Unit Test Update through LLM-Driven Context Collection and Error-Type-Aware Refinement

Yuanhe Zhang, Zhiquan Yang, Shengyi Pan, Zhongxin Liu*
The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China
{yuanhezhang, zhiquanyang, shengyi.pan, liu_zx}@zju.edu.cn

***Abstract*—Unit testing is critical for ensuring software quality and software system stability. The current practice of manually maintaining unit tests suffers from low efficiency and the risk of delayed or overlooked fixes. Therefore, an automated approach is required to instantly update unit tests, with the capability to both repair and enhance unit tests. However, existing automated test maintenance methods primarily focus on repairing broken tests, neglecting the scenario of enhancing existing tests to verify new functionality. Meanwhile, due to their reliance on rule-based context collection and the lack of verification mechanisms, existing approaches struggle to handle complex code changes and often produce test cases with low correctness.**

To address these challenges, we propose TESTUPDATER, a novel Large Language Model (LLM) based approach that enables automated just-in-time test updates in response to production code changes. By emulating the reasoning process of developers, TESTUPDATER first leverages the LLM to analyze code changes and identify relevant context, which it then extracts and filters. This LLM-driven context collector can flexibly gather accurate and sufficient context, enabling better handling of complex code changes. Then, through carefully designed prompts, TESTUPDATER guides the LLM step by step to handle various types of code changes and introduce new dependencies, enabling both the repair of broken tests and the enhancement of tests. Finally, emulating the debugging process, we introduce an error-type-aware iterative refinement mechanism that executes the LLM-updated tests and repairs failures, which significantly improves the overall correctness of test updates.

Since existing test repair datasets lack scenarios of test enhancement, we further construct a new benchmark, UPDATES4J, with 195 real-world samples from 7 projects, enabling execution-based evaluation of test updates. Experimental results show that TESTUPDATER achieves a compilation pass rate of 94.4% and a test pass rate of 86.7%, outperforming the state-of-the-art method SYNTER by 15.9% and 20.0%, respectively. Furthermore, TESTUPDATER exhibits 12.9% higher branch coverage and 15.2% greater line coverage than SYNTER.

I. INTRODUCTION

Unit testing is a fundamental practice in modern software development [1]. It verifies the functionality of individual components within a codebase, helping developers detect regressions early in the development cycle and maintain system stability [2], [3]. However, as software evolves, unit tests that are not updated accordingly can become outdated and incompatible with changes in production code, potentially resulting in compilation or runtime failures. Moreover, outdated unit tests fail to cover the newly introduced functionalities in the updated codebase, which can lead to undetected bugs and reduced test coverage [4]. Therefore, keeping unit tests up to date is essential for maintaining the quality and effectiveness of test suites [5], [6].

Manually updating unit tests to reflect changes in production code, however, is a time-consuming and error-prone process [7], [8]. This challenge is pronounced considering large and complex projects, where frequent updates require continuous maintenance of tests. Therefore, automatically updating unit tests is crucial to reduce manual efforts and improve the efficiency and accuracy of test maintenance.

Early research efforts propose rule-based approaches to automatically repair outdated tests [9]–[11]. However, these approaches depend on predefined rules or patterns, which fail to cover all possible test repair scenarios. With the advancement of deep learning, some researchers fine-tune pre-trained models (PTMs) to perform test repair and update, such as CEPROT [12] and TARGET [13]. Recently, researchers have explored using large language models (LLMs) to automatically repair broken unit tests. Liu *et al.* [14] introduce the first LLM-based approach (namely SYNTER) that uses GPT-4 [15] to repair obsolete test cases. SYNTER enhances LLM's ability by collecting and reranking test-repair-oriented contexts.

However, the existing approaches face the following three challenges:

Challenge 1: **Test enhancement for new functionality.** The real-world software evolution requires both test repair and test enhancement. Automatically enhancing test cases to verify new functionality is challenging because it requires correctly determining how the new functionality should be verified and collecting the relevant context necessary to implement the corresponding test logic. Such capabilities go beyond simple syntactic or signature changes and demand a deep semantic understanding of the codebase. Tools like TARGET and SYNTER, which focus on repairing broken tests due to syntactic or signature changes [13], [14], fall short in scenarios requiring test enhancement to reflect new behavioral logic.

Challenge 2: **Context collecting for complex code changes.** A key challenge in test updates is to accurately capture sufficient context in complex code changes. The context may include method dependencies, class definitions, and related variables, all of which help the LLM understand code changes and produce more effective results. Existing methods rely on rule-based context extraction strategies, which fail to generalize beyond simple code edits and struggle to handle complex code changes. In our evaluation, this challenge was

* Corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00206

evident in SYNTER, where at least 16.9% of the updated test cases failed to compile due to missing or inadequate context.

Challenge 3: **Correctness assurance after the LLM updates the test.** Another major challenge in test update lies in how to correct erroneous outputs after the test has been updated by the LLM. Although providing sufficient context can often enable the LLM to perform reasonable updates, it may still make unnecessary modifications or introduce undeclared dependencies. As observed by Yuan *et al.* [16], although most LLM-generated tests are syntactically valid, a large portion still suffer from compilation or execution errors. Existing approaches typically lack mechanisms for validating and refining the LLM's output, which requires manual debugging and limiting the effectiveness.

To address ***Challenge 1***, we propose TESTUPDATER, an LLM-based approach that can not only repair but also enhance tests (e.g., verifying new functionality) instantly in response to production code changes, a process we refer to as *just-in-time test update*. We design carefully crafted prompts to guide the LLM step by step, enabling it to reason about behavioral changes and introduce new dependencies when necessary. For ***Challenge 2***, in contrast to prior rule-based context collectors, we ask the LLM to mimic the reasoning process of developers to determine which classes or methods should be referenced when updating tests, and retrieve them through a language server [17]. This design enables flexible and accurate dependency resolution, allowing TESTUPDATER to handle a wide variety of code change scenarios. Additionally, we introduce another important but previously overlooked context source: the predefined variables in the test class, which LLMs can use directly. For ***Challenge 3***, we innovatively introduce an error-type-aware iterative refinement mechanism as a post-processing module to further enhance the tests updated by the LLM. Inspired by developers' debugging practices, after executing the LLM-updated test, TESTUPDATER detects and classifies errors (e.g., missing symbol, type mismatch, assertion failure) and maps each type to a pre-defined refinement strategy. Unlike prior single-shot generation approaches, our iterative repair strategy effectively guarantees the reliability of LLM-updated tests.

Furthermore, existing test repair datasets often lack scenarios involving test enhancement. To evaluate the effectiveness of our approach, we construct a new dataset UPDATES4J by collecting samples of method-level production-test co-evolution from the open-source Java projects. UPDATES4J contains 195 samples from 7 different projects, covering real-world scenarios of software evolution and supporting execution-based evaluation. Based on UPDATES4J, we assess the performance of TESTUPDATER in terms of *compilation pass rate*, *test pass rate*, and *test coverage* with three different LLMs: `GPT-4.1`, `Llama-3.3-70B-Instruct`, and `DeepSeek-V3`. The best result achieved by TESTUPDATER yields a 94.4% compilation pass rate and an 86.7% test pass rate, representing improvements of 15.9% and 20.0% over the state-of-the-art (SOTA) method SYNTER [14], respectively. In terms of test coverage, TESTUPDATER achieves an overall branch coverage of 46.0% and a line coverage of 69.1%, surpassing SYNTER by 12.9% and 15.2%, respectively. Furthermore, we validate the effectiveness of key components in TESTUPDATER through an ablation study.

In summary, we make the following contributions in this paper:

- **TESTUPDATER: A novel LLM-based test update approach that goes beyond test repair.** TESTUPDATER can not only repair but also enhance the outdated unit tests. By dynamically combining the reasoning ability of LLMs with language servers and compiler feedback, TESTUPDATER facilitates the algorithmic emulation of developers' behavior in test update, and further formalizes this process into three essential steps: context collection, test generation, and iterative refinement. We open-source our replication package for follow-up works [1].
- **UPDATES4J: A test update evaluation dataset.** We construct a new dataset UPDATES4J to fill the gap in existing benchmarks that focus solely on test repair, lacking coverage of test enhancement scenarios. UPDATES4J consists of 195 samples from 7 open-source Java projects, capturing real-world test update scenarios and enabling execution-based evaluations of automated test update approaches.
- **An empirical evaluation of TESTUPDATER on UPDATES4J.** We conduct extensive experiments to evaluate the effectiveness of our approach. The experimental results demonstrate that TESTUPDATER can effectively update outdated unit test cases and outperforms the LLM-based test repair method SYNTER [14], and the fine-tune methods CEPROT [12] and TARGET [13].

II. MOTIVATION

Fig. 1 presents a production-test co-update example. The method `deleteTopicRequest()` invoked in the focal method `deleteTopicRequests()` is updated to include a second parameter, `userName`. Consequently, the focal method is modified to invoke `getUserName()`, which uses `mailService.getUserName()` for the variable `userName`. To accommodate the update of production code, corresponding adjustments are made in the test method to 1) update the mock of `deleteTopicRequest()` to include the additional `userName` parameter. 2) mock the additional parameter with `mailService.getUserName()`.

The test updated by SYNTER incorrectly mocks the username with `commonUtilsService.getUserName()`, which results in compilation failures (highlighted in red). The reason is that through rule-based algorithms, SYNTER only collects three diff texts in the focal class, fails to cover the necessary context of `mailService.getUserName()` within the `getUserName()` method. Therefore, without knowing mock `mailService.getUserName()` for username, the LLM uses `commonUtilsService.getUserName()` just as the other mock behaviors in the test.

[1]https://github.com/ZJU-CTAG/TestUpdater

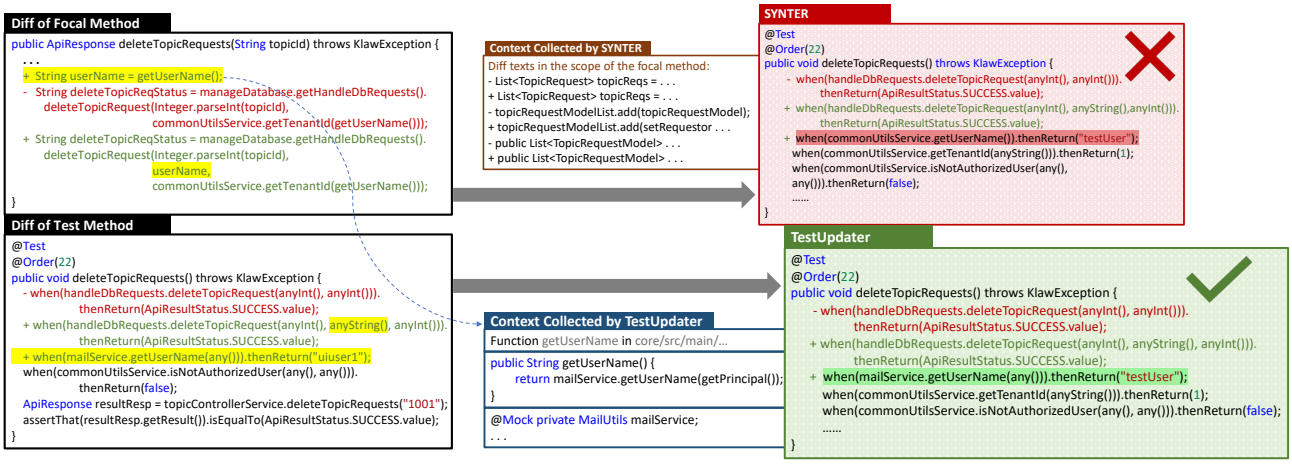

Fig. 1: A Motivation Example For Context Collection from commit *0f5599* in Project *Aiven-Open/klaw*

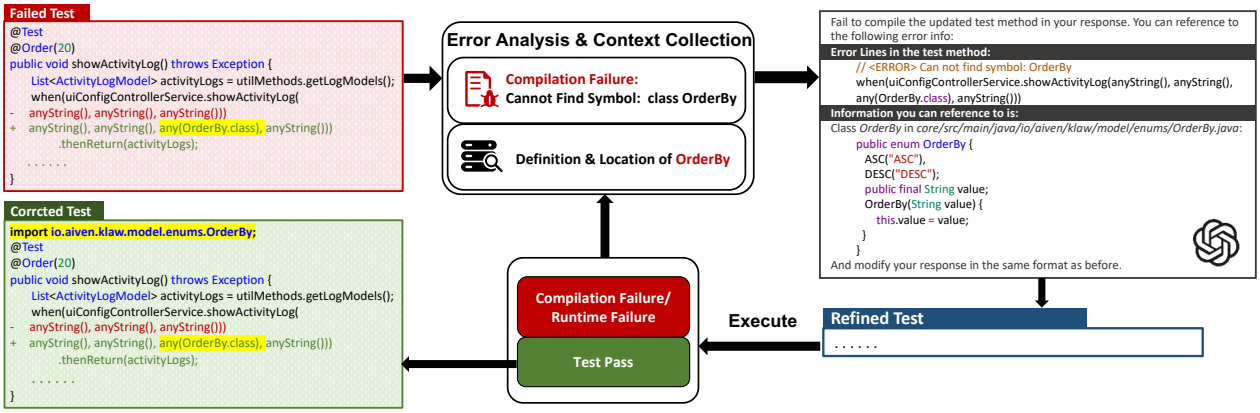

Fig. 2: A Motivation Example for Error-Type-Aware Iterative Refinement from commit *32e27ee* in Project *Aiven-Open/klaw*

To accurately locate the required context, we observe that developers often start by examining the definitions of relevant functions and classes when updating tests (`getUsername()` in this case). However, rule-based algorithms are unable to replicate the reasoning process used by experienced developers because they lack the flexibility to understand complex relationships and contextual nuances that go beyond predefined patterns. Given that LLMs possess human-like capabilities in code understanding and reasoning, we propose leveraging LLMs to identify the required context. To this end, we employ an LLM-driven context collector to accurately locate the critical context—the definition of the `getUsername()` function. Combined with the mock instance `mailService` defined in the test class, TESTUPDATER collects sufficient context and successfully performs the test update.

Another motivation is shown in Fig. 2. The red box in the figure highlights a failed test case where the LLM used `any(OrderBy.Class)` to mock a new parameter, but the compilation failed due to missing relevant dependencies. This issue arises because the LLM did not obtain the location for `OrderBy` or failed to correctly import the dependency based on location information. A simple idea to address this is to invoke the LLM again with the specific information about the `OrderBy` class. The challenge lies in determining which information needs to be extracted based on the type of error. To this end, we introduce an error-type aware iterative repair mechanism that extracts context and assists the LLM in fixing the test, as shown in the figure.

III. APPROACH

In this section, we first introduce the overall framework of TESTUPDATER. Then, we describe each key step in detail, including *context collection*, *updated test generation*, and *iterative refinement*.

A. Overview

To make it clear, in the rest of this paper, we use T_o for the original test method, T_u for the updated test method, F_o and F_u for the original and updated focal method, respectively. We define our approach as follows.

$$\text{TESTUPDATER}(F_o, F_u, T_o, Ctx, Err) \rightarrow T_u$$

where Ctx represents the collected context and Err represents the error feedback from execution. The goal of TESTUPDATER is to instantly update the associated test method according to the changes between F_o and F_u, and generate T_u that successfully compiles and passes against F_u. The overall framework of TESTUPDATER is illustrated in Fig. 4. The entire process can be generally divided into three stages:

1) **Context Collection**: TESTUPDATER initiates the update process through the LLM-driven context collector, which gathers the following contexts: 1. Update Related Components: methods and classes that may be necessary for the test update, including their definitions and locations. 2. Test Class Fields: all variables declared at the class level within the test class.
2) **Updated Test Generation**: Based on the collected context, TESTUPDATER constructs a structured prompt and invokes the LLM to automatically generate the updated test cases and introduce the required dependencies.
3) **Iterative Refinement**: TESTUPDATER compiles and runs the LLM-updated test cases. If compilation or test failures occur, TESTUPDATER performs targeted context gathering according to the error types and invokes the LLM to perform refinements. The iterative validate-repair loop continues until the test cases compile and pass, or a preset iteration limit is reached, with fallback to minimal modifications to ensure correctness.

B. Context Collection

Context is crucial for enabling LLMs to generate correct and relevant test updates while reducing hallucinations. Unlike prior methods that statically extract diff hunks or local signatures, we leverage the LLM to identify dependent methods and classes by reasoning over code changes, and then retrieve their definitions and locations by using a language server. Additionally, we introduce an additional type of context that has been overlooked by prior work - the predefined variables in the test class. Overall, we categorize these contexts into two types: *update related components* and *test class fields*.

Update Related Components. In the actual process of test update, developers usually need to consult the definitions and locations of related methods and classes. By analyzing their definitions, developers can better understand their functionality, thereby reasonably mocking new objects, adjusting parameter configurations, or adding test logic. By identifying the locations, developers can determine the necessary dependencies to import, which ensures the correct compilation and execution of the updated test cases.

Therefore, we design the Update Related Components context, which contains definitions and locations of relevant methods or classes. These methods and classes are typically referenced directly or indirectly in the updated test cases. The whole process is illustrated in Fig. 3, where an LLM mimics the developer's reasoning to determine the necessary information and further utilizes tools (e.g., a language server, static analysis tools) to extract the required data.

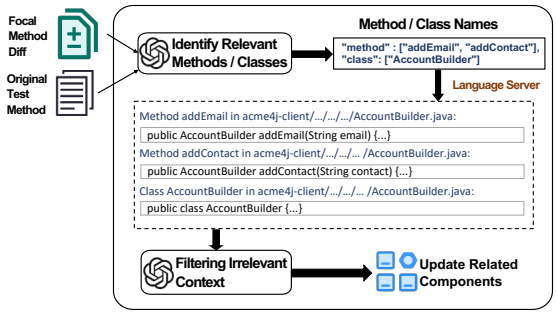

Fig. 3: The process for collecting Update Related Components.

The first step is to identify the names of the methods or classes that may be involved in the update. This process relies on a deep understanding of code semantics [18], a level of comprehension that rule-based methods employed in existing works often fail to achieve. Considering the strong reasoning capability of LLMs and their astonishing performance in multiple code understanding tasks [19] and automated software engineering tasks [20], TESTUPDATER leverages the LLM to analyze changes in the focal method and automatically identify the needed methods and classes. Specifically, we construct the prompt (c.f. our replication package for the detailed prompt) using chain-of-thought techniques [21] with the unified diff of F_o and F_u, and T_o as inputs. To ease the subsequent processing, we instruct the LLM to return the identification results in JSON format.

After identifying the methods and class names, the next step is to obtain their definitions and locations. TESTUPDATER leverages a Language Server [17] to locate the definitions of relevant methods and classes, and parse the code into Abstract Syntax Trees (ASTs) [22] to extract the corresponding code snippets. Language Servers are often integrated into Integrated Development Environments (IDEs) to assist programmers in efficient development, which analyzes the entire codebase to provide advanced language features such as code completion, syntax checking, go-to-definition, and reference finding.

The raw definitions of methods and classes extracted from the codebase cannot be directly provided to the LLM, as they often include extraneous information, such as unrelated methods or variables. This irrelevant content introduces additional overhead and may mislead the LLM during the test update process. To address this, TESTUPDATER directs the LLM to filter out unnecessary context from the raw definitions. Meanwhile, TESTUPDATER explicitly collects and preserves the source file paths of these methods and classes, enabling the LLM to accurately generate the required import statements.

Test Class Fields. In practical software development, developers frequently reuse class-level variables defined in test classes when updating unit test methods [23], which promotes consistency and reduces code redundancy. These variables usually contain instances of the classes under test, mocked dependencies, and configuration constants. Motivated by this observation, TESTUPDATER also collects Test Class Fields—all class-level variables defined within the test class, as

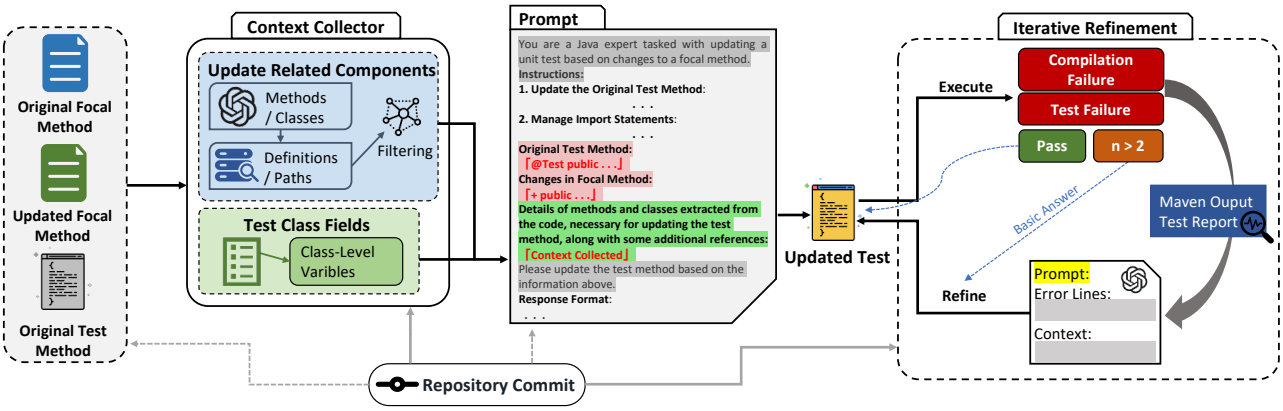

Fig. 4: Overall Framework of TESTUPDATER

a supplementary. Test Class Fields can prevent the LLM from unnecessarily mocking already defined variables, significantly reducing errors introduced by additional test logic.

C. Updated Test Generation

After collecting the contexts, the next step is to invoke the LLM to update the test cases. This phase consists of two key components: *prompt construction* and *post-processing*.

Prompt Construction. To enable the LLM to effectively perform both test repair and enhancement, we design a structured prompt template using the Markdown format. The prompt includes the following three inputs: (1) the unified diff of the focal method, (2) the original test method, (3) the collected context. As shown in Fig. 4, the prompt first sets the LLM's role as a Java expert. Then, through step-by-step instructions, it guides the model to accomplish two tasks: *update the original test method* and *introduce required dependencies*.

Regarding the *update original test method* part, the prompt provides five clear and actionable instructions to help the LLM repair and enhance the test method based on the changes in the focal method:

1) Update the test method using the provided context to align with changes in the focal method.
2) Ensure the updated test correctly validates the new logic.
3) If the focal method introduces new functionality, generate new test logic accordingly.
4) For any new parameters, mock the required objects or use default values.
5) If the core functionality of the focal method remains unchanged, only repair the original test.

Updating test code is often accompanied by the addition of new dependencies. In the *introduce required dependencies* section, the prompt instructs the LLM to add `import` statements only for newly introduced dependencies, avoiding duplicated imports. As for the response format, we instruct the LLM to respond only with Java code, beginning with new import statements and following with the updated test method, as illustrated in Listing 1.

Post-Processing. After receiving outputs from the LLM, TESTUPDATER performs a post-processing step. It parses the generated Java code to extract two key components: the updated test method and the `import` statements. These two components are then integrated into the test class during the subsequent iterative refinement phase, ensuring the completeness and executability of the generated test code.

```
// New import statements
import com.example.NewDependency;
import static org.mockito.Mockito.when;

// Updated test methods
@Test
public void testFocalMethod() {
    // Updated test logic here
}
```

Listing 1: An example of the response format

D. Iterative Refinement

By emulating developers' manual debugging processes, we design an error-type-aware iterative refinement module to further enhance the correctness of the updated test. Guided by the findings from ChatTester [16] that symbol resolution and type error dominate compilation failures and most execution failures (85.5%) are assertion errors, TESTUPDATER classifies errors into three major categories, i.e., missing symbol, type mismatch, and assertion error. For each category, we design a targeted refinement strategy and extract the precise context required, rather than merely passing raw error messages back to the LLM. The details are as follows.

Error Analysis and Context Collection. To ensure successful execution of the generated test case, TESTUPDATER first replaces the T_o with T_u and inserts the new `import` statements generated by LLM into the test file. Then, TESTUPDATER executes T_u and performs error analysis based on the type of failure encountered:

Compilation Failure: *1) Cannot find symbol.* This is the most common error encountered in the updated test code,

which typically arises from missing or wrong `import` statements or references to undefined classes, methods, or variables within the test method. For this failure case, TESTUPDATER extracts the name and the corresponding line number of the missing symbol from the compiler's error message. It then queries the language server to locate the symbol's definition or the symbol that invokes or accesses it, such as `symbolName.symbolName`. Once the missing symbol's definition is found, TESTUPDATER constructs a prompt including the error location and definition to assist the LLM in refining the test case, as shown in Fig. 2.

2) Argument type mismatch (cannot be applied to given types). This error occurs when the argument types or number of arguments passed to a method do not match its definition. In such cases, TESTUPDATER queries the language server to retrieve the method's definition, along with the error line, and uses these contexts to construct the prompt for repair.

3) Other compilation errors. The above two error types cover the majority of compilation failures caused by LLM-generated test updates. For other errors, either no additional context is needed (e.g., incompatible types), or the cases are too rare to justify custom handling. TESTUPDATER simply extracts the error messages and locations and then passes them to the LLM.

Test Failure: Test failure means that T_u fails to pass, which is typically caused by assertion errors. Similar to compilation failure, TESTUPDATER locates the test report and extracts context from it, including the expected and actual values of assertions, as well as the line number of the failed assertions. Then, TESTUPDATER extracts the corresponding assertions from the test code. Together, the expected and actual values and the assertions are used to construct a prompt (which shares a similar design to the one shown in Fig. 2) to guide the LLM in repairing T_u.

The Validate-Repair Loop. The entire process of validation and refinement is performed iteratively, with each round building upon the feedback from the previous failures to incrementally correct T_u. This validate-repair loop continues until T_u passes or the iteration limit is reached. Our initial experiments show that most errors can be corrected within two rounds of repair. For the remaining cases, however, the LLM tends to fall into erroneous loops, making it unlikely to produce correct results even after additional attempts. Thus, we limits the LLM to a maximum of two repair attempts.

While this loop can successfully correct the majority of common issues, it may still fail in some scenarios. For example, when the focal method introduces new parameters and the associated testing logic is intricate, the LLM may struggle to construct correct test logic. To address such cases, TESTUPDATER includes a fallback mechanism. If a correct pass is not achieved within the maximum number of repair iterations, the LLM is instructed to generate a *basic answer*, which abandons the addition of new testing logic and instead applies a minimal modification strategy, such as assigning default values to the new parameters. This ensures that the test method can at least compile and run successfully.

In summary, by integrating LLMs with language servers and compiler feedback, TESTUPDATER not only simulates the developer's manual update process but also advances test maintenance from rule-based, static repair toward a dynamic, reasoning-driven update paradigm. The LLM-driven context collector enables semantic dependency resolution beyond simple syntactic diffs, while the error-type-aware refinement guides updates through targeted search rather than ad-hoc retries. Together, these components establish a generalizable framework that enhances both the correctness and adaptability of automated test updates.

IV. DATASET: UPDATES4J

Existing datasets for test repair, such as TARBENCH [13], primarily focus on repairing failed test cases, while overlooking test cases that continue to pass after code changes but require additional test logic. To address this limitation, we construct a new dataset, named UPDATES4J. UPDATES4J contains both broken test repair and unbroken test enhancement scenarios by collecting real-world production-test co-evolution samples. Additionally, UPDATES4J incorporates the necessary runtime environment configurations to facilitate execution-based evaluation. The dataset construction process involves two main stages: *raw data collection* and *data validation*.

A. Raw Data Collection

We follow two steps to collect the initial data:

Project Collection. We first utilize the GitHub API to collect open-source Java projects. To facilitate compilation and the generation of test coverage reports, we only select projects that can be built with Maven and support JaCoCo. To ensure diversity and quality, we collect the following two categories of projects: (1) 966 projects with 50 to 1000 stars on GitHub. These projects vary in structure and maturity, reflecting common development scenarios. (2) 64 projects from the top 500 most-starred Java projects on GitHub. These projects are typically well-maintained, actively developed, and follow standardized practices.

Sample Collection. First, we collect commits that involve the modifications of both production and test code. Then, we employ the tree-sitter [24] tool to parse the ASTs to find the modified production and test methods. If a modified test method invokes a modified production method, they are considered as a co-evolution sample. According to Sun *et al.* [25], such modifications occurring within the same commit are most likely to represent the true production-test co-evolution. To control the evaluation costs, we select 10 projects from each of the two categories that contain a sufficient number of qualifying samples.

In total, we collect 4,286 raw samples from 20 projects. Each sample includes both the original and updated pairs of production and test methods.

B. Data Validation

We conduct two steps to ensure the quality of the data:

TABLE I: Details of Projects in UPDATES4J

Name	Samples	Stars
Aiven-Open/klaw	50	161
alibaba/nacos	30	31k
apache/rocketmq	20	21.7k
apache/shenyu	48	8.6k
OpenAPITools/openapi-generator	5	23.3k
prebid/prebid-server-java	24	76
shred/acme4j	18	541

Test Execution To ensure the correctness of each test update, the original and updated tests must pass on the original and updated production code, respectively. Consequently, we execute the original and updated test methods before and after the code commit. We extract the required Java version from the *pom.xml* file and determine the Maven version by consulting the *README* or Maven wrapper configuration files. If these versions are not specified, we automatically run the test methods across multiple versions. A sample is considered valid only when both test methods pass successfully. The runnable Java and Maven configurations are recorded in UPDATES4J.

Manual Inspection We further manually check the samples to eliminate updates in which the production and test code are not semantically related. To control the manual inspection cost, we select 7 projects that have a moderate size and fast compilation speed from the 20 projects. Additionally, since a single commit may contain multiple semantically redundant modifications [26], we manually removed duplicate samples (i.e., same changes related to one changed focal method) to reduce data redundancy.

After the automated and manual filtering, we collected 195 method-level co-evolution samples of production and test code from seven selected projects. Detailed information about the evaluation dataset UPDATES4J is shown in TABLE I. The dataset includes 165 broken test cases and 30 enhanced unbroken test cases, which reflect the realistic proportions in practical development scenarios. And there are 92 method signature changes and 103 internal logic changes in UPDATES4J.

V. EXPERIMENTAL SETUP

In this section, we first introduce the baselines we used. Then we describe the evaluation metrics in our experiments. Finally, we present the experimental settings of our evaluation.

A. Baselines

We select the following methods as our baselines:

CEPROT. CEPROT [12] is a PLM-based methods that fine-tunes CodeT5 [27] to automatedly update test cases. We configure CEPROT using the settings that achieved the best results in its original experiments.

TARGET. TARGET [13] is also a PLM-based method that fine-tunes code language models to automate test repair. We replicate TARGET with the best settings in its experiments, which employs the CodeT5+ [28] model.

SYNTER. SYNTER [14] is the first LLM-based test repair approach designed to repair broken test cases caused by syntactic breaking changes. Although it primarily focuses on test repair, its methodology is closely related to the test update task addressed in this study, as both aim to align test code with changes in production code.

NAIVELLM. NAIVELLM is a special baseline designed to evaluate the effectiveness of directly applying LLMs to the test update task. It simply provides the LLM with the code changes of the focal method along with the original test method. By comparing TESTUPDATER with NaiveLLM, we can better evaluate the effectiveness of our designs in context collection and iterative refinement.

B. Evaluation Metrics

Inspired by previous studies on automated test generation [16], [29], test repair [14], and automated program repair [30], we employ the following three metrics to evaluate the effectiveness of TESTUPDATER: *compilation pass rate*, *test pass rate*, and *test coverage*.

Compilation Pass Rate (CPR) refers to the proportion of generated test cases in the evaluation dataset that can compile successfully, reflecting the syntactic correctness of the generated test code.

Test Pass Rate (TPR) refers to the proportion of updated tests that can successfully execute and pass. This metric is widely adopted in studies regarding test generation [16], [31]. The test pass rate assesses whether the generated tests conform to the current system behavior.

Test Coverage (Cov.). Test coverage is a critical indicator of the effectiveness and quality of test suites. Common coverage metrics include line coverage and branch coverage, which evaluate the breadth and depth of behavioral validation performed by the test code, respectively.

C. Experimental Settings

LLM Selection. In the experiments, we evaluate three representative large language models (LLMs): Llama-3.3-70B-Instruct, GPT-4.1, and DeepSeek-V3. These models are selected to balance open-source and closed-source options, Llama-3.3-70B-Instruct and DeepSeek-V3 are open-source models, while GPT-4.1 is a closed-source commercial model. All three LLMs have a substantial number of parameters and large context windows, showing strong performance on diverse benchmarks, which is suitable for the test update task that requires powerful code understanding capabilities.

Implementation Details. We employ LangChain [32] to invoke LLM APIs and construct prompt templates. Llama-3.3-70B-Instruct is deployed locally using the vLLM [33] inference engine, whereas GPT-4.1 and DeepSeek-V3 are accessed via their respective official APIs. When evaluating the test cases, we executed them under the specific Java and Maven version recorded in UPDATES4J. To mitigate the effect of stochasticity, we evaluate each LLM three times with a temperature of 0.1 on the dataset, and report the average compilation pass rate and test pass rate for TESTUPDATER as well as the baselines.

TABLE II: Compilation and Test Pass Rates across Models (195 samples)

Model	Method	CPR (%)	TPR (%)
	CEPROT	20.5	9.7
	TARGET	45.1	41.0
Llama-3.3-70B	NAIVELLM	49.2	35.9
	SYNTER	65.1	55.3
	TESTUPDATER	**89.7**	**77.4**
GPT-4.1	NAIVELLM	45.1	36.9
	SYNTER	79.5	68.2
	TESTUPDATER	**91.8**	**84.1**
DeepSeek-V3	NAIVELLM	55.9	44.6
	SYNTER	78.5	66.7
	TESTUPDATER	**94.4**	**86.7**

VI. EVALUATION RESULTS

We formulate the following research questions (RQs) to evaluate the performance of our proposed method TESTUPDATER on the UPDATES4J dataset:

- **RQ1: Effectiveness of TESTUPDATER in Compilation and Test Pass Rate.** How effective is TESTUPDATER in test update in terms of *compilation pass rate* (*CPR*) and *test pass rate* (*TPR*)?
- **RQ2: Effectiveness of TESTUPDATER in Test Coverage.** How effective is TESTUPDATER in test update in terms of *test coverage (Cov.)*?
- **RQ3: Ablation Study.** How effective are the core components of TESTUPDATER in the test update?

A. RQ1: Effectiveness of TESTUPDATER *in Compilation and Test Pass Rate.*

To evaluate the effectiveness of TESTUPDATER in unit test update, we compare it with baseline methods under two metrics: ***compilation pass rate (CPR)*** and ***test pass rate (TPR)***. The results are shown in Table II.

Regarding two PLM-based methods, CEPROT achieves a low *CPR* of 20.5% and a *TPR* of 9.7%. This is because CodeT5 has a limited maximum output length, resulting in most of its generated tests being incomplete. Although TARGET outperforms CEPROT, its performance is still significantly lower than TESTUPDATER. The reason is that TARGET can only handle broken test cases with a single-hunk modification, also with a limited sequence length. TARGET can only generate valid outputs for 101 out of 195 cases.

As for LLM-based methods, TESTUPDATER significantly outperforms SYNTER and NAIVELLM across all three LLMs on both metrics. Compared to SYNTER (i.e., the best-performing baseline), TESTUPDATER improves *CPR* and *TPR* by at least 12.3% and 15.9%, respectively. TESTUPDATER achieves the best performance when using DeepSeek-V3 as the underlying LLM, obtaining the highest *CPR* of 94.4% and *TPR* of 86.7%. DeepSeek-V3 and GPT-4.1 achieve better performances than Llama-3.3-70B-Instruct, which indicates that our method's effectiveness is positively correlated with the performance of the underlying LLM. TESTUPDATER demonstrates the largest relative improvement over SYNTER on Llama-3.3-70B-Instruct, with enhancements of 24.6% on *CPR* and 22.0% on *TPR*, respectively. This suggests that our method yields more substantial improvements when applied to the LLM with relatively lower performance.

TABLE III: Overall Test Coverage under DeepSeek-V3

Method	Branch Cov. (%)	Line Cov. (%)
NAIVELLM	21.5	36.6
SYNTER	33.1	53.9
TESTUPDATER	46.0	69.1
GROUND TRUTH	53.0	80.4

Answer to RQ1: TESTUPDATER achieves the best performance in both compilation and test pass rates, validating its superior performance in test update. We also find that the effectiveness of TESTUPDATER is positively correlated with model capability and shows the most significant gains on weaker LLMs.

B. RQ2: Effectiveness of TESTUPDATER *in Test Coverage.*

Since TESTUPDATER achieves the highest compilation and test pass rates under the DeepSeek-V3 model in RQ1, we use the results from DeepSeek-V3 to evaluate the method's test coverage. Specifically, we compare the ***branch coverage (Branch Cov.)*** and ***line coverage (Line Cov.)*** of three methods: NAIVELLM, SYNTER, TESTUPDATER, along with the GROUND TRUTH (i.e., the developer-update test cases from the UPDATES4J dataset). Table III presents the overall coverage, which is the average line or branch coverage across all test cases in the dataset, regardless of whether the tests pass or fail.

From the results, TESTUPDATER achieves a branch coverage of 46.0% and a line coverage of 69.1%, which significantly outperforms NAIVELLM and SYNTER with improvements of at least 12%. The main reason why TESTUPDATER's coverage is still lower than the *Ground Truth* is that no matter how effective the method is, some test cases will inevitably fail and be recorded as 0% coverage.

To enable a fairer comparison of coverage between the methods, we further evaluate the jointly-passed coverage by analyzing the test cases that both methods successfully executed. Table IV compares TESTUPDATER with SYNTER on the jointly passed test cases. Table V compares TESTUPDATER with the GROUND TRUTH on the 169 test cases successfully passed by TESTUPDATER.

TABLE IV: Coverage Comparison Between TESTUPDATER and SYNTER

Method	112 Jointly Passed Tests		Remaining 73 Tests	
	Branch (%)	Line (%)	Branch (%)	Line (%)
SYNTER	49.8	80.5	12.6	21.3
TESTUPDATER	**54.8**	**81.1**	**38.3**	**58.0**

According to Table IV, among the 112 test cases successfully passed by both methods, TESTUPDATER achieves a branch coverage of 54.8% and a line coverage of 81.1%,

TABLE V: Coverage Comparison between TESTUPDATER and Ground Truth

Method	Branch Cov. (%)	Line Cov. (%)
TESTUPDATER	**52.7**	**80.2**
Ground Truth	52.1	80.1

both higher than SYNTER. In the remaining 73 test cases, the gap becomes more pronounced: TESTUPDATER achieves 38.3% branch coverage and 58.0% line coverage, significantly higher than SYNTER. These results suggest that TESTUPDATER performs better than SYNTER on easier cases (i.e., signature-related code changes), and significantly better on more complex or difficult ones that SYNTER fails to handle.

As shown in Table V, for the 169 test cases successfully passed by TESTUPDATER, the average branch and line coverage reaches 52.7% and 80.2%, slightly higher than the GROUND TRUTH. This indicates that TESTUPDATER has achieved, and even slightly exceeded, the level of code coverage provided by manually written tests. A possible explanation is that the passed tests generated by TESTUPDATER tend to systematically explore different branches and execution paths.

Answer to RQ2: TESTUPDATER outperforms SYNTER in both overall and jointly-pass test coverage, and surpasses GROUND TRUTH in individual comparisons. It indicates that TESTUPDATER is effective at verifying new functionality that SYNTER fails to cover, and sometimes even reaches or outperforms professional software developers.

C. RQ3: Ablation Study.

To investigate the individual contributions of the core components in TESTUPDATER, we conducted an ablation study by evaluating different combinations. TESTUPDATER consists of two key module: ***Context Collection*** and ***Iterative Refinement***. Removing both modules results in the baseline NAIVELLM method. Accordingly, we compared the performance of NAIVELLM, TESTUPDATER without context collection (TESTUPDATER$_{wo}$*CC*), TESTUPDATER without iterative refinement (TESTUPDATER$_{wo}$*IR*), and the full TESTUPDATER method in terms of compilation pass rate and test pass rate. The results are summarized in Table VI.

TABLE VI: Ablation Study on the Effect of TESTUPDATER Components

Method	CPR (%)	TPR (%)
NAIVELLM	55.9	44.6
TESTUPDATER$_{wo}$*IR*	83.4	67.7
TESTUPDATER$_{wo}$*CC*	90.8	82.1
TESTUPDATER	94.4	86.7

The ablation results indicate that NAIVELLM has relatively low performance, suggesting limited test update effectiveness. Adding the context collection module (TESTUPDATER$_{wo}$*IR*) yields an improvement of about 25% in compilation and test pass rate. Incorporating the iterative refinement module (TESTUPDATER$_{wo}$*CC*) leads to a higher improvement, with the compilation and test pass rate rising around 35%, highlighting the critical role of iterative refinement in improving the executability of generated code.

The comparatively smaller gain from the *context collection* module can be attributed to the fact that the LLM introduces new, error-prone test logic to enhance coverage, which necessitates correction in the iterative refinement phase. By integrating these modules, TESTUPDATER effectively enhances and updates obsolete test cases.

Answer to RQ3: In summary, both *context collection* and *iterative refinement* modules contribute positively to TESTUPDATER 's performance, with iterative refinement showing a more significant impact.

VII. DISCUSSION

A. Failure Analysis.

To further investigate the limitations of TESTUPDATER in test update tasks, we analyzed failed update cases and summarized the main causes of failure. The observed failure types can be categorized as follows:

Function Parameter Mismatch. When the focal method changes, especially involving additions or modifications of multiple parameters, the LLM may struggle to accurately infer the meaning or type of new parameters. This often leads to invocation errors in the generated test code, typically resulting in compilation failures (e.g., "cannot be applied to given types") due to incorrect argument passing.

Overly Complex Focal Method Changes. If the modification to the focal method involves large-scale refactoring, complex control flows, or deep data dependencies, the LLM may fail to fully understand the change even when context is provided. As a result, the generated test logic may not adequately cover all new functional paths or assertion conditions, leading to test failures.

Context Collection Failure. During the context collection phase, TESTUPDATER relies on the LLM to help identify key methods and classes relevant to the focal method changes. However, in some cases, the LLM returns elements not from the local source code but from external Maven dependencies. Since the language server can only index local project files, it fails to resolve these external references, resulting in missing context. Meanwhile, the LLM may fail to correctly identify required methods or classes.

B. Impact of Test Enhancement.

To examine the trade-off between benefits and overhead of test enhancement, we modified our pipeline to let the LLM perform only test repair (repair-only) and compared it with the original TESTUPDATER (repair-enhance). Table VII summarizes the overall coverages and average time overheads (Avg. Time). DeepSeek-V3 is used in this experiment because it achieved the best performance in RQ1. Compared with the repair-only workflow, the full repair-enhance workflow achieves much higher coverage with little extra time. This

shows that enabling test enhancement is crucial for verifying new functionality and improving test quality with negligible efficiency loss.

TABLE VII: Repair-Only vs. Repair-Enhance

Workflow	Branch Cov. (%)	Line Cov. (%)	Avg. Time
repair-only	35.3	54.7	70.7s
repair-enhance	46.0	69.1	71.8s

C. Evaluation on TARBENCH for Test Repair Comparison.

TARBENCH [13] is a benchmark constructed by Yaraghi et al. to evaluate the test repair performance of TARGET [13], comprising 7,103 test instances focused on single-hunk test method repairs when the system under test evolves. In contrast, our UPDATES4J benchmark focuses on general test method updates in response to changes in the focal methods. To facilitate a comprehensive comparison with existing test repair methods, we also evaluated TESTUPDATER on TARBENCH and compared its performance with TARGET.

Considering the large size of TARBENCH, we utilized a locally deployed model Llama-3.3-70B-Instruct as the underlying LLM of TESTUPDATER to accommodate the potential high costs. For TARGET, we used the best results reported in its original paper. TARGET generates 40 outputs per test case, and we ran TESTUPDATER five times per test case. A case is considered passed if any of these outputs succeed. As presented in Table VIII, TESTUPDATER achieves performance comparable to TARGET on TARBENCH, demonstrating its effectiveness in dedicated single-hunk test repair scenarios. We anticipate that by leveraging more advanced LLMs, such as GPT-4.1 or DeepSeek-V3, TESTUPDATER 's performance can be further enhanced (as presented in RQ1). Besides, in contrast to TARGET that requires task-specific pretraining, TESTUPDATER showcases a more general approach, which requires only the off-the-shelf LLMs but achieves comparable performance on single-hunk test repair scenarios and much better performance on general test update scenarios.

TABLE VIII: Test Repair Performance on TARBENCH

Method	CPR (%)	TPR (%)
TARGET	89.6	80.0
TESTUPDATER	87.4	83.6

D. Test Update vs. Test Generation

To update test cases, one can also leverage test generation tools to generate brand-new unit tests instead of updating the existing ones. To investigate the effectiveness of this method in just-in-time test update, we evaluated the state-of-the-art test generation approach HITS [29] on our dataset. All settings (e.g., prompts and post-processing strategies) were kept the same as HITS. The only difference is the underlying LLM, which we used DeepSeek-V3 with temperature set to 0.1, consistent with our experimental setup (see Section V.C) to ensure fairness. Among the 195 test cases, HITS achieved a compilation pass rate of 33.3% and a test pass rate of 18.9%, substantially lower than existing LLM-based test update baselines and our approach (see Table II).

We observed that most failures in HITS stem from import errors. Fig. 5 presents the test generated by HITS for the same case shown in Fig. 2 (which is correctly updated by our approach). Symbols highlighted in red were not imported, resulting in "cannot find symbol" errors. Although HITS iteratively attempts to fix errors based on the reported messages, these messages (see the right part of Fig. 5) only indicate the line numbers of the missing symbols, without providing their package paths for importing. Consequently, the errors remain unrecoverable regardless of the number of iterations.

E. Efficiency

The time overhead of TESTUPDATER consists of three main components: (1) initializing the language server, which typically takes from a few seconds to a few minutes; (2) invoking the LLM; and (3) compiling and executing the updated test methods. Based on our measurements, updating a single test case takes an average of 71.8 seconds and 4 LLM invocations. Although our method is less time-efficient than SYNTER (10–30 seconds) due to the necessity of compiling and executing tests in each refinement iteration, it prioritizes correctness and offers potential for future improvements.

In terms of token usage, each update consumes an average of 2,197 tokens, approximately 1,000 more than SYNTER due to more frequent invocations and larger context. However, this additional cost leads to noticeably better performance.

F. Threats to Validity

Flaky Test Detection. To eliminate the potential impact of flaky tests on our evaluation and avoid introducing instability to developers, we executed each test case 10 times under identical conditions. A test is considered flaky if it yields inconsistent outcomes across runs. Our results show that all GROUND TRUTH and TESTUPDATER-generated test cases were consistent, indicating no flakiness.

Internal Validity. The dataset used in the experiments is sourced from publicly available code repositories. While it aims to cover common types of changes in real-world software evolution, it may not fully represent all test update scenarios. Additionally, the UPDATES4J dataset may carry a risk of data leakage. However, considering that the same LLMs are used in the baseline models, the relative performance improvement of TESTUPDATER in test generation remains valid from a controlled variable perspective.

External Validity. The generalization of our method may be limited since TESTUPDATER currently targets the Maven-JUnit framework, which may not represent projects with different build systems or programming languages. However, the Maven-JUnit framework is commonly used in real-world software. Future work could extend TESTUPDATER to support more diverse ecosystems.

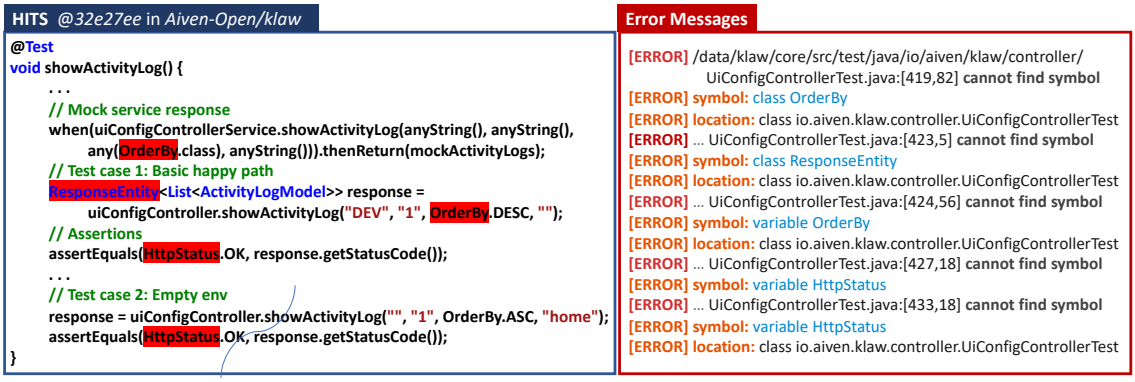

Fig. 5: An Import Error Example of HITS.

VIII. Related Works

A. Automated Test Repair

Automated test repair aims to fix broken test cases caused by changes in the corresponding production code. Early approaches relied on heuristics and program analysis. ReAssert [9] applied strategies such as literal replacement and assertion adjustment. Symbolic Test Repair [10] leveraged symbolic execution to modify failing literals. TestCareAssistant [34] combined static and dynamic analysis to adjust inputs and assertions, while Test Fix [11] formulated test repair as a search problem solved via genetic algorithms.

More recent work explores learning-based and LLM-based methods. CEPROT [12] employs a transformer-based model to identify and update obsolete test cases in a two-stage pipeline, showing strong performance across Java projects. TARGET [13] formulates test repair as a code-to-code translation task, using rule-based algorithms to collect context and finetuning models to achieve 66.1% exact match accuracy on the TARBENCH dataset. SYNTER [14] is the first LLM-based method, focusing on syntactic-breaking changes. SYNTER leverages static analysis and a neural reranker to enhance the performance of LLMs and outperform CEPROT and NAIVELLM in text similarity and accuracy. Unlike these methods, our approach not only repairs broken tests but also enhances non-broken tests to improve coverage.

B. Automated Test Generation

Automated test generation aims to reduce manual effort in writing test cases. Traditional tools such as EvoSuite [35], Randoop [36], and Symstra [37] rely on evolutionary algorithms, random testing, or symbolic execution. A series of learning-based methods has recently been proposed. AthenaTest [38] leverages BART [39] to generate test cases, while Atlas [40] and Tufano *et al.* [41] focus on generating assertion statements. However, these methods often lack guarantees of syntactic correctness or executability [42].

Recent research has explored the integration of LLMs into automated test generation. ChatTester [16] is the first LLM-based approach, demonstrating superior performance over EvoSuite and AthenaTest. Chen *et al.* [31] proposed a practical framework, ChatUniTest, employing a generate-verify-repair loop to improve accuracy and coverage. SymPrompt [43] introduces path constraint prompts inspired by symbolic execution to guide LLMs for higher code coverage. HITS [29] applies code slicing to divide complex methods into smaller units, improving both line and branch coverage. Compared with our work, we focus on repairing and enhancing existing tests in the process of software evolution, whereas these methods are directed at generating new tests for a specific method.

IX. Conclusion and Future Works

In this paper, we propose TESTUPDATER, an LLM-based approach for automated just-in-time unit test updates that can effectively update outdated unit tests caused by production code changes. TESTUPDATER leverages LLM for collecting adequate and precise context, guides test updates with carefully crafted prompts, and introduces an iterative refinement mechanism to increase the correctness of the LLM-generated tests. Notably, TESTUPDATER not only repairs broken tests but also enhances test quality by validating newly introduced logic. To support evaluation, we construct UPDATES4J, a benchmark dataset containing real-world co-evolution scenarios of production and test methods. Experimental results show that TESTUPDATER outperforms all the baselines SYNTER, NAIVELLM, CEPROT, and TARGET across compilation pass rate and test pass rate. Meanwhile, TESTUPDATER also outperforms LLM-based baselines SYNTER and NAIVELLM in test coverage. Future work includes expanding support for additional languages and frameworks, optimizing efficiency for industrial-scale deployment. Overall, this paper provides a novel approach and practical reference for continuous test maintenance during software evolution and demonstrates the potential of LLMs in automated test update.

Acknowledgment

This research/project was supported by the National Natural Science Foundation of China (No.62202420), Zhejiang Provincial Natural Science Foundation of China (No.LZ25F020003), and the Fundamental Research Funds for the Central Universities (No.226-2025-00067).

REFERENCES

[1] M. Olan, "Unit testing: test early, test often," *Journal of Computing Sciences in Colleges*, vol. 19, no. 2, pp. 319–328, 2003.

[2] P. Runeson, "A survey of unit testing practices," *IEEE software*, vol. 23, no. 4, pp. 22–29, 2006.

[3] M. Skoglund and P. Runeson, "A case study on regression test suite maintenance in system evolution," in *20th IEEE International Conference on Software Maintenance.*, 2004, pp. 438–442.

[4] Z. Lubsen, A. Zaidman, and M. Pinzger, "Using association rules to study the co-evolution of production & test code," in *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, 2009, pp. 151–154.

[5] S. Shamshiri, "Automated unit test generation for evolving software," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 1038–1041.

[6] V. Hurdugaci and A. Zaidman, "Aiding software developers to maintain developer tests," in *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, 2012, pp. 11–20.

[7] J. Kasurinen, O. Taipale, and K. Smolander, "Software test automation in practice: empirical observations," *Advances in Software Engineering*, vol. 2010, no. 1, p. 620836, 2010.

[8] D. G. Widder, M. Hilton, C. Kästner, and B. Vasilescu, "A conceptual replication of continuous integration pain points in the context of travis ci," in *Proceedings of the 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 647–658.

[9] B. Daniel, D. Dig, T. Gvero, V. Jagannath, J. Jiaa, D. Mitchell, J. Nogiec, S. H. Tan, and D. Marinov, "ReAssert: a tool for repairing broken unit tests," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 1010–1012.

[10] B. Daniel, T. Gvero, and D. Marinov, "On test repair using symbolic execution," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, 2010, pp. 207–218.

[11] Y. Xu, B. Huang, G. Wu, and M. Yuan, "Using genetic algorithms to repair junit test cases," in *Proceedings of the 21st Asia-Pacific Software Engineering Conference*, vol. 1, 2014, pp. 287–294.

[12] X. Hu, Z. Liu, X. Xia, Z. Liu, T. Xu, and X. Yang, "Identify and update test cases when production code changes: A transformer-based approach," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering*, 2023, pp. 1111–1122.

[13] A. S. Yaraghi, D. Holden, N. Kahani, and L. Briand, "Automated test case repair using language models," *IEEE Transactions on Software Engineering*, 2025.

[14] J. Liu, J. Yan, Y. Xie, J. Yan, and J. Zhang, "Fix the tests: Augmenting LLMs to repair test cases with static collector and neural reranker," in *2024 IEEE 35th International Symposium on Software Reliability Engineering*, 2024, pp. 367–378.

[15] OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad *et al.*, "Gpt-4 technical report," 2024.

[16] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Evaluating and improving chatgpt for unit test generation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1703–1726, 2024.

[17] "Official page for Language Server Protocol," https://microsoft.github.io/language-server-protocol/.

[18] W. Sun, Z. Guo, M. Yan, Z. Liu, Y. Lei, and H. Zhang, "Method-level test-to-code traceability link construction by semantic correlation learning," *IEEE Transactions on Software Engineering*, 2024.

[19] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*, 2022, pp. 1–10.

[20] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, 2024.

[21] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[22] "Abstract syntax tree," https://en.wikipedia.org/wiki/Abstract_syntax_tree.

[23] T. Zhu, Z. Liu, T. Xu, Z. Tang, T. Zhang, M. Pan, and X. Xia, "Exploring and improving code completion for test code," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 2024, pp. 137–148.

[24] M. Brunsfeld, "Tree-sitter: A parser generator tool and an incremental parsing library," https://tree-sitter.github.io/tree-sitter/.

[25] W. Sun, M. Yan, Z. Liu, X. Xia, Y. Lei, and D. Lo, "Revisiting the identification of the co-evolution of production and test code," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–37, 2023.

[26] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 309–319.

[27] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021*, 2021, pp. 8696–8708.

[28] Y. Wang, H. Le, A. D. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, "CodeT5+: Open code large language models for code understanding and generation," 2023.

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

[30] Q. Zhang, C. Fang, Y. Xie, Y. Ma, W. Sun, Y. Yang, and Z. Chen, "A systematic literature review on large language models for automated program repair," 2024.

[31] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "ChatUnitTest: A framework for LLM-based test generation," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 572–576.

[32] LangChain, "Langchain official website," https://www.langchain.com/.

[33] vLLM Developers, "vLLM: Efficient Large Language Model Serving," https://github.com/vllm-project/vllm.

[34] M. Mirzaaghaei, F. Pastore, and M. Pezzè, "Automatic test case evolution," *Software Testing, Verification and Reliability*, vol. 24, no. 5, pp. 386–411, 2014.

[35] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011, pp. 416–419.

[36] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for Java," in *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, 2007, pp. 815–816.

[37] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2005, pp. 365–381.

[38] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," 2020.

[39] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," 2019.

[40] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1398–1409.

[41] M. Tufano, D. Drain, A. Svyatkovskiy, and N. Sundaresan, "Generating accurate assert statements for unit test cases using pretrained transformers," in *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, 2022, pp. 54–64.

[42] Y. Yang, X. Xia, D. Lo, and J. Grundy, "A survey on deep learning for software engineering," *ACM Computing Surveys*, vol. 54, no. 10s, pp. 1–73, 2022.

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