Comprehend, Imitate, and then Update: Unleashing the Power of LLMs in Test Suite Evolution

Tangzhi Xu[1], Jianhan Liu[1], Yuan Yao[1], Cong Li[2], Feng Xu[1], Xiaoxing Ma[1]
[1]State Key Laboratory of Novel Software Technology, Nanjing University, China
[2]ETH Zurich, Switzerland
{xutz,jh.liu}@smail.nju.edu.cn, {y.yao,xf,xxm}@nju.edu.cn, cong.li@inf.ethz.ch

***Abstract*—Software testing plays a crucial role in software engineering, ensuring the reliability and correctness of evolving systems. Well-maintained test suites are essential for ensuring software quality. However, in modern development cycles that emphasize rapid feature iteration, the co-evolution of test suites often lags behind, leading to more appearance of obsolete tests. To this end, automated approaches for updating obsolete test code have been proposed, and recent approaches have achieved the state-of-the-art performance with the support of large language models (LLMs). This paper presents COMMITUP, a new approach that leverages LLMs to effectively automate method-level obsolete test code updates. COMMITUP mimics how humans solve the problem, first comprehending the code modifications, searching for similar examples to imitate, and finally performing the update. We evaluate COMMITUP on a curated dataset from real-world Java projects. The results demonstrate the superior performance of COMMITUP, achieving 96.4%, 94.4%, 93.1% success rates for generating compilable, runtime failure-free, and full coverage updates, respectively. We believe our study can provide new insight into LLM-based test code update. The dataset and code are available at https://github.com/SoftWiser-group/CommitUp.**

***Index Terms*—Test suite evolution, Large language model, Software quality maintenance**

I. INTRODUCTION

The test suite plays a critical role in ensuring the reliability, security, and functional correctness of modern software [1], [2]. During software development, production code is frequently modified to accommodate new requirements, fix defects, or improve structure.[3], [4] These changes often necessitate updates to the test suite to ensure the continued co-evolution of production code and test code, thereby preserving overall system quality [5], [6]. It has been observed that well-maintained test suites faithfully encode the intended program logic [7], enabling early fault detection that dramatically reduces downstream repair costs [8], [9], [10]; conversely, obsolete tests accumulate technical debt and erode development efficiency [11].

Despite its importance, sustaining the co-evolution of production code and test code remains notoriously difficult and tedious. Due to the continuous evolution of customer requirements, developers often prioritize delivering new product features over maintaining test-related documentation, thus outdated or missing test specifications have become a bottleneck for the consistent evolution of test code [12], [13]. In practice, test updates heavily rely on manual effort, remaining largely time-consuming and error-prone [14], [15], [16]. In the era of rapid commits and pervasive CI/CD pipelines, the frequency and scale of code changes amplify this burden. These trends underscore the necessity for *automated* and *effective* test code update to sustain its co-evolution with the production code [17].

With the advancement of machine learning, a growing body of work has been devoted to automating test code update during the evolution of production code. For example, CEPROT [18] fine-tunes a CodeT5 model using co-evolution edit sequences to identify and update obsolete test code; TARGET [19] formulates test repair as a language translation task, applying a two-stage fine-tuning of CodeT5+ on essential context data characterizing the test breakage to repair broken tests. Although such fine-tuning based strategies achieve favorable results (e.g., higher CodeBLEU), their performance degrades sharply when the product–test code pair is lengthy or multiple hunks are modified, which is commonplace in practice. Motivated by the superior generative capacity of large language models (LLMs), recent approaches pivot to LLM-based solutions. SYNTER [20] constructs test-specific contextual inputs by mining relevant information from the entire repository, and feeds them into LLMs for test repair. REACCEPT [21] advances this line of work by combining retrieval-augmented generation (RAG) with dynamic validation, achieving state-of-the-art performance on automated test updates.

Although LLM-based approaches have achieved remarkable results, they still suffer from the following limitations. ① *Lack of in-depth comprehension of code modifications.* Existing approaches often treat code modifications as plain text, without explicitly reasoning over behavioral changes [22]. Also, they tend to ignore contextual information in both the production code side and the test code side, making the following test code update less effective [23]. ② *Imprecise retrieval of guidance examples.* The RAG framework has been adopted by existing work, to select guidance examples of previous test update. However, existing work relies solely on embedding-based matching to identify guidance examples, which suffers from low sensitivity to code-specific identifiers [24], [25] and limited capacity for fine-grained semantic understanding[26]. ③ *Insufficient feedback from the validators.* Although feedback from validators has been incorporated in LLM-based test code update, existing methods typically use raw log outputs without any structured explanation or contextual cues. Such limited signals can obscure the true cause of failures, leading the model to produce incorrect or misaligned modifications [27].

CommitUp. To fill the gap, we propose a new approach named CommitUp to automatically update obsolete test code when the production code evolves. CommitUp systematically organizes large language models and tools into a workflow that closely mimics human-like update logic "*comprehend, imitate, and update*". Specifically, given a production code diff and the corresponding outdated test code, CommitUp works as follows. ① *Comprehension*: To facilitate deeper understanding, CommitUp explicitly analyzes the intent behind the production code changes and the original test logic, and collects both test code side and production code side contexts to support subsequent test code update. ② *Imitation*: For more precise retrieval of guidance examples, CommitUp applies a two-stage retrieval to identify the most relevant guidance example, which adds a re-ranking step on the candidate examples retrieved by the keyword-sensitive BM25 algorithm. ③ *Update*: To deliver sufficient and informative feedback, CommitUp builds an iterative generate-validate-feedback framework for test code update, with special focus on automated root-cause analysis and location based on runtime error messages.

Evaluation. To evaluate the effectiveness of CommitUp, we first construct a dataset of 248 real production-test co-evolution examples from open-source Java projects on GitHub. We then compare CommitUp against two state-of-the-art baselines: TaRGet [19], a fine-tuned model from CodeT5+, and ReAccept [21], an LLM-based approach with dynamic validation. We also consider three different backbone LLMs, including GPT-4o, Deepseek-V3, and Qwen-2.5-Coder-7B. The results demonstrate the effectiveness of CommitUp.

First, CommitUp is consistently and significantly better compared to the existing baselines. When equipped with GPT-4o, it achieves a 96.4% compilation success rate, a 94.4% runtime failure-free rate, and a 93.1% full coverage rate, outperforming the best competitor by 10.9%, 17.4%, and 18.5%, respectively. Similar improvements are observed with Deepseek-V3 and Qwen-2.5-Coder-7B.

Second, to further validate the effectiveness of our design, we conduct an ablation study evaluating the effectiveness of each component in CommitUp. The results confirm that all the components (i.e., comprehension, imitation, and update) contribute significantly to achieving the overall performance.

Third, we further analyze the experimental results and show that CommitUp is stable across different projects and various code change types, and less sensitive to input length and production code complexity.

Contribution. The main contributions of this paper include:

- *Approach.* We present a novel approach named CommitUp to automatically update the obsolete test code on the method level.
- *Dataset and Evaluation.* We curate a dataset from real-world Java projects for the production-test co-evolution task. Extensive experimental evaluations on the dataset show the superior performance of CommitUp in updating the obsolete test code.
- *Tool.* We implement CommitUp in a tool, and we will make it publicly available to facilitate follow-up research.

II. Problem Statement

This section presents the key terminology used throughout this paper and problem statement of obsolete test code update in the context of production–test co-evolution.

Production Code. We define *production code* as the function that implements specific functionality and serves as the target of testing. P_i and P_{i+1} denote the production code before and after the update, respectively. The change between them is referred to as the production code diff, denoted as Diff_p.

Test Code. *Test code* refers to the unit test function designed to verify the correctness of the corresponding production code. We use T_i to denote the unit test associated with P_i, and T_{i+1} denotes the test associated with P_{i+1} in the actual project. T_{i+1} acts as the ground-truth in the experiment. T_u denotes the test code updated by CommitUp.

Co-Evolution Pair. A *co-evolution pair* (CE-Pair) is a tuple (P_i, P_{i+1}, T_i, T_{i+1}) that captures simultaneous changes in both the production and test code.

Task Definition. This work targets the method-level co-evolution of production and test code, i.e., updating the test code in response to a corresponding update in the production code. During inference, only (P_i, P_{i+1}, T_i) are given. The goal is to generate a test function T_u that satisfies the following criteria: (1) it compiles successfully, (2) it executes without failure, and (3) it fully covers the updated regions of P_{i+1}.

III. Approach

A. Overview

Figure 1 shows the overview of CommitUp, which takes (P_i, P_{i+1}, T_i) as input and outputs T_u. It consists of three main stages: *comprehension*, *imitation*, and *update*.

- **Comprehension Stage**. This stage aims to comprehend the production code change and the original test code. It contains three components: *intent analysis*, *test context extraction*, and *calling context extraction*. During intent analysis, an LLM is used to explicitly interpret the intent behind the production code change and the test code. These interpretations are inserted into the code as annotations. The LLM is also asked to provide a suggestion of promising locations for modification in the test code. CommitUp then extracts additional contextual information, which will be used in the update stage. Specifically, it first extracts the test context from the test class file that contains T_i, capturing structural constraints and neighboring test methods. Then, it constructs the calling context for P_{i+1} by collecting relevant invoked functions within P_{i+1}. Such contextual information provides necessary background for the subsequent update.
- **Imitation Stage**. This stage provides the LLM with a CE-Pair that potentially exhibits similar change patterns to guide the update via example-based imitation. Specifically, we first construct an external CE-Pair database containing

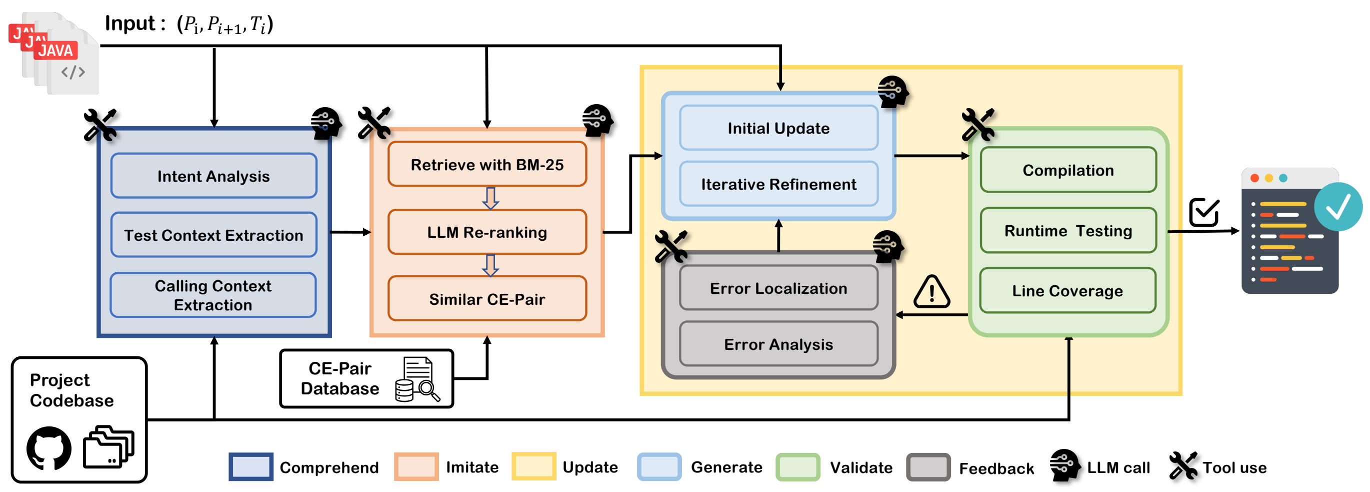


Fig. 1: Overview of CommitUp.

production code diff and its updated test code. Then, we apply a two-stage retrieval to identify the most relevant guidance example. In retrieval, we first employ the BM25 algorithm to ensure high recall by retrieving a broad set of candidate CE-Pairs based on keyword matching, followed by a re-ranking technique to accurately select the most semantically relevant match.

- **Update Stage**. Finally, CommitUp employs a *generate-validate-feedback* loop to iteratively perform the test code update. The *generator* utilizes the comprehension information and the imitation example gathered in the preceding stages to initially update the test code. The *validator* then deploys the updated test code into a real execution environment for validating whether it satisfies the requirements of compilation, execution, and coverage. If all requirements are met, the updated code is outputted as the final result T_u. Otherwise, CommitUp activates the *feedback analyzer* to analyze the failure logs and error details (e.g., uncovered lines) returned by the *validator*, identify the specific statements causing the failure/error, and generate corresponding repair suggestions. The *generator* then incorporates the information provided by the *feedback analyzer* to refine the test code update. This generate-validate-feedback loop repeats until the validation passes or a predefined maximum number of iterations is reached.

B. Comprehension Stage

1) Intent Analysis: We use an LLM to analyze the intent behind the test code update task. It comprises two sub-components: interpreting the production code change and interpreting the test code.

Interpreting Production Code Change. To guide effective test updates, it is critical to accurately capture what changed in the production code and the reasons behind those changes. Given that detailed documentation is often lacking, we leverage commit messages and code diffs as the primary information sources. Commit messages provide high-level summaries of developers' intent, while diffs expose the concrete syntactic edits. These two signals jointly enable the LLM to infer the rationale behind changes.

The LLM is instructed to annotate the production code change by generating comments adjacent to modified code lines. Each comment should concisely describe the change, its purpose, and its behavioral impact. Specifically, we provide the following instructions to LLMs:

- Each comment should clearly describe what the change does, why it is necessary, and how it affects the behavior of the code.
- The explanation should be understandable to someone unfamiliar with the codebase.

To enhance accuracy, we also introduce expert-annotated examples of complex edits, which serve as in-context demonstrations for grounding the LLM's reasoning.

Interpreting Test Code. A thorough understanding of the existing test logic is also essential for updating the test code. To support this, the LLM receives both the annotated production code change from the previous step and the original version of the test code. Similar to the production code, the LLM is then asked to summarize the intent of the test code and analyze its coverage. For test logic potentially affected by production code changes, the LLM is asked to insert a `SUGGESTION` tag with a concise explanation. These tags highlight candidate update locations for future update actions.

2) Test Context Extraction: The test class itself encodes rich contextual signals for guiding updates. In practice, auxiliary information within the class such as shared variable initialization and mock object setup tends to be stable and rarely affected by production code changes. Likewise, import statements and file-level configurations are also typically invariant across revisions. These stable elements impose structural constraints and behavioral consistency, serving as anchors for reliable test code updates. Moreover, multiple test functions within the same test class often target closely related production functions. For example, the immediately preceding test method may provide critical contextual information, such as setup logic or resource initialization (e.g., environment configuration in @Before methods). Such neighboring tests offer semantic cues and style references, thus improving the understanding of the high-level test intent.

Specifically, we perform static analysis on the test class containing T_i and extract three types of contextual signals. First, we collect the import statements via direct string matching. Then we identify class-level initialization logic including field declarations and constructors by traversing the corresponding AST nodes. Finally, we locate T_i and extract the complete body of its immediately preceding test method within the class.

3) Calling Context Extraction: To enhance LLM's understanding of P_{i+1}, we provide the implementations of its invoked functions selectively. The calling context allows LLM to better interpret the functional semantics of P_{i+1}, especially when the invoked functions are privately defined within the codebase.

Specifically, we perform static analysis on the P_{i+1} 's AST, enabling accurate extraction of invoked functions and collect their signatures into a candidate list. This list, along with Diff_p and T_i, is provided to LLM. The LLM selects the function names that require additional context. Here, we ask LLMs to select the most critical invoked functions instead of including all of them due to the context window limits of LLMs. Using the selected names, we perform a project-wide string-based matching to locate the corresponding function bodies. The retrieved function bodies along with their corresponding file names are then combined to form the cross-file calling context.

C. Imitation Stage

The assumption behind our imitation stage is that, semantically similar CE-Pairs often exhibit reusable patterns that are beneficial for guiding new test updates. Therefore, we construct an external database of CE-pairs and apply a retrieval-augmented generation (RAG) framework to identify similar examples. Retrieval is performed solely on the production code side, for it reflecting the inherent causality between production and test code evolution. This design aligns with a cause-to-effect reasoning paradigm, enabling the system to retrieve a CE-Pair that is semantically aligned with the current production code change. Once retrieved, the CE-pair is integrated into the update pipeline as a reference for test update.

1) CE-Pair Database Construction: We create the CE-Pair database containing 5100 entries from code change instances provided by CEPROT [18]. We exclude any data overlap with our evaluation dataset and strictly enforce timestamp checks to prevent data leakage. While CEPROT encodes changes as edit sequences, we instead adopt code diffs to represent the transformation between code versions. Prior work [28] shows that edit sequences, though expressive, are composed of fine-grained operations that are difficult for retrieval models to process. In contrast, code diffs preserve higher-level structural and semantic information, making them more effective for similarity computation.

2) Coarse-Grained Retrieval: Prior to retrieval, we perform automatic rule-based standardization of the code. Specifically, we normalize the code by tokenizing the raw source text while keeping all syntax-relevant symbols, ensuring that structural information in the programming language is retained. All identifiers are lower-cased to mitigate variance introduced by naming styles, and meaningless whitespace and empty tokens are removed to preserve only semantically meaningful content.

We then adopt the classical BM25 algorithm [29] to perform coarse-grained retrieval of code changes, using only the production code diff for querying. Specifically, given a production code diff Diff_p, we retrieve a few most similar production code diffs from the CE-Pair database. We compute similarity scores using BM25 with the default hyper-parameter setup (i.e., $k = 1.2$ and $b = 0.75$), since they work pretty well for most corpuses [30], [31]. The resulting CE-Pairs serve as candidate examples.

3) Re-ranking: To refine the initial retrieval results, we apply a re-ranking step to better capture deep semantic and structural similarity. We adopt `bge-reranker-v2-gemma` [32], which is a fine-tuned LLM that supports diverse retrieval augmentation needs. We use it as a cross-encoder to jointly encode the query and each of the candidate examples, and assign a relevance score to each CE-Pair.

For each query-candidate pair, we construct the input in the following format: "*Given a query code change [A] and a candidate code change [B], determine whether [B] is similar to [A] by providing a prediction of either 'Yes' or 'No'.*", and feed it into the bge-reranker-v2-gemma model. From the model's output, we extract the logit corresponding to the token "Yes" as the relevance score. A higher score indicates a stronger semantic similarity between the query and the candidate. Candidates are then re-ranked based on their relevance scores, and the top-ranked CE-Pair is selected for downstream usage.

D. Update Stage

The update stage follows a three-step iterative framework to progressively update the test code, i.e., *generate*, *validate*, and *feedback*, each playing a distinct role in the update pipeline.

1) Generator: Once the LLM completes comprehension and imitation stages, the *generator* module initiates the test update process. In the first iteration, it feeds the annotated production code diff and the test code, the test and calling context, and the example CE-Pair into the LLM to generate an initial test update. In subsequent iterations, the *generator* incorporates runtime feedback—including error messages, the locations of faulty lines, and diagnostic insights—to guide further refinement. This process continues until the generated test code can fully cover the modified parts of the production code or the maximum number of iterations is reached.

2) Validator: After the updated test code T_u is generated by the *generator*, the *validator* inserts it into the test file, then compiles and executes the updated test in a real-world production environment to verify its validity. This validation phase serves the following key objectives:

- Ensure that T_u conforms to the language's syntax and type system, allowing it to pass compilation without errors;
- Check for runtime failures during execution, such as assertion violations or uncaught exceptions;
- Assess whether T_u successfully covers the new lines introduced by the production code change, in addition to the lines previously covered by T_i.

We use JUnit to conduct test execution and use Jacoco to collect line coverage data. If T_u passes validation, it is considered a successful update, thereby completing the iterative loop. Otherwise, the *validator* provides the error message to the feedback stage for further analysis.

3) Feedback Analyzer: When T_u fails the validation stage, the *feedback analyzer* module extracts relevant error information, and formulates targeted guidance to refine the next update.

Error Localization. The *feedback analyzer* adopts tailored strategies for different failure types.

If T_u triggers a compilation error or runtime failure, COMMITUP captures the error logs from the build system or runtime environment. These logs, along with P_{i+1} and T_u, are fed into the LLM to identify all potential fault locations, expressed in the form of (filename, line number) pairs. The corresponding code snippets are then retrieved in a structured format: "*line-number: code-content*". If T_u fails to cover the desired production code, COMMITUP parses the Jacoco-generated coverage report to identify uncovered lines. These are also reported in the "*line-number: code-content*" format for consistency.

Error Analysis. After locating the errors, COMMITUP analyzes their root causes, which may include issues, such as undefined references, incorrect variable scopes, improper API usages, or version incompatibilities and etc. Based on the root cause, COMMITUP collects the execution environment configuration, error messages, and the updated test code annotated with failure locations; it then generates targeted repair suggestions and submits them to the *generator* for incremental refinement.

In cases of repeated failure, COMMITUP optionally adopts a *soft rollback* strategy. Rather than relying on a fixed threshold of failures, the decision is delegated to the LLM itself. At the end of the error-feedback prompt, we instruct the LLM that if it encounters failures across multiple iterations with no clear solution, it may choose to comment out or delete the problematic lines. Once such an action is performed, the process continues with the next iteration, and the validator's feedback is supplied to guide further corrections. This flexible mechanism not only mitigates cascading errors caused by faulty generations but also preserves useful progress, leading to more human-like debugging behavior and a cleaner context for subsequent updates.

IV. EXPERIMENTAL SETUP

A. Dataset Construction

Existing datasets for evaluating obsolete test code updates suffer from limited scale, insufficient diversity, and complex setup requirements, which hinder reproducibility and limit their effectiveness in benchmarking method performance. For example, TARBENCH [19] is restricted to updates only involving a single modification block. To this end, we devote considerable effort to ensuring the constructed dataset maintains sufficient scale, diversity, and ease of deployment, by automating the execution environment, enforcing consistency across runs, selecting diverse and high-quality code repositories, and thus better supporting the evaluation of generalization and effectiveness.

TABLE I: Details of the curated dataset. "C/S" means "Commits/Samples", representing the number of collected commits and samples, respectively.

Project	**C/S**	**Project**	**C/S**
commons-lang	2/3	dddlib	15/48
datumbox-framework	3/3	openmrs-core	32/36
basex	74/122	OpenID	2/3
wildfly	1/1	bitcoinj	1/1
DependencyCheck	11/12	spring	2/2
wicket	3/3	jetty.project	2/3
java-sdk	7/8	curator	1/1
bonecp	1/2	**Total**	157/248

Specifically, we begin by utilizing Java projects from the REACCEPT dataset [21], thus forming a good initial set for our purpose. From this set, we select repositories that use Maven as their build system and have at least 400 stars. For each commit C in the selected projects, we analyze changes within unit test methods. As defined in Section II, we assume P_{i+1} represents the code changes introduced in C, and P_i corresponds to its parent commit. We then match code changes by method name strictly (e.g., Method $\Rightarrow$ MethodTest) and further conduct manual verification to extract all CE-Pairs. Subsequently, we manually ensure each test method is influenced by only one production method change within the commit and retain only those CE-Pairs that successfully compile. Additionally, we include only those pairs where T_{i+1} fully covers the modified parts of P_{i+1} in our dataset. To ensure that each sample in our dataset is an obsolete test code, we replace T_{i+1} with T_i in C. If this causes compilation error, run-time failure, or insufficient coverage, the sample is retained in the dataset. In this context, T_{i+1} serves as the ground truth for text similarity based evaluation.

The details of the dataset are provided in Table I. The "Project" column lists the (abbreviated) names of the collected projects, and the "C/S" column represents the number of collected commits and extracted samples per project (a single commit may contain multiple samples). Overall, we collect 248 samples from 157 commits across 15 projects.

B. Baselines

We compare COMMITUP against the following baselines:

- **TaRGet** [19]. TaRGet leverages pre-trained code language models for automated test code update. It formulates the problem as a language translation problem and fine-tunes CodeT5+ using essential contextual information characterizing test failures.
- **LLM-Standalone**. This is a basic LLM-based method. We directly provide (P_i, P_{i+1}, T_i) along with a task description to the LLM.
- **Synter** [20]. SYNTER presents a framework that augments LLMs with a static collector and neural reranker to automatically repair test cases affected by syntactic breaking changes.
- **ReAccept** [21]. ReAccept is the state-of-the-art LLM-based approach for automatic unit test updates. It builds an LLM agent based on the ReAct [33] framework, combining retrieval-augmented generation (RAG) and dynamic validation techniques to update the test code.

For TaRGet, wo use the open-source best-trained model.[1] Since TaRGet supports only a single test breakage per input, for data instances with multiple modifications, we applied TaRGet iteratively, feeding one modification at a time and merging the outputs as the final result. In cases where our dataset samples exceeded TaRGet's maximum input token limit (512 tokens), we treat the result as a compilation error. For LLM-Standalone and Synter, we use the same configuration as CommitUp. For ReAccept, we adopt the best-performing configuration reported in the paper, including temperature = 0, top-p = 1, and dynamic validation iterations = 6.[2]

C. Evaluation Metrics

To comprehensively assess the quality of the updated test code, we employ two types of evaluation metrics: one based on text similarity, and the other based on execution results.

Text Similarity Metrics. Following prior work [19], we use CodeBLEU [34] to measure the textual similarity between the updated test cases and the ground-truth tests. CodeBLEU is designed specifically for evaluating code generation tasks. Compared to traditional BLEU, CodeBLEU accounts for both syntactic and semantic correctness in code.

Execution-Based Metrics. Text similarity metrics such as CodeBLEU have been shown to correlate poorly with functional correctness and may assign high scores to tests that fail to compile or execute [35], [36], [37]. Therefore, following recent work on test generation [38], [39], [40] and code evaluation benchmarks [41], [42], [43], we adopt execution-based metrics as the primary evaluation criteria. Although it is more time-consuming and harder to configure, it provides a more realistic measure of functional correctness. Ensuring successful compilation and execution is paramount in practice, rendering execution-based metrics more critical than text similarity metrics.

[1]https://figshare.com/articles/dataset/_b_TaRBench_A_Comprehensive_Benchmark_for_Automated_Test_Case_Repair_b_/25008893

[2]https://github.com/Timiyang-ai/REACCEPT

Execution outcomes are categorized into four types: (1) compilation error; (2) compilation success but runtime failure; (3) runtime success without fully covering the production code; (4) runtime success with full line coverage of the modified parts of production code. We choose line coverage instead of branch coverage because some cases do not contain any branches, making it meaningless to measure branch coverage for the production code. Based on these outcomes, we define the following metrics:

- **Compilation Success Rate (CSR)**: The percentage of updated test code that compiles successfully.
- **Runtime Failure-Free Rate (RFR)**: The percentage of updated test code that executes without runtime failures.
- **Full Coverage Rate (FCR)**: The percentage of updated test code whose execution achieves complete line coverage of the modified parts of P_{i+1}.

D. Configurations

We employ three large language models in this study: OpenAI's GPT-4o, Deepseek's Deepseek-v3, and Qwen's Qwen-2.5-coder-7b, for these models represent the leading closed-source model, the representative open-source model, and a lightweight open-source alternative, respectively.

GPT-4o and Deepseek-v3 are accessed via paid APIs, while qwen-2.5-coder-7b is deployed locally on a server equipped with a NVIDIA GeForce RTX 3090 GPU. All models are configured with temperature = 0 and top-p = 1. For compilation and coverage verification, we use JDK 8, Maven 3.2.5, and JaCoCo 0.8.7.

In the imitation stage, we populate the retrieval database with 5,100 entries and select the top-10 candidates for coarse-grained retrieval. For the `bge-reranker-v2-gemma` model, we use an embedding dimension of 768 and cap the input length at 1,024 tokens. In the update stage, we limit the maximum number of refinement iterations to 6.

V. Experimental Results

We study the following research questions:

- **RQ1 (Effectiveness Comparison):** How does CommitUp compare with the state-of-the-art techniques?
- **RQ2 (Ablation Study):** How does each component of CommitUp contribute to the final performance?
- **RQ3 (Applicability and Stability):** How stable does CommitUp perform across different situations?

A. RQ1:Effectiveness Comparision

We first evaluate CommitUp and baseline methods on the collected dataset. The results are shown in Table II. Overall, CommitUp, backed by GPT-4o, significantly outperforms all baselines on execution-based metrics, achieving CSR/RFR/FCR scores of 0.964/0.944/0.931, respectively. Compared to the strongest baseline (GPT-4o-backed ReAccept), CommitUp improves CSR, RFR, and FCR metrics by 10.9%, 17.4%, and 18.5%, respectively. Regarding CodeBLEU, ReAccept achieves the highest score (with GPT-4o). This result indicates that generated tests with greater textual similarity to the

TABLE II: Effectiveness comparison of different methods backed with different LLMs. "LLM-SA" refers to the LLM-Standalone baseline. "DS-V3", "QC-7B" refers to Deepseek-V3 and Qwen-2.5-coder-7b. The best results are highlighted. The average execution time and cost is also reported.

Method	Model	CodeBLEU	CSR	RFR	FCR	Time	Cost
TaRGet	CodeT5+	0.691	0.391	0.286	0.093	0.5s	-
LLM-SA	QC-7B	0.403	0.298	0.290	0.290	0.6s	-
	DS-V3	0.724	0.520	0.379	0.375	2s	0.0005$
	GPT-4o	0.779	0.589	0.456	0.448	1.2s	0.0040$
Synter	QC-7B	0.694	0.427	0.383	0.286	16s	-
	DS-V3	0.749	0.597	0.548	0.399	18s	0.0008$
	GPT-4o	0.735	0.665	0.621	0.516	17s	0.0060$
ReAccept	QC-7B	0.690	0.472	0.379	0.351	7s	-
	DS-V3	0.792	0.806	0.746	0.734	9s	0.0013$
	GPT-4o	**0.905**	0.855	0.770	0.746	7s	0.0100$
CommitUp	QC-7B	0.542	0.544	0.452	0.415	9s	-
	DS-V3	0.762	0.927	0.883	0.875	11s	0.0015$
	GPT-4o	0.819	**0.964**	**0.944**	**0.931**	13s	0.0140$

ground truth, do not necessarily mean better success rates in compilation, execution, and production code coverage.

The reasons for the improvement are as follows. TaRGet, originally designed for repair test breakage, is not optimized for broader test update scenarios; additionally, its smaller backbone model constrains its capability on complex or lengthy inputs. Similarly, Synter is specifically designed to repair compilation errors caused by syntactic breaking changes, and thus shows inferior performance across broader test update scenarios. Compared to ReAccept, CommitUp's performance improvement is primarily attributed to two key differences: 1) a deeper semantic comprehension that provides annotations to the current code modifications, and 2) an elaborate feedback mechanism that analyzes the error location and root cause.

CommitUp also maintains its advantage across different LLM backbones. When using Deepseek-V3, CommitUp outperforms ReAccept by 12%, 14%, and 14% on CSR, RFR, and FCR, respectively, and surpasses LLM-Standalone by over 40%–50% across all three metrics. Even compared to GPT-4o-backed ReAccept, Deepseek-V3-backed CommitUp exhibits noticeable advantages. These results demonstrate that CommitUp effectively harnesses the potential of the underlying LLM to update the test code. Moreover, as the capacity of the base LLM improves, the performance of CommitUp consistently improves.

In addition to effectiveness, CommitUp is also efficient in both time and cost. We report the average execution time and LLM token cost for each test case update problem in Table II. Here, the execution time does not include the compilation time, and the cost of locally deployed methods is excluded. As can be observed from the table, CommitUp achieves the best results with only moderate runtime (9–13s) and low cost (0.0015–0.0140$ per update), whereas competing methods such as Synter require 16–19s per update, and ReAccept incurs a comparable cost but delivers inferior performance compared to CommitUp. Notably, CommitUp backed by Deepseek-V3 achieves even better results than ReAccept powered by GPT-4o, while reducing the cost to nearly 1/7. This demonstrates that CommitUp strikes a favorable balance between effectiveness and efficiency, making it practical for adoption.

TABLE III: Contribution of each component for updating obsolete test code (backed with GPT-4o), when each component is gradually integrated. Percentages in parentheses represent the enhancement to their preceding stage.

Phase	CSR	RFR	FCR
LLM-SA	0.589	0.456	0.448
+ Comprehension	0.706 (+20%)	0.532 (+17%)	0.524 (+17%)
+ Imitation	0.754 (+7%)	0.621 (+17%)	0.617 (+18%)
+ Error Analysis	0.915 (+21%)	0.815 (+31%)	0.802 (+30%)
+ Iterative Framework	0.964 (+5%)	0.944 (+16%)	0.931 (+16%)

Summary: CommitUp consistently and significantly outperforms state-of-the-art baselines in updating obsolete test code, especially on execution-based metrics. While all methods show better results with GPT-4o, CommitUp with DeepSeek-V3 is even better than the baselines with GPT-4o.

B. RQ2: Ablation Study

We next analyze the contribution of each component to performance improvement through an ablation study. Compared to directly using an LLM for generation, CommitUp introduces four components to produce high-quality test code updates: (1) *comprehension* (intent analysis, calling and test context extraction), (2) *imitation* (extraction of similar CE-Pairs), (3) *error analysis* (without the iterative loop), and (4) *iterative framework* (the generate-validate-feedback loop). The first ablation study starts with the basic LLM-based generation (i.e., LLM-Standalone), and progressively adds each component until the full method is assembled. The results are presented in Table III.

The monotonic growth across stages confirms that all components contribute positively. Specifically, the *comprehension* stage/component primarily attacks the lack of understanding of the production code diff and the test code, boosting the compilation success rate (CSR) by 20% and reducing runtime failures by 17% (RFR). When *imitation* is enabled, a CE-Pair with similar change pattern further improves the performance, especially on RFR (+17%) and FCR (+18%). *Error analysis* achieves the most significant performance gains, yielding 21% CSR, 31% RFR, and 30% FCR improvements. The main reason is that this component guides the generator towards concrete defect locations. Finally, by providing a refinement loop, the *iterative framework* drives all the three evaluation metrics to

TABLE IV: Contribution of each component (backed with GPT-4o), when it is removed from COMMITUP. Percentages in parentheses indicate the relative performance decrease compared to COMMITUP when the corresponding component is removed.

Phase	CSR	RFR	FCR
COMMITUP	0.964	0.944	0.931
w/o Comprehension	0.903(-6%)	0.851(-10%)	0.831(-11%)
w/o Imitation	0.923(-4%)	0.859(-9%)	0.847(-9%)
w/o Error Analysis	0.895(-7%)	0.839(-11%)	0.815(-13%)
w/o Iterative Framework	0.915(-5%)	0.815(-14%)	0.802(-14%)

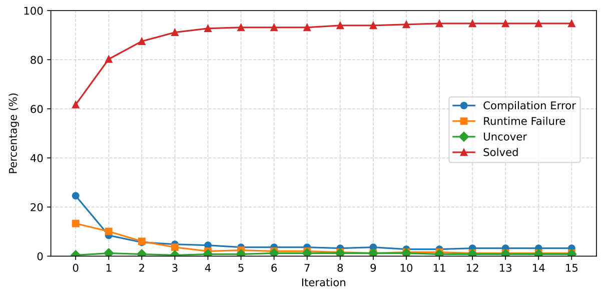


Fig. 2: Performance curves of COMMITUP under the number of update iterations (up to 15 iterations).

over 90%, especially benefiting the RFR and FCR metrics (both +16%).

We also conducted another ablation study, starting from COMMITUP and removing one component each time to verify the effect of the removed component. The results are shown in Table IV. The removal of each component leads to a degradation in COMMITUP's performance. For example, removing the *comprehension* stage leads to a noticeable drop across all metrics, especially on FCR (-11%). The absence of *error analysis* produces the most severe declines on CSR (-7%), confirming that pinpointing defect locations is critical for effective corrections. Discarding the *iterative framework* has the strongest negative effect on runtime and functional success rates (–14%), emphasizing the necessity of iterative refinement to achieve robust improvements. Overall, each component plays a complementary role, and their joint integration enables COMMITUP to reach consistently high performance across all evaluation dimensions.

In addition, to investigate the impact of the number of iterations in our iterative framework, we further extend the maximum number of iterations to 15 for evaluation. As shown in Figure 2, the first feedback cycle delivers the greatest improvement: the proportion of solved updates jumps from 61% to 80%. Similar results are observed in compilation error and runtime failure rates. As the number of iterations increases, the number of cases satisfying both compilation and coverage requirements steadily grows. This improvement stabilizes after the fourth iteration, with FCR gradually rising from 92.7% to 94.7% by the fifteenth iteration.

Summary: All the components of COMMITUP contribute positively and significantly. The error analysis component provides the most relative improvements; the imitation stage and the iterative framework are more useful to reduce runtime failures and improve code coverage.

C. *RQ3: Applicability and Stability*

Next, we evaluate COMMITUP's performance across multiple dimensions: (1) per-project effectiveness, (2) sensitivity to input length, (3) effectiveness of different code change types, and (4) impact of code complexity. In this part, we report the FCR metric for simplicity, while similar results are observed on the other metrics. We compare COMMITUP against REACCEPT under two backbone models, Deepseek-V3 and GPT-4o, denoted as COMMITUP-DS, COMMITUP-GPT and REACCEPT-DS, REACCEPT-GPT, respectively.

Per-Project Effectiveness. We first report the effectiveness of COMMITUP across different projects in Figure 3, where we select projects with more than three test cases. Overall, COMMITUP consistently outperforms REACCEPT, regardless of the underlying LLM backbone or the specific project. Moreover, COMMITUP is more stable, achieving 83%-100% success rate (FCR); in contrast, REACCEPT exhibits substantial performance fluctuations across projects, with success rates ranging from as low as 33% to as high as 100%.

Sensitivity to Input Length. Prior studies [44], [45] have shown that excessive method length significantly increases maintenance complexity, making manual test suite evolution more challenging. An open question remains: does method length similarly impact the effectiveness of automated approaches, especially LLM-based approaches? To investigate this factor, we measure the total number of tokens in ($Diff_p$, T_i) and group samples into four buckets with [0–256), [256–512), [512–1024), and [1024+] tokens. The results in Figure 4 show that, COMMITUP outperforms REACCEPT in all cases. Moreover, COMMITUP is more stable across different buckets, while REACCEPT becomes less effective when the length exceeds a certain limit (e.g., 512 tokens).

Effectiveness of Different Edit Types. In most version control systems, source code changes typically fall into three categories: CREATE, EDIT, and DELETE. For production-test co-evolution, the most common edit-type combinations are *EDIT–EDIT*, *CREATE–CREATE*, and *DELETE–DELETE* [46]. We evaluate the performance of COMMITUP on each category in Figure 5. For the most common EDIT-EDIT case, COMMITUP significantly outperforms REACCEPT. In the rarer but more challenging CREATE–CREATE case (which means the production code introduces a new code block and the test code adds new tests specifically for it), REACCEPT struggles significantly, with only 42.9% accuracy. In contrast, COMMITUP shows remarkable effectiveness, demonstrating its ability to synthesize new test logic.

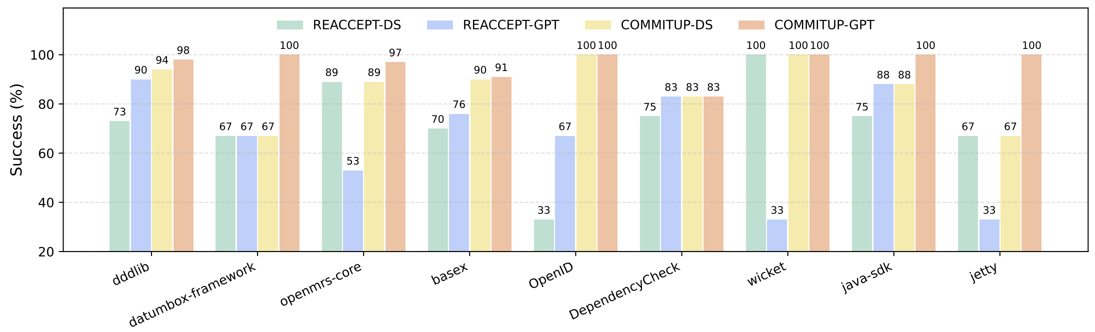


Fig. 3: The success rates (FCR) of different projects. COMMITUP is more stable across projects.

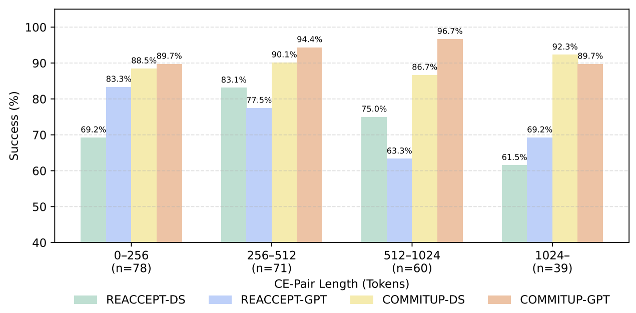


Fig. 4: The success rates (FCR) with different input lengths. COMMITUP is better and more stable, while REACCEPT becomes less effective when the CE-Pair length becomes larger.

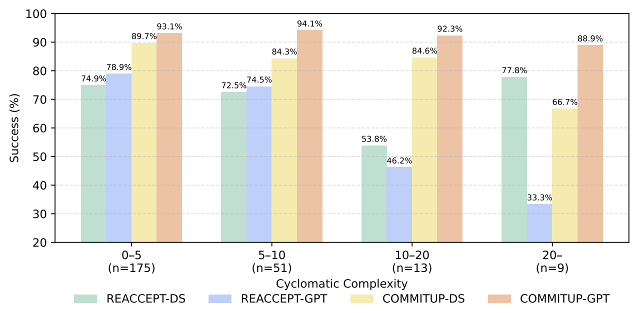


Fig. 6: The success rates (FCR) with different levels of code complexity. COMMITUP is more stable, while REACCEPT becomes less effective when the production code becomes more complex.

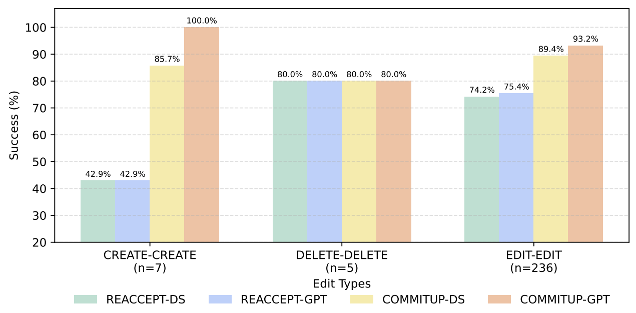


Fig. 5: The success rates (FCR) of different code edit types. COMMITUP is more stable, while REACCEPT is less effective when both the production and test code introduce new functionality (the CREATE-CREATE case).

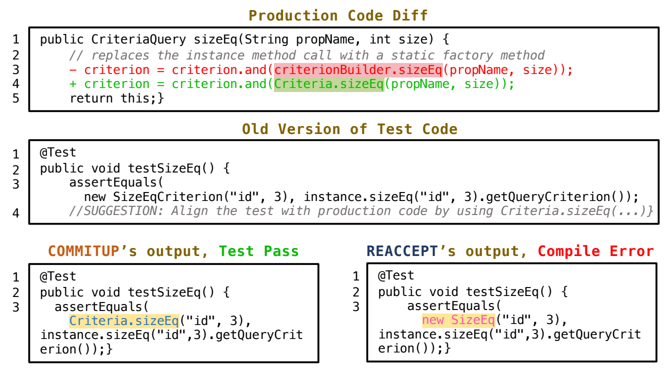
Impact of Code Complexity. Intuitively, functions with higher complexity are known to be more error-prone and harder to maintain [47], which also makes the associated test code more challenging to update. We next evaluate whether our approach remains effective when the code complexity increases. Specifically, we group samples by the cyclomatic complexity [48] of P_{i+1}. The results are shown in Figure 6. COMMITUP maintains consistently strong performance across varying levels of code complexity. In contrast, when the production code becomes more complex, REACCEPT begins to degrade notably. For example, when the cyclomatic complexity reaches 10–20, REACCEPT-GPT drops to 46.2% and REACCEPT-DS to 53.8%, while COMMITUP-GPT still achieves 92.3% FCR.

Summary: COMMITUP is more stable across different projects and different code change types, and less sensitive to CE-Pair length and production code complexity.

VI. DISCUSSIONS

Case Study. We conducted a case study and show two successfully updated samples of COMMITUP from *dddlib*. Due to space constraints, only the essential content is presented in the figure. Figure 7 highlights the strength of COMMITUP's *comprehension* stage. It accurately interprets the intent and provides correct suggestions, achieving success in the first attempt, while REACCEPT repeatedly fails. Figure 8 demonstrates the advantage of the feedback mechanism in our *update* stage. With our error localization and root cause analysis, COMMITUP successfully corrects the update in the second iteration. In contrast, REACCEPT, relying solely on raw error messages, accumulates errors over iterations and diverges significantly from the correct result.

Results on CodeBLEU. One threat to validity is from the CodeBLEU results. It has been argued that when the reference is collected from the real world, high CodeBLEU represents high readability and style consistency, thus better


Production Code Diff

```
public CriteriaQuery sizeEq(String propName, int size) {
    // replaces the instance method call with a static factory method
    - criterion = criterion.and(criterionBuilder.sizeEq(propName, size));
    + criterion = criterion.and(Criteria.sizeEq(propName, size));
    return this;}
```

Old Version of Test Code

```
@Test
public void testSizeEq() {
    assertEquals(
      new SizeEqCriterion("id", 3), instance.sizeEq("id", 3).getQueryCriterion());
    //SUGGESTION: Align the test with production code by using Criteria.sizeEq(...)}
```

COMMITUP's output, Test Pass

```
@Test
public void testSizeEq() {
  assertEquals(
    Criteria.sizeEq("id", 3),
instance.sizeEq("id",3).getQueryCrit
erion());}
```

REACCEPT's output, Compile Error

```
@Test
public void testSizeEq() {
    assertEquals(
      new SizeEq("id", 3),
instance.sizeEq("id",3).getQueryCrit
erion());}
```

Fig. 7: Example from *dddlib-566855c1*. REACEEPT failed this example. In contrast, COMMITUP successfully generates the updated test code in the first update. The key reason is due to the comprehension stage of COMMITUP, which accurately identifies the intent behind the change and provides precise suggestions to guide the update.

maintainability [49], [50]. Since recent leading LLMs have demonstrated scoring patterns on code-related tasks that are highly consistent with human evaluations [51], [52], we use LLMs to assess whether the code updated by COMMITUP maintains comparable quality. To avoid self-preference bias from the same model [53], [54] to assess the results generated by GPT-4o. We provide the LLM with expert-written criteria and ask it to evaluate the code from three dimensions: *clarity*, *readability*, and *consistency*. Each dimension is scored on a scale from 1 (poor) to 5 (excellent). As shown in the Figure 9, COMMITUP achieves strong performance across all evaluation dimensions, with scores consistently above 4.0 and close to expert-written examples. This makes COMMITUP a viable solution in practical development settings.

Data Leakage. Another potential threat to validity lies in data leakage: some ground-truth instances in our constructed dataset may have appeared in the training data of LLMs. However, we observe that under identical LLM configurations, COMMITUP consistently outperforms both LLM-standalone and prior LLM-based approaches, demonstrating that our workflow plays a central role in enabling accurate and effective test code updates.

Instability of LLMs' Output. LLMs are inherently non-deterministic due to their probabilistic decoding mechanisms. This introduces a potential threat to the reliability and reproducibility of our evaluation. To mitigate this threat, we adopt several strategies. First, we set the temperature to 0 to reduce randomness. Second, we design prompts using a standardized structure to minimize ambiguity and encourage stable responses. Third, we employ a fixed workflow which ensures consistent execution across all instances. Finally, we evaluate our method using multiple LLM backbones, demonstrating its robustness and consistent effectiveness across different models.

Validity and Redundancy of T_u. Among the 231 updated tests T_u whose executions cover the modified parts of P_{i+1}, following existing work [55], [56], [57], we further verify if P_i and P_{i+1} produced different outcomes upon the execution of T_u. Specifically, we found 226 of them (97.8%) led to different outcomes (e.g., P_{i+1} passed the new assertions that P_i failed, etc.), indicating that the generated tests are highly effective at exposing behavioral differences. Among them, we found 19 updated tests even achieved higher coverage than the ground truth T_{i+1}. Another possible issue is that COMMITUP may generate redundant test cases. However, this is limited by design as the iterative process of COMMITUP stops as soon as the generated tests satisfy the coverage requirement. Additionally, we manually checked the 19 cases with higher coverage, and found that removing any of the corresponding additional tests reduced the line coverage, indicating that these tests are not redundant.

Limitations. First, our experiments are conducted exclusively on Java projects, including the associated validation environment. Consequently, adapting COMMITUP to other programming languages may require additional engineering effort, potentially affecting its ease of use in non-Java contexts. Nevertheless, the core design of COMMITUP is language-agnostic and does not rely on Java-specific features, suggesting that it can be reasonably ported to other languages. Second, our evaluation may not fully guarantee semantic correctness—e.g., cases with missing oracles or loose boundaries may score well but fail to verify intended logic. In rare scenarios, the production code itself may contain a bug while the test remains correct, yet such cases still manifest as run-time failures.[58] While such issues were not observed, more precise validation may be needed.

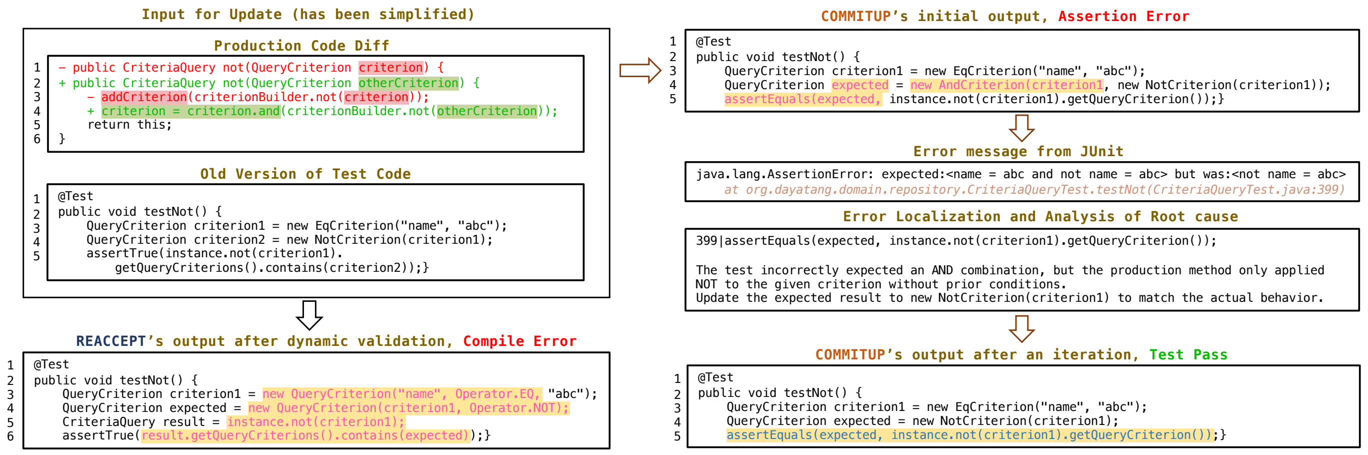
VII. RELATED WORK

A. Co-Evolution of Production-Test Code

Automated maintenance of unit tests typically involves two sequential tasks: (1) detecting when existing tests become obsolete after a production code change, and (2) updating or repairing those tests so that the test suite remains compatible with the new production code.

Fir the obsolete test detection problem, Lubsen et al. [59], [60] introduces an association rule mining approach to analyze the co-evolution of production and test code. CEMENT [61] is a static technique that infers evolutionary coupling between test and production code units based on their co-change history. SITAR [46] conducts an empirical study to identify key factors that correlate with test update needs. CHOSEN [62] examines common assumptions used in constructing co-evolution datasets, and proposes a two-stage strategy to more accurately identify product–test co-evolution instances. CEPROT [18] improves test obsolescence identification by fine-tuning a CodeT5-encoder-based classifier on edit sequences. REACCEPT [21] advances the state-of-the-art by leveraging large language models to extract "identification experience" from co-evolution examples, combining it with template retrieval to significantly enhance identification accuracy.

A growing body of methods have been proposed to update the obsolete test code. Traditional methods are mainly pattern-


Input for Update (has been simplified)
Production Code Diff
Old Version of Test Code
REACCEPT's output after dynamic validation, Compile Error
COMMITUP's initial output, Assertion Error
Error message from JUnit
Error Localization and Analysis of Root cause
COMMITUP's output after an iteration, Test Pass

```
- public CriteriaQuery not(QueryCriterion criterion) {
+ public CriteriaQuery not(QueryCriterion otherCriterion) {
    - addCriterion(criterionBuilder.not(criterion));
    + criterion = criterion.and(criterionBuilder.not(otherCriterion));
    return this;
}
```

```
@Test
public void testNot() {
    QueryCriterion criterion1 = new EqCriterion("name", "abc");
    QueryCriterion criterion2 = new NotCriterion(criterion1);
    assertTrue(instance.not(criterion1).
        getQueryCriterions().contains(criterion2));}
```

```
@Test
public void testNot() {
    QueryCriterion criterion1 = new QueryCriterion("name", Operator.EQ, "abc");
    QueryCriterion expected = new QueryCriterion(criterion1, Operator.NOT);
    CriteriaQuery result = instance.not(criterion1);
    assertTrue(result.getQueryCriterions().contains(expected));}
```

```
@Test
public void testNot() {
    QueryCriterion criterion1 = new EqCriterion("name", "abc");
    QueryCriterion expected = new AndCriterion(criterion1, new NotCriterion(criterion1));
    assertEquals(expected, instance.not(criterion1).getQueryCriterion());}
```

```
java.lang.AssertionError: expected:<name = abc and not name = abc> but was:<not name = abc>
    at org.dayatang.domain.repository.CriteriaQueryTest.testNot(CriteriaQueryTest.java:399)
```

```
399|assertEquals(expected, instance.not(criterion1).getQueryCriterion());

The test incorrectly expected an AND combination, but the production method only applied
NOT to the given criterion without prior conditions.
Update the expected result to new NotCriterion(criterion1) to match the actual behavior.
```

```
@Test
public void testNot() {
    QueryCriterion criterion1 = new EqCriterion("name", "abc");
    QueryCriterion expected = new NotCriterion(criterion1);
    assertEquals(expected, instance.not(criterion1).getQueryCriterion());}
```

Fig. 8: Example from *dddlib-32f7a3d1*. REACEEPT failed this example. In contrast, COMMITUP successfully generates the updated test code in the second update. The key reason is due to the feedback analysis module of COMMITUP, which precisely identifies the error location and root cause, thus enabling a successful correction.

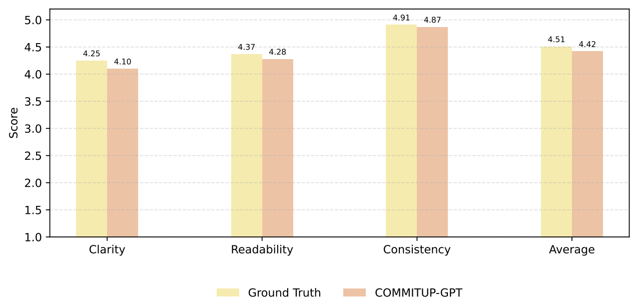


Fig. 9: Comparison of COMMITUP and ground truth across Clarity, Readability, Consistency, and overall Average scores.

based. For example, Mehdi et al. [63] introduced TestCare-Assistant, which uses information available in existing test cases and defines a set of heuristics to repair test cases. Shimmi and Rahimi [64] leveraged production-test co-evolution patterns to recommend relevant test cases for code changes. Recent work has resorted to machine learning techniques. CEPROT [18] fine-tunes a CodeT5 on code-edit sequences to translate an outdated test into an updated version, on a multi-project benchmark. TARGET [19] fine-tunes pre-trained code language models with change-focused context to translate broken Java unit tests into compilable, behavior-preserving repairs. SYNTER [20] automatically repairs obsolete unit tests by using static analysis to gather candidate contextual code, neural reranking to select the most relevant snippets, and a large language model to generate the corrected tests. REACCEPT [21] couples retrieval-argumented prompts with dynamic validation techniques to automate the co-evolution of production and test code, achieving the state-of-the-art performance.

B. Unit Test Generation

Our work is also related to unit test generation. Existing work includes search-based [1], [65], deep learning-based [66], and LLM-based approaches [38], [67], [68].

Evosuite [1] is a typical search-based test generation tool, which automatically generates high-coverage test suites for Java programs and infers assertions by leveraging genetic algorithms. ATHENATEST [66] is a deep learning approach for generating test cases. It trains a Transformer to generate tests from a large corpus of focal methods and test cases. CODEMOSA [69] injects LLM-generated seed tests into SBST tools to escape insufficient coverage, enabling search to reach hard-to-cover branches and lines that pure evolutionary methods miss. ChatUniTest [67] adopts an adaptive focal context retriever and a generation-validation-repair loop around an LLM. LLMSym [70] combines LLM-driven code synthesis with symbolic execution and Z3 solving, generating inputs that satisfy path constraints and significantly increasing Python coverage over standalone symbolic tools. HITS [39] slices long focal methods into prompt-sized segments so the LLM can generate tests segment-by-segment, yielding higher line and branch coverage than EvoSuite and earlier LLM baselines.

VIII. CONCLUSIONS

In this paper, we propose COMMITUP to update the obsolete test code as the production code evolves. COMMITUP adopts a comprehend-imitate-update workflow, and refines the update stage with an iterative generate-validate-feedback framework. Experimental evaluations on a dataset collected from real projects show the effectiveness of COMMITUP. In the future, we plan to evaluate COMMITUP in human-perceived standards and extend our approach to other programming languages.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (Grants #62025202), the Frontier Technologies R&D Program of Jiangsu (BF2024059), and the Collaborative Innovation Center of Novel Software Technology and Industrialization. Yuan Yao is the corresponding author.

References

[1] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *ESEC/FSE '11*, 2011.

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

[3] Q. L. Dilavrec, D. E. Khelladi, A. Blouin, and J.-M. Jézéquel, "Untangling spaghetti of evolutions in software histories to identify code and test co-evolutions," *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 206–216, 2021.

[4] S. Levin and A. Yehudai, "The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes," *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 35–46, 2017.

[5] A. Zaidman, B. V. Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, pp. 325–364, 2008.

[6] A. Zaidman, B. V. Rompaey, S. Demeyer, and A. van Deursen, "Mining software repositories to study co-evolution of production & test code," *2008 1st International Conference on Software Testing, Verification, and Validation*, pp. 220–229, 2008.

[7] H. M. Abushama, H. A. Alassam, and F. A. Elhaj, "The effect of test-driven development and behavior-driven development on project success factors: A systematic literature review based study," *2020 International Conference on Computer, Control, Electrical, and Electronics Engineering (ICCCEEE)*, pp. 1–9, 2021.

[8] B. W. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Computer*, vol. 34, pp. 135–137, 2001.

[9] J. Campos, A. Arcuri, G. Fraser, and R. Abreu, "Continuous test generation: enhancing continuous integration with automated test generation," *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014.

[10] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," *2013 35th International Conference on Software Engineering (ICSE)*, pp. 802–811, 2013.

[11] R. L. Nord, I. Ozkaya, P. B. Kruchten, and M. Gonzalez-Rojas, "In search of a metric for managing architectural technical debt," *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pp. 91–100, 2012.

[12] A. U. Rehman, A. Nawaz, M. T. Ali, and M. Abbas, "A comparative study of agile methods, testing challenges, solutions & tool support," in *2020 14th International Conference on Open Source Systems and Technologies (ICOSST)*. IEEE, Dec. 2020, p. 1–5.

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

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

[15] J. Imtiaz, S. Sherin, M. U. khan, and M. Z. Iqbal, "A systematic literature review of test breakage prevention and repair techniques," 2019. [Online]. Available: https://arxiv.org/abs/1909.10750

[16] J. Sohn and M. Papadakis, "Using evolutionary coupling to establish relevance links between tests and code units. a case study on fault localization," 2022. [Online]. Available: https://arxiv.org/abs/2203.11343

[17] Y. Jani, "Implementing continuous integration and continuous deployment (ci/cd) in modern software development," *International Journal of Science and Research (IJSR)*, vol. 12, pp. 2984–2987, 06 2023.

[18] 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," *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1111–1122, 2023.

[19] A. S. Yaraghi, D. Holden, N. Kahani, and L. Briand, "Automated test case repair using language models," *IEEE Transactions on Software Engineering*, vol. 51, pp. 1104–1133, 2024.

[20] 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," *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 367–378, 2024.

[21] J. Chi, X. Wang, Y. Huang, L. Yu, D. Cui, J. Sun, and J. Sun, "Reaccept: Automated co-evolution of production and test code based on dynamic validation and large language models," *ISSTA*, 2025.

[22] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang, "Clarifygpt: Empowering llm-based code generation with intention clarification," 2023. [Online]. Available: https://arxiv.org/abs/2310.10996

[23] J. Li, X. Shi, K. Zhang, L. Li, G. Li, Z. Tao, J. Li, F. Liu, C. Tao, and Z. Jin, "Coderag: Supportive code retrieval on bigraph for real-world code generation," 2025. [Online]. Available: https://arxiv.org/abs/2504.10046

[24] N. Rossi, J. Lin, F. Liu, Z. Yang, T. Lee, A. Magnani, and C. Liao, "Relevance filtering for embedding-based retrieval," in *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*. ACM, Oct. 2024, p. 4828–4835. [Online]. Available: http://dx.doi.org/10.1145/3627673.3680095

[25] L. Zhang, Y. Wu, Q. Yang, and J.-Y. Nie, "Exploring the best practices of query expansion with large language models," 2024. [Online]. Available: https://arxiv.org/abs/2401.06311

[26] Z. Zeng, D. Zhang, J. Li, P. Zou, and Y. Yang, "Benchmarking the myopic trap: Positional bias in information retrieval," 2025. [Online]. Available: https://arxiv.org/abs/2505.13950

[27] H. Han, S. won Hwang, R. Samdani, and Y. He, "Convcodeworld: Benchmarking conversational code generation in reproducible feedback environments," 2025. [Online]. Available: https://arxiv.org/abs/2502.19852

[28] P. Yin, G. Neubig, M. Allamanis, M. Brockschmidt, and A. L. Gaunt, "Learning to represent edits," *ArXiv*, vol. abs/1810.13337, 2018.

[29] S. E. Robertson and H. Zaragoza, "The probabilistic relevance framework: Bm25 and beyond," *Found. Trends Inf. Retr.*, vol. 3, pp. 333–389, 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:207178704

[30] A. Schuth, F. Sietsma, S. Whiteson, and M. De Rijke, "Optimizing base rankers using clicks: a case study using bm25," in *Advances in Information Retrieval: 36th European Conference on IR Research (ECIR)*. Springer, 2014, pp. 75–87.

[31] A. Trotman, A. Puurula, and B. Burgess, "Improvements to bm25 and language models examined," in *Proceedings of the 19th Australasian Document Computing Symposium*, ser. ADCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 58–65. [Online]. Available: https://doi.org/10.1145/2682862.2682863

[32] P. Zhang, S. Xiao, Z. Liu, Z. Dou, and J.-Y. Nie, "Retrieve anything to augment large language models," 2023.

[33] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," *ArXiv*, vol. abs/2210.03629, 2022.

[34] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *ArXiv*, vol. abs/2009.10297, 2020.

[35] Y. Dong, J. Ding, X. Jiang, Z. Li, G. Li, and Z. Jin, "Codescore: Evaluating code generation by learning code execution," *ACM Transactions on Software Engineering and Methodology*, vol. 34, pp. 1 – 22, 2023.

[36] M. Evtikhiev, E. Bogomolov, Y. Sokolov, and T. Bryksin, "Out of the bleu: how should we assess quality of the code generation models?" *J. Syst. Softw.*, vol. 203, p. 111741, 2022.

[37] A. Naik, "On the limitations of embedding based methods for measuring functional correctness for code generation," *ArXiv*, vol. abs/2405.01580, 2024.

[38] 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.

[39] Z. Wang, K. Liu, G. Li, and Z. Jin, "Hits: High-coverage llm-based unit test generation via method slicing," *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1258–1268, 2024.

[40] X. Cheng, F. Sang, Y. Zhai, X. Zhang, and T. Kim, "Rug: Turbo llm for rust unit test generation," *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 2983–2995, 2025.

[41] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, Z. Wang, L. Shen, A. Wang, Y. Li, T. Su, Z. Yang, and J. Tang, "Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x," *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023.

[42] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "SWE-bench: Can language models resolve real-world github issues?" in *The Twelfth International Conference on Learning Representations*, 2024.

[43] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica, "Livecodebench: Holistic and

contamination free evaluation of large language models for code," *ArXiv*, vol. abs/2403.07974, 2024.

[44] S. A. Chowdhury, G. Uddin, and R. Holmes, "An empirical study on maintainable method size in java," 2022. [Online]. Available: https://arxiv.org/abs/2205.01842

[45] Z. Peng, T.-H. P. Chen, and J. Yang, "Revisiting test impact analysis in continuous testing from the perspective of code dependencies," *IEEE Transactions on Software Engineering*, vol. PP, pp. 1–1, 12 2020.

[46] S. Wang, M. Wen, Y. Liu, Y. Wang, and R. Wu, "Understanding and facilitating the co-evolution of production and test code," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 272–283.

[47] S. B.A, "Development of an enhanced automated software complexity measurement system," 2020.

[48] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[49] W. Takerngsaksiri, M. Fu, C. Tantithamthavorn, J. Pasuksmit, K. Chen, and M. Wu, "Code readability in the age of large language models: An industrial case study from atlassian," 2025. [Online]. Available: https://arxiv.org/abs/2501.11264

[50] T. Lee, J.-B. Lee, and H. In, "A study of different coding styles affecting code readability," *International Journal of Software Engineering and Its Applications*, vol. 7, pp. 413–422, 09 2013.

[51] R. Wang, J. Guo, C. Gao, G. Fan, C. Y. Chong, and X. Xia, "Can llms replace human evaluators? an empirical study of llm-as-a-judge in software engineering," *Proc. ACM Softw. Eng.*, vol. 2, pp. 1955–1977, 2025.

[52] W. Tong and T. Zhang, "Codejudge: Evaluating code generation with large language models," *Conference on Empirical Methods in Natural Language Processing*, vol. abs/2410.02184, 2024.

[53] A. Panickssery, S. R. Bowman, and S. Feng, "Llm evaluators recognize and favor their own generations," *Advances in Neural Information Processing Systems*, vol. abs/2404.13076, 2024.

[54] K. Wataoka, T. Takahashi, and R. Ri, "Self-preference bias in llm-as-a-judge," *ArXiv*, vol. abs/2410.21819, 2024.

[55] N. Mündler, M. N. Müller, J. He, and M. T. Vechev, "Swt-bench: Testing and validating real-world bug-fixes with code agents," in *Neural Information Processing Systems*, 2024.

[56] T. Ahmed, J. Ganhotra, R. Pan, A. Shinnar, S. Sinha, and M. Hirzel, "Otter: Generating tests from issues to validate swe patches," *ArXiv*, vol. abs/2502.05368, 2025.

[57] M. V. Pham, H. N. Phan, H. N. Phan, C. C. Le, T. N. Nguyen, and N. D. Q. Bui, "Swe-synth: Synthesizing verifiable bug-fix data to enable large language models in resolving real-world bugs," *ArXiv*, vol. abs/2504.14757, 2025.

[58] K. Wataoka, T. Takahashi, and R. Ri, "Self-preference bias in LLM-as-a-judge," in *Neurips Safe Generative AI Workshop 2024*, 2024. [Online]. Available: https://openreview.net/forum?id=tLZZZIgPJX

[59] Z. Lubsen, A. Zaidman, and M. Pinzger, "Studying co-evolution of production & test code using association rule mining," 2009.

[60] ——, "Using association rules to study the co-evolution of production & test code," *2009 6th IEEE International Working Conference on Mining Software Repositories*, pp. 151–154, 2009.

[61] J. Sohn and M. Papadakis, "Cement: On the use of evolutionary coupling between tests and code units. a case study on fault localization," *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, pp. 133–144, 2022.

[62] 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, pp. 1 – 37, 2023.

[63] M. Mirzaaghaei, F. Pastore, and M. Pezzè, "Supporting test suite evolution through test case adaptation," *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 231–240, 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:16158936

[64] S. Shimmi and M. Rahimi, "Leveraging code-test co-evolution patterns for automated test case recommendation," *2022 IEEE/ACM International Conference on Automation of Software Test (AST)*, pp. 65–76, 2022.

[65] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 75–84.

[66] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers," *ArXiv*, vol. abs/2009.05617, 2020.

[67] Z.-Q. Xie, Y.-L. Chen, C. Zhi, S. Deng, and J. Yin, "Chatunitest: a chatgpt-based automated unit test generation tool," *ArXiv*, vol. abs/2305.04764, 2023.

[68] S. Gu, Q. Zhang, K. Li, C. Fang, F. Tian, L. Zhu, J. Zhou, and Z. Chen, "Testart: Improving llm-based unit testing via co-evolution of automated generation and repair iteration," 2024.

[69] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 919–931, 2023.

[70] Z. Jiang, M. Wen, J. Cao, X. Shi, and H. Jin, "Towards understanding the effectiveness of large language models on directed test input generation," *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1408–1420, 2024.