (EPR)**: percentage that run without runtime errors or assertion failures.
- **Line Coverage (LC)**: ratio of executable lines in the focal method covered by all generated tests.
- **Branch Coverage (BC)**: percentage of conditional branches covered.
- **Line Coverage of Passed Tests (LCP)**: line coverage computed only on tests that compile and execute successfully.
- **Branch Coverage of Passed Tests (BCP)**: branch coverage computed similarly on passing tests.
- **Average Time pre Task (AvT)**: average time required by a method to generate test class for a single task.
- **Average Test Cases Number (AvTC)**: average number of test cases contained in each generated test class.

Coverage is measured using Jacoco [8], an open-source tool for measuring code coverage in Java projects, ensuring objective and consistent evaluation. By combining correctness (CPR, EPR), coverage metrics (LC, BC, LCP, BCP) and efficiency metrics (AvT, AvTC), we provide a balanced and robust assessment of test quality, facilitating fair comparison between KTester and baseline approaches.

3.2.2 Results. Table 2 presents the effectiveness comparison across four methods. Overall, KTester achieves the best performance on all six evaluation metrics, demonstrating its ability to generate high-quality unit tests both in terms of correctness and coverage.
Correctness. KTester reaches a perfect Compile Pass Rate (CPR) of **100%**, indicating that all generated test classes are syntactically valid and type-correct. While UTGen also achieves 100% CPR, the other LLM-based baselines fall short, suggesting potential issues in code generation or dependency handling. Regarding the Execution Pass Rate (EPR), which measures runtime correctness, KTester attains 77.07%, slightly lower than UTGen. This is expected, as UTGen

Table 2: Comparison of Effectiveness across Baselines

	UTGen	ChatTester	ChatUniTest	HITS	KTester
CPR	**100**	55.18	98.97	97.82	**100**
EPR	**90.05**	50.17	65.26	71.38	76.41
LC	31.17	28.51	44.64	52.27	**63.94**
BC	28.72	24.94	38.06	45.93	**55.46**
LCP	30.69	22.26	33.47	43.85	**55.52**
BCP	28.11	19.63	28.51	38.81	**47.21**
AvT (s)	1068	354.83	200.96	625.69	**152.68**
AvTC	9.23	3.32	4.77	15.78	7.33

Table 3: Comparison of Effectiveness across KTester implemented with different models

	KTester-gpt	KTester-claude	KTester-deepseek
CPR	100	100	93.92
EPR	76.41	81.75	82.41
LC	63.94	66.46	71.22
BC	55.46	60.66	65.75
LCP	55.52	56.30	65.19
BCP	47.21	50.76	59.33

Test Class (generated by HITS)

```
public class Flat3Map_2_0_Test {
  @Test
  void testGet_MatchingKey_ReturnsValue() {
    Flat3Map<String, Integer> flat3Map=new Flat3Map<>();
    Field sizeField=Flat3Map.class.getDeclaredField("size");
    sizeField.setAccessible(true);
    sizeField.set(flat3Map, 1);
    ...
    Integer expectedValue = 100;
    Integer actualValue = flat3Map.get("key1");
    assertEquals(expectedValue, actualValue);
}}
```

Using reflection to access private fields

Test Class (generated by KTester)

```
class Flat3Map_get_Test {
    private Flat3Map<Integer, String> flat3Map;
    @BeforeEach
    void setupBeforeEach() {
        flat3Map = new Flat3Map<>();
        flat3Map.put(1, "One");
        flat3Map.put(2, "Two");
        flat3Map.put(3, "Three");
  } @Test
    void testGet_ExistingKey() {
        assertEquals("One", flat3Map.get(1), "...");
        assertEquals("Two", flat3Map.get(2), "...");
        assertEquals("Three", flat3Map.get(3), "...");
}}
```

Using related method to set up

Figure 10: Test Classes generated by HITS and KTester.

builds upon EvoSuite-generated tests, which provide a strong foundation for runtime correctness. Nevertheless, KTester still surpasses all other LLM-based methods in EPR, demonstrating its effectiveness in generating executable and correct test cases. This result indicates that incorporating project knowledge yields substantial benefits, as it more effectively guides LLMs toward generating executable test code.

Sufficiency. In terms of code coverage, KTester achieves the highest Line Coverage (LC) of **63.94%** and Branch Coverage (BC) of **55.46%**, significantly outperforming HITS (52.27% LC, 45.93% BC) while generating less test case (7.33 vs. 15.78). ChatTester yields the lowest coverage, highlighting its limited ability to explore program logic. UTGen is constrained by EvoSuite's inability to operate on certain project-specific packages (e.g., *com.apple.spark* in the batch-processing-gateway project) and by the path-explosion problem inherent to SBST techniques. As a result, its coverage exceeds only that of ChatTester. When restricting evaluation to only the tests that both compile and execute successfully, KTester maintains the lead with **55.52%** Line Coverage of Passed Tests (LCP) and **47.21%** Branch Coverage of Passed Tests (BCP), confirming the practical adequacy of its outputs.

These results provide evidence that integrating project-specific and test-domain knowledge into LLM-based generation pipelines is associated with improved test adequacy and correctness, as reflected by higher coverage and pass rates. In particular, KTester demonstrates stronger capability in exercising program logic and control flow, making it a more effective and practical solution for automated unit test generation.

Figure 10 shows the test class generated for the *get()* method with HITS and KTester. HITS used reflection to access and set private values, which increases the difficulty of understanding. In fact, since Flat3Map is a Map data structure, we can better modify private fields through the relevant methods *put ()* in this class, thus covering every conditional branch during testing.

Generalizability. To assess whether KTester maintains its performance when implemented with alternative LLMs, we implemented KTester using claude-3.5-haiku-20241022 [1] and deepseek-v3.1 [3], and the experimental results are reported in Table 3, which achieved even better results than gpt4o-mini, confirming its generalizability.

Finding 1: KTester outperforms all baselines across eight metrics. Compared to the strongest baseline (HITS), and while generating on average 8.45 fewer test cases, it improves execution pass rate by 5.03% and line coverage by 11.67%, demonstrating clear gains in both correctness and sufficiency.

3.3 RQ2: Ablation Study

3.3.1 Design. To assess the individual impact of key components in KTester on unit test generation, we conducted an ablation study. Four variants were created, each removing or modifying one component while keeping the rest of the pipeline intact. This design isolates each component's effect on correctness and coverage. The variants include:

- **KTester-UTE**: without the procedure of **u**sage **t**race **e**xtraction (section 2.1).
- **KTester-FMR**: without the step of **f**unctionally related **m**ethod **r**etrieval (section 2.1.2).
- **KTester-MVG**: using only conditional branch guidance instead of the whole **m**ulti-**v**iew guidance strategy (section 2.2.2).
- **KTester-DGT**: replace the processes of test cases design and test method transformation (section 2.2.3) with a single process that **d**irectly **g**enerates **t**est methods using a multi-view guidance strategy.

Table 4: Comparison of Effectiveness across Variants and original KTester.

	CPR	EPR	LC	BC	LCP	BCP
KTester-UTE	93.63	62.00	57.53	51.27	45.37	39.83
KTester-FMR	95.66	66.99	59.17	52.32	46.83	41.02
KTester-MVG	97.45	75.31	53.67	46.26	45.48	39.36
KTester-DGT	96.14	64.96	50.55	44.31	41.07	35.65
KTester	**100**	**76.41**	**63.94**	**55.46**	**55.52**	**47.21**

Using the same experimental setup and metrics as in RQ1, we evaluated all variants and compared their results to the original KTester to measure each component's contribution to test quality.

3.3.2 Results. Table 4 summarizes the performance of KTester variants. All variants exhibit performance drops across correctness and coverage metrics, confirming that each component contributes to KTester. Notably, KTester-UTE shows the largest correctness degradation, with CPR and EPR decreasing by 6.37% and 14.41%, respectively, highlighting the critical role of usage trace extraction in improving test correctness. KTester-DGT leads to the most substantial coverage losses, with LC and BC dropping by 13.39% and 11.15%, respectively, indicating that separating test case design from test method generation is crucial for test adequacy. The other variants—KTester-FMR and KTester-MVG have comparatively smaller impacts across both correctness and coverage metrics, but also result in performance reductions.

Finding 2: Usage trace extraction (KTester-UTE) is the primary contributor to test correctness, while modular test case transformation (KTester-DGT) contributes most to test adequacy. Other components also contribute positively with smaller and less distinguishable effects.

3.4 RQ3: User Study

3.4.1 Design. To investigate whether the tests generated by our approach exhibit superior readability and maintainability, we conducted a user study targeting professional developers. We first selected 10 tasks from the dataset where both KTester and the baseline methods successfully produced compilable test cases. Ensuring compilability allows participants to concentrate on qualitative properties of the test code—such as clarity, structure, and maintainability—without being influenced by technical correctness issues.

We recruited participants through a public invitation distributed across the computer science departments of six universities. From the volunteers, we selected 15 participants (5 Ph.D. students and 10 Master's students), all with 2–5 years of Java development experience. Participants received compensation to encourage careful and unbiased evaluation.

For each task, participants were presented with the focal method under test, its API documentation and relevant class context, then the related test classes generated by different methods. To avoid bias, the identities of the methods were fully anonymised, and the presentation order was randomised. Participants were instructed to respond to the following three questions, each designed to target a key quality dimension of the generated test code:

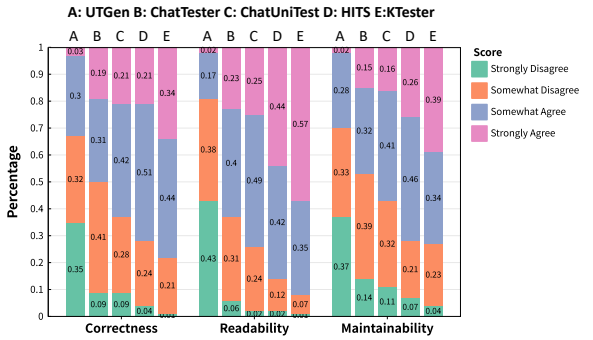

Figure 11: Score Distribution of Correctness, Readability and Maintainability

(1) To what extent do you agree that the unit test class is functionally correct? Specifically, does it adequately cover relevant input combinations, contain at least one meaningful assertion per test method, and reliably detect functional faults through deterministic and repeatable execution?
(2) To what extent do you agree that the unit test class is highly readable? For instance, are the method and variable names clear and descriptive? Does each test follow the Arrange–Act–Assert structure to clearly communicate intent? Are assertion failure messages informative and helpful for debugging?
(3) To what extent do you agree that the unit test class is maintainable? For example, does it minimise redundancy or duplicated code? Is the test class support the easy addition of new test cases? Can developers easily update affected tests when the source code changes?

Each question is rated on a four-point Likert scale—strongly agree, somewhat agree, somewhat disagree, and strongly disagree. This setup enables quantitative comparison of subjective test quality across approaches. Following prior work [45, 58], we adopt an even-numbered scale to avoid ambiguous "neutral" responses, and clearly separates positive from negative judgments.

3.4.2 Results. Figure 11 presents the distribution of developer ratings across the three evaluation dimensions. Across all three dimensions, our method (KTester, labeled as E) consistently outperforms the baselines in terms of perceived quality.

In the dimension of Correctness, KTester achieves the highest proportion of "Strong Agree" ratings (0.34), while keeping "strongly disagree" scores low (0.01). This trend indicates a strong alignment with expected test behavior as perceived by developers. In terms of Readability, KTester demonstrates the most favorable distribution, with over 90% of responses rated as "strongly Agree" (0.57) or "Somewhat Agree" (0.35), suggesting that participants found its tests clearer and more accessible compared to those from other methods. For Maintainability, KTester has the highest proportion of "Strong Agree" responses (0.39) and the lowest proportion of "strongly disagree" scores (0.04). This suggests that developers found the structure and logic of the tests generated by our approach easier to understand and adapt.

Interestingly, although UTGen uses an LLM to improve test readability, it still received more than 67% negative responses across all dimensions. This is mainly due to (1) function signatures and variable names often remaining unmodified, leaving the test code hard to understand, and (2) UTGen's EvoSuite component testing private methods via public callers, while LLM-based methods rely on reflection, making participants feel hard to identify impacted tests after source code changes and thus negatively affecting assessments of correctness and maintainability.

Finding 3: KTester outperforms all baselines in human evaluation across correctness, readability, and maintainability.

4 Related Work

In this section, we mainly introduce related work on LLM-based unit test generation.

4.1 LLM-based Unit Test Generation

Large language models (LLMs) have become a powerful tool for automated unit test generation, offering human-like test code that overcomes the readability and maintainability limitations of traditional methods [15, 18, 19, 34, 42]. Existing LLM-based approaches can be broadly categorized into two paradigms: fine-tuning and prompt-based methods.

Fine-tuning approaches frame test generation as a supervised sequence-to-sequence task, learning mappings from focal methods to complete test cases using curated corpora, *e.g.,* AthenaTest [51], TeCo [43], and subsequent variants [23, 26, 27], which explore different encoder–decoder architectures to capture the semantic relationships between code and tests. EXLONG [60] further fine-tunes CodeLlama for exception-oriented test generation. Unlike these approaches, our method injects project and testing knowledge through prompt design and generation workflows, enabling greater flexibility and compatibility with future LLMs.

Prompt-based methods, in contrast, exploit the zero-shot reasoning abilities of instruction-tuned models. Systems such as ChatTester [58], ChatUniTest [20], and ASTER [45] use carefully designed prompts and multi-step reasoning to guide models like GPT in test identification, input generation, and assertion construction. Extensions incorporate additional context or search strategies, including documentation and usage examples in TestPilot [50] and IDE-based search in TestSpark [49].

Recent work explores hybrid approaches that combine LLMs with complementary techniques. CODAMOSA [38] integrates LLM-generated test seeds with evolutionary algorithms to improve coverage-driven test generation, CoverUp [13] refines tests through coverage-guided dialogues, and SymPrompt [48] leverages symbolic execution to steer prompting.

Despite these advancements, most approaches lack project-specific context (*e.g.,* instantiation patterns, dependencies, and API usage) and overlook core testing principles such as boundary analysis, equivalence partitioning, and exception handling, yielding tests without execution error but insufficiently validate functionality.

4.2 Knowledge-enhanced LLMs for SE Tasks

Recent studies have explored enhancing LLM performance on SE tasks by incorporating task-relevant context. In APR, Xia et al. [55] show that simply providing the buggy function can outperform traditional tools, while ChatRepair [56] leveraged failing test names and assertions for interactive prompting. Other work enriches context with bug-localized code [46], relevant identifiers [37, 54], stack traces [33], and bug reports [29].

In parallel, retrieval-augmented generation (RAG) integrates external knowledge by retrieving relevant information from codebases or databases. For code completion, ReACC [41] retrieves semantically similar code snippets, and Wu et al. [53] propose a selective RAG framework to reduce redundancy. For code generation, recent work [14, 40] represents code repositories as graphs or structured knowledge to retrieve and add repository-level context to LLMs.

5 Threats to Validity

A key threat lies in the use of LLMs. To ensure fairness, all approaches are evaluated using the same model version. For baselines, we rely on official implementations or carefully reimplement them following published details, verifying outputs to align with reported results. Benchmark tasks and ground-truth tests are directly reused from prior work for consistency.

Another threat concerns generality and subjectivity. Although experiments focus on Java for comparability [52, 58], KTester is not inherently tied to Java. Extending to other frameworks mainly requires prompt and library updates, while supporting other languages only involves replacing the AST analysis tool (e.g., tree-sitter), resulting in low adaptation cost. Future work will explore such extensions. While correctness and coverage are objective metrics, they may not fully capture clarity and structure. we mitigate this via a user study on readability and maintainability with 14 professional developers, though results may still reflect participant bias and limited sampling. We believe this is sufficient, but a broader sampling could further strengthen validity.

6 Conclusions

We present KTester, a project-aware, testing-domain-knowledge-guided framework for LLM-based unit test generation. By extracting comprehensive project knowledge and decoupling test case design from test code generation, KTester effectively guides the LLM to produce semantically rich and maintainable tests. Our multi-perspective prompting strategy ensures diverse and thorough coverage of testing scenarios, while structured test templates enhance code clarity and reusability. Extensive evaluations on real-world open-source projects show that KTester consistently outperforms existing state-of-the-art methods in correctness, coverage, readability, and maintainability. Future work explores integrating dynamic analysis for enriching project knowledge and extending the approach to other languages and testing paradigms.

Acknowledgments

This work was supported by National Natural Science Foundation of China (Grant No. 62402113), the General Program of the Natural Science Foundation of Guangdong Province, China (Grant

No. 2025A1515011631), Social Science Planning Project (Grant No. SZ2025A002), and GMCC-SYSU Joint Lab for Smart Applications.

References

[1] 2024. claude-3-5-haiku-20241022 model overview. https://docs.claude.com/en/docs/about-claude/models/overview#legacy-models

[2] 2025. codellama-7b. https://ollama.com/library/codellama:7b

[3] 2025. deepseek-v3.1 Release. https://api-docs.deepseek.com/news/news250821

[4] 2025. http://javaparser.org/. (2025).

[5] 2025. https://github.com/ZJU-ACES-ISE/chatunitest-maven-plugin. (2025).

[6] 2025. https://platform.openai.com. (2025).

[7] 2025. https://spoon.gforge.inria.fr/about.html. (2025).

[8] 2025. https://www.jacoco.org/jacoco/. (2025).

[9] 2025. KTester. https://github.com/SYSUSELab/KTester

[10] 2025. UTGen Replication Package. https://github.com/amirdeljouyi/UTGen

[11] Frances E. Allen. 1970. Control flow analysis. *SIGPLAN Not.* 5, 7 (July 1970), 1–19. doi:10.1145/390013.808479

[12] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 263–272.

[13] Juan Altmayer Pizzorno and Emery D. Berger. 2025. CoverUp: Effective High Coverage Test Generation for Python. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE128 (June 2025), 23 pages. doi:10.1145/3729398

[14] Mihir Athale and Vishal Vaddina. 2025. Knowledge Graph Based Repository-Level Code Generation. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*. IEEE, 169–176. doi:10.1109/llm4code66737.2025.00026

[15] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 50:1–50:39. doi:10.1145/3182657

[16] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525. doi:10.1109/TSE.2014.2372785

[17] Arianna Blasi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. 2023. 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 (ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 19, 11 pages. doi:10.1145/3551349.3556961

[18] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[19] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 380–394. doi:10.1109/SP.2012.31

[20] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 572–576. doi:10.1145/3663529.3663801

[21] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: Dynamic symbolic execution for invariant inference. In *Proceedings of the 30th international conference on Software engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 281–290. doi:10.1145/1368088.1368127

[22] Ermira Daka and Gordon Fraser. 2014. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering (ISSRE '14)*. IEEE, IEEE Computer Society, USA, 201–211. doi:10.1109/ISSRE.2014.11

[23] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. 2024. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology* 171 (2024), 107468. doi:10.1016/J.INFSOF.2024.107468

[24] Pedro Delgado-Pérez, Aurora Ramírez, Kevin J. Valle-Gómez, Inmaculada Medina-Bulo, and José Raúl Romero. 2023. InterEvo-TR: Interactive Evolutionary Test Generation With Readability Assessment. *IEEE Transactions on Software Engineering* 49, 4 (2023), 2580–2596. doi:10.1109/TSE.2022.3227418

[25] Amirhossein Deljouyi, Roham Koohestani, Maliheh Izadi, and Andy Zaidman. 2025. Leveraging Large Language Models for Enhancing the Understandability of Generated Unit Tests. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. IEEE, 1449–1461. doi:10.1109/ICSE55347.2025.00032

[26] 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*. ACM, 423–435. doi:10.1145/3597926.3598067

[27] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the 46th IEEE/ACM international conference on software engineering*. ACM, 1–13. doi:10.1145/3597503.3623343

[28] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45. doi:10.1016/J.SCICO.2007.01.015

[29] Sarah Fakhoury, Saikat Chakraborty, Madanlal Musuvathi, and Shuvendu K. Lahiri. 2024. NL2Fix: Generating Functionally Correct Code Edits from Bug Descriptions. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings* (Lisbon, Portugal) *(ICSE-Companion '24)*. Association for Computing Machinery, New York, NY, USA, 410–411. doi:10.1145/3639478.3643526

[30] 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* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. doi:10.1145/2025113.2025179

[31] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.* 24, 2, Article 8 (Dec. 2014), 42 pages. doi:10.1145/2685612

[32] Siqi Gu, Quanjun Zhang, Kecheng Li, Chunrong Fang, Fangyuan Tian, Liuchuan Zhu, Jianyi Zhou, and Zhenyu Chen. 2025. TestART: Improving LLM-based Unit Testing via Co-evolution of Automated Generation and Repair Iteration. arXiv:2408.03095 [cs.SE] doi:10.48550/arXiv.2408.03095

[33] Mirazul Haque, Petr Babkin, Farima Farmahinifarahani, and Manuela Veloso. 2025. Towards Effectively Leveraging Execution Traces for Program Repair with Code LLMs. In *Proceedings of the 4th International Workshop on Knowledge-Augmented Methods for Natural Language Processing*, Weijia Shi, Wenhao Yu, Akari Asai, Meng Jiang, Greg Durrett, Hannaneh Hajishirzi, and Luke Zettlemoyer (Eds.). Association for Computational Linguistics, Albuquerque, New Mexico, USA, 160–179. doi:10.18653/v1/2025.knowledgenlp-1.17

[34] Mark Harman, Yue Jia, and Yuanyuan Zhang. 2015. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 1–12.

[35] Mark Harman and Phil McMinn. 2009. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering* 36, 2 (2009), 226–247. doi:10.1109/TSE.2009.71

[36] Paul Jaccard. 1902. Lois de distribution florale dans la zone alpine. *Bulletin de la Société vaudoise des sciences naturelles* 38 (01 1902), 69–130. doi:10.5169/seals-266762

[37] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1430–1442. doi:10.1109/ICSE48619.2023.00125

[38] 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 *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 919–931. doi:10.1109/ICSE48619.2023.00085

[39] Hareton KN Leung and Lee White. 1989. Insights into regression testing (software testing). In *Proceedings. Conference on Software Maintenance-1989*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 60–69. doi:10.1109/ICSM.1989.65194

[40] Jia Li, Xianjie Shi, Kechi Zhang, Lei Li, Ge Li, Zhengwei Tao, Jia Li, Fang Liu, Chongyang Tao, and Zhi Jin. 2025. CodeRAG: Supportive Code Retrieval on Bigraph for Real-World Code Generation. *CoRR* abs/2504.10046 (2025). arXiv:2504.10046 doi:10.48550/ARXIV.2504.10046

[41] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*. Association for Computational Linguistics, 6227–6240. doi:10.18653/V1/2022.ACL-LONG.431

[42] 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 Computer Society, 153–163. doi:10.1109/ICSTW.2011.100

[43] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2111–2123. doi:10.1109/ICSE48619.2023.00178

[44] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 75–84.

[45] Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. 2025. ASTER: Natural and Multi-Language Unit Test Generation with LLMs. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 413–424. doi:10.1109/ICSE-SEIP66354.2025.00042

[46] Julian Aron Prenner and Romain Robbes. 2024. Out of Context: How important is Local Context in Neural Program Repair?. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3597503.3639086

[47] Per Runeson. 2006. A survey of unit testing practices. *IEEE software* 23, 4 (2006), 22–29. doi:10.1109/MS.2006.91

[48] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM. *Proc. ACM Softw. Eng.* 1, FSE, Article 43 (July 2024), 21 pages. doi:10.1145/3643769

[49] Arkadii Sapozhnikov, Mitchell Olsthoorn, Annibale Panichella, Vladimir Kovalenko, and Pouria Derakhshanfar. 2024. Testspark: Intellij idea's ultimate test generation companion. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. ACM, 30–34. doi:10.1145/3639478.3640024

[50] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* 50, 1 (2023), 85–105. doi:10.1109/TSE.2023.3334955

[51] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020). doi:10.48550/arXiv.2009.05617

[52] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1258–1268. doi:10.1145/3691620.3695501

[53] Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. REPOFORMER: selective retrieval for repository-level code completion. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 2183, 21 pages.

[54] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2024. The Plastic Surgery Hypothesis in the Era of Large Language Models. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering* (Echternach, Luxembourg) *(ASE '23)*. IEEE Press, 522–534. doi:10.1109/ASE56229.2023.00047

[55] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1482–1494. doi:10.1109/ICSE48619.2023.00129

[56] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 819–831. doi:10.1145/3650212.3680323

[57] Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. 2013. Characteristic studies of loop problems for structural test generation via symbolic execution. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 246–256.

[58] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2024. Evaluating and Improving ChatGPT for Unit Test Generation. *Proc. ACM Softw. Eng.* 1, FSE (2024), 24 pages. doi:10.1145/3660783

[59] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. The fuzzing book. https://www.fuzzingbook.org

[60] Jiyang Zhang, Yu Liu, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2025. exLong: Generating exceptional behavior tests with large language models. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 1462–1474. doi:10.1109/ICSE55347.2025.00176