REACCEPT: Automated Co-evolution of Production and Test Code Based on Dynamic Validation and Large Language Models

JIANLEI CHI, Xidian University, China
XIAOTIAN WANG, Harbin Engineering University, China
YUHAN HUANG, Xidian University, China
LECHEN YU, Microsoft, USA
DI CUI, Xidian University, China
JIANGUO SUN, Xidian University, China
JUN SUN, Singapore Management University, Singapore

Synchronizing production and test code, known as PT co-evolution, is critical for software quality. Given the significant manual effort involved, researchers have tried automating PT co-evolution using predefined heuristics and machine learning models. However, existing solutions are still incomplete. Most approaches only detect and flag obsolete test cases, leaving developers to manually update them. Meanwhile, existing solutions may suffer from low accuracy, especially when applied to real-world software projects.

In this paper, we propose ReAccept, a novel approach leveraging large language models (LLMs), retrieval-augmented generation (RAG), and dynamic validation to fully automate PT co-evolution with high accuracy. ReAccept employs an experience-guided approach to generate prompt templates for the identification and subsequent update processes. After updating a test case, ReAccept performs dynamic validation by checking syntax, verifying semantics, and assessing test coverage. If the validation fails, ReAccept leverages the error messages to iteratively refine the patch. To evaluate ReAccept's effectiveness, we conducted extensive experiments with a dataset of 537 Java projects and compared ReAccept's performance with several state-of-the-art methods. The evaluation results show that ReAccept achieved an update accuracy of 60.16% on the correctly identified obsolete test code, surpassing the state-of-the-art technique CEPROT by 90%. These findings demonstrate that ReAccept can effectively maintain test code, improve overall software quality, and significantly reduce maintenance effort.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**.

Additional Key Words and Phrases: Product-Test Co-evolution, Test Generation, Large Language Model, Dynamic Validation

ACM Reference Format:
Jianlei Chi, Xiaotian Wang, Yuhan Huang, Lechen Yu, Di Cui, Jianguo Sun, and Jun Sun. 2025. REACCEPT: Automated Co-evolution of Production and Test Code Based on Dynamic Validation and Large Language Models. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA055 (July 2025), 23 pages. https://doi.org/10.1145/3728930

Authors' Contact Information: Jianlei Chi, chijianlei@xidian.edu.cn, Xidian University, Xi'an, Shaanxi, China; Xiaotian Wang, wang_xiaotian@hrbeu.edu.cn, Harbin Engineering University, Harbin, Heilongjiang, China; Yuhan Huang, hyh25959@gmail.com, Xidian University, Xi'an, Shaanxi, China; Lechen Yu, yulechen@microsoft.com, Microsoft, Redmond, WA, USA; Di Cui, cuidi@xidian.edu.cn, Xidian University, Xi'an, Shaanxi, China; Jianguo Sun, jgsun@xidian.edu.cn, Xidian University, Xi'an, Shaanxi, China; Jun Sun, junsun@smu.edu.sg, Singapore Management University, Singapore, Singapore.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 2994-970X/2025/7-ARTISSTA055
https://doi.org/10.1145/3728930

1 Introduction

Software testing is an indispensable phase in the software development lifecycle. During this phase, developers scrutinize an application's output and performance using a suite of pre-defined test cases [41]. These test cases may be frequently updated along with the evolution of the application, in order to: a) keep existing test cases valid, and b) validate new features committed to the application.

Prior studies have demonstrated the importance of synchronizing the test suite with the application itself [24, 61]. Unfortunately, maintaining such consistency in practice is challenging even for experienced developers. Existing test cases (i.e., *test code*) may not be updated promptly when new changes are committed to the code base (i.e., *production code*). As a result, developers may not be capable of detecting and reproducing bugs incurred by those new changes [61]. Researchers have identified multiple factors that hinder an effective co-evolution of production and test code (PT co-evolution) [24] such as:

- manually maintaining the test suite is time-consuming [18, 47, 50];
- developers in charge of maintaining the test code may misunderstand the functionality of certain test cases [15];
- after modifying the production code, systematically identifying all affected test cases is challenging due to the complexity of software systems [9].

It is thus important to develop techniques for automatic PT co-evolution, i.e., automatically identifying and updating obsolete test cases and even introducing new test cases whenever the production code is updated. This challenge has recently attracted much research interest in the software engineering areas [24, 52, 54].

Existing approaches leverage learning techniques, such as K-Nearest Neighbors (KNN) [21], Neural Machine Translation (NMT) [26], and fine-tuned pre-trained models [17, 22], to identify implicit correlations between production code and test code. Although many approaches aim to identify obsolete test cases, they exhibit significantly low accuracy in updating those cases. According to previous evaluations on a group of obsolete test cases, CEPROT, a pre-trained model fine-tuned with 4,676 samples, achieved only an accuracy of 35.92%. Meanwhile, KNN and NMT-based approaches performed even worse, with accuracies of 7.77% and 22.33%, respectively [22]. Such low accuracies may result from insufficient training data, overfitting to specific patterns, or failing to extract general patterns. Furthermore, learning-based approaches lack mechanisms to interactively and iteratively improve accuracy during the inference phase. Although reinforcement learning can be applied to guide test repair or obsolete test case updating, it requires expert knowledge to design the model, which may be cumbersome for application developers [31].

Since 2022, Large Language Models (LLMs) have gained significant popularity across multiple research areas. These models, trained on vast amounts of data, outperform traditional deep learning models in code generation tasks [14, 19, 62]. During code generation, users can interact with an LLM and refine intermediate outputs with additional prompts. This interactive process allows users to incrementally improve the generated code, leading to more accurate and customized results. In this work, we apply LLMs, as well as, dynamic validation techniques to the PT co-evolution problem, aiming to improve the accuracy of updating obsolete test cases in an incremental and reactive manner. We propose a novel LLM-based approach, ReAccept, which stands for "REasoning-Action mechanism and Code dynamic validation assisted Co-Evolution of Production and Test code" (see Figure 2 for ReAccept's workflow). ReAccept achieved an average accuracy of 71.84% on our collected data set, significantly outperforming prior approaches. All test cases updated by ReAccept have been dynamically validated to confirm correctness in both syntax and semantics. In contrast, most prior approaches only compared the updated test cases with the expected results using certain metrics (e.g., BLEU [42] and CodeBLEU [46]). Such metrics hardly capture code semantics, which makes the overall approach ineffective.

ReAccept is capable of identifying obsolete test cases and then automatically updating them according to the production code. To improve the accuracy of PT co-evolution, ReAccept leverages ReAct [57] mechanism and Retrieval Augmentation Generation (RAG) [30] to interact with the underlying LLM automatically. ReAccept is designed to minimize the human effort to review intermediate results and steer the LLM. To gauge the quality of a LLM-generated test case, ReAccept employs a set of third-party tools, including the Java compiler [4], JUnit [5], and JaCoCo [3]. These tools dynamically validate the test case by checking syntax (Java compiler), verifying semantics (JUnit), and assessing test coverage (JaCoCo). After the dynamic validation, ReAccept creates new prompts to assist the LLM in refining the test case. This process continues until the test case passes all three validations or a predefined limit is reached.

To evaluate the effectiveness of ReAccept, we constructed a comprehensive dataset (including and extending those from previous work [22, 52, 54]), containing 537 projects with 23403 samples. We compared the performance of ReAccept with KNN, NMT, and CEPROT. The evaluation results indicate that overall, ReAccept attained an average accuracy of 71.84% on the updating tasks, significantly outperforming existing approaches. Furthermore, when configured to perform both identification and update, ReAccept achieved an average accuracy of 60.16%, surpassing KNN's 8.51%, NMT's 19.83%, and CEPROT's 31.62%. Apart from these comparisons with existing approaches, we also conducted ablation studies, revealing that choosing the right LLMs and adjusting parameters may further improve ReAccept's performance. In a nutshell, the evaluation results indicate that ReAccept can effectively automate PT co-evolution and thus reduce developers' effort to maintain the consistency between production code and test code.

In summary, we make the following contributions in this work.

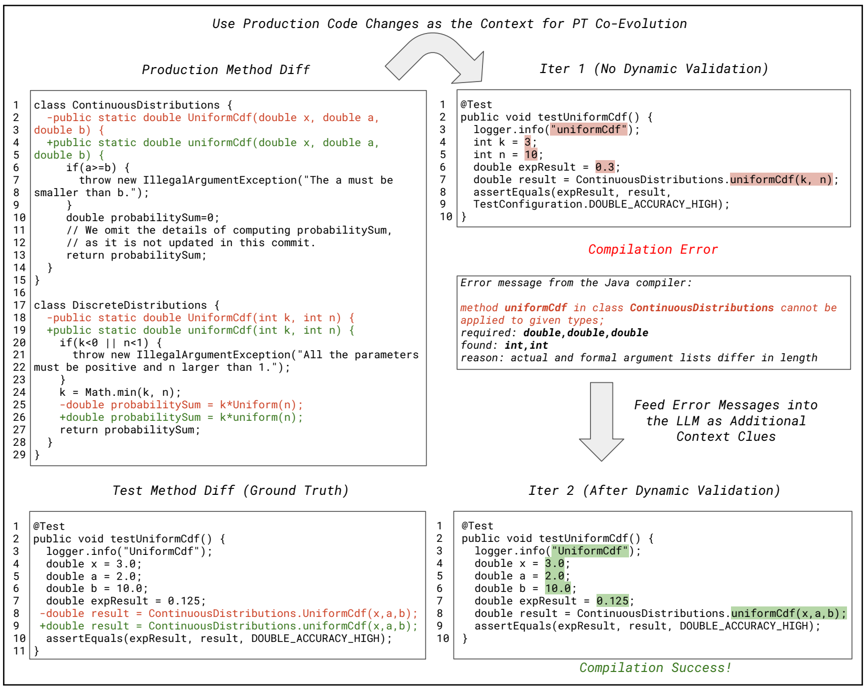
- We designed and implemented ReAccept, an automatic approach for PT co-evolution, which combines dynamic validation and LLM prompt engineering. ReAccept effectively accomplishes PT co-evolution with minimized human effort.
- We built a dataset comprising 537 projects with 23403 samples to help the community conduct research on PT co-evolution. We have made the replication package publicly available [1].
- The evaluation results show that ReAccept achieves an average accuracy of 71.84% on the updating tasks and an average accuracy of 60.16% when identifying and updating obsolete test code simultaneously, revealing a significant improvement over prior approaches.

The remainder of this paper is organized as follows. Section 2 delineates a motivation example and explains the difficulties in PT co-evolution. Section 3 illustrates the workflow of ReAccept, highlighting the details of both the identification and update phases. We introduce the conducted evaluations for ReAccept in Section 4, and compare its performance with a group of prior work. We list some related work in Section 5, discussing their correspondence with ReAccept. Finally, in Section 6, we make a summary for this paper with a few future directions.

2 Motivating Example

In the software development lifecycle, production code frequently undergoes numerous version updates, and the corresponding test code is expected to be updated promptly. Such practice is known as PT co-evolution. Failing PT co-evolution may cause obsolete test code, resulting in harmful effects such as reduced test coverage, increased maintenance costs, and undetected bugs.

Although maintaining the co-evolution of test code with production code is important, it often presents challenges in practice, especially in large software projects managed by multiple teams.


Use Production Code Changes as the Context for PT Co-Evolution
Production Method Diff
Iter 1 (No Dynamic Validation)
Compilation Error
Feed Error Messages into the LLM as Additional Context Clues
Test Method Diff (Ground Truth)
Iter 2 (After Dynamic Validation)
Compilation Success!

```
class ContinuousDistributions {
  -public static double UniformCdf(double x, double a,
double b) {
  +public static double uniformCdf(double x, double a,
double b) {
     if(a>=b) {
       throw new IllegalArgumentException("The a must be
smaller than b.");
     }
     double probabilitySum=0;
     // We omit the details of computing probabilitySum,
     // as it is not updated in this commit.
     return probabilitySum;
  }
}

class DiscreteDistributions {
  -public static double UniformCdf(int k, int n) {
  +public static double uniformCdf(int k, int n) {
    if(k<0 || n<1) {
      throw new IllegalArgumentException("All the parameters
must be positive and n larger than 1.");
    }
    k = Math.min(k, n);
    -double probabilitySum = k*Uniform(n);
    +double probabilitySum = k*uniform(n);
    return probabilitySum;
  }
}
```

```
@Test
public void testUniformCdf() {
  logger.info("uniformCdf");
  int k = 3;
  int n = 10;
  double expResult = 0.3;
  double result = ContinuousDistributions.uniformCdf(k, n);
  assertEquals(expResult, result,
  TestConfiguration.DOUBLE_ACCURACY_HIGH);
}
```

```
Error message from the Java compiler:

method uniformCdf in class ContinuousDistributions cannot be
applied to given types;
required: double,double,double
found: int,int
reason: actual and formal argument lists differ in length
```

```
@Test
public void testUniformCdf() {
  logger.info("UniformCdf");
  double x = 3.0;
  double a = 2.0;
  double b = 10.0;
  double expResult = 0.125;
 -double result = ContinuousDistributions.UniformCdf(x,a,b);
 +double result = ContinuousDistributions.uniformCdf(x,a,b);
  assertEquals(expResult, result, DOUBLE_ACCURACY_HIGH);
}
```

```
@Test
public void testUniformCdf() {
  logger.info("UniformCdf");
  double x = 3.0;
  double a = 2.0;
  double b = 10.0;
  double expResult = 0.125;
  double result = ContinuousDistributions.uniformCdf(x,a,b);
  assertEquals(expResult, result, DOUBLE_ACCURACY_HIGH);
}
```

Fig. 1. Motivation example: commit 082e6c4 in Datumbox Framework

Previous work [24] identified four factors that complicate PT co-evolution in real-world software projects:

- Maintaining a test suite is expensive. Given the limited time and resources, developers often deprioritize the co-evolution of test code.
- Developers may not be cognizant of all relevant test code related to a specific feature, resulting in the negligence of obsolete test code.
- Even experienced developers may not understand the whole project due to the complexity of modern software systems. When new production code is introduced into the project, the team maintaining the test code may not be able to update it independently.
- Manually identifying and updating test code demands extensive domain-specific knowledge, and few tools are available to help developers accomplish the co-evolution of test code.

Aware of such difficulties, we aim to develop a new methodology to facilitate PT co-evolution, ensuring that only verified and correct test code is committed to the code base, thereby greatly reducing the effort required from developers. Solely applying LLM to update obsolete test code may be impractical for real-world software projects. Since LLM may produce undesirable or even erroneous output, developers must continuously validate the generated test code. To relieve developers from such cumbersome tasks, we propose an approach to help LLM interact with third-party dynamic validation tools automatically, guaranteeing the quality of generated test code.

To illustrate the necessity of dynamic validation when updating test code, we exhibit an example from the datumbox-framework project [7] in Figure 1. The example involves two classes, `ContinuousDistributions` and `DiscreteDistributions`, each defining a method `UniformCdf`. The former takes three double parameters as input, whereas the latter accepts two integer parameters. In a specific commit, the developer refactored both methods, renaming them from `UniformCdf`

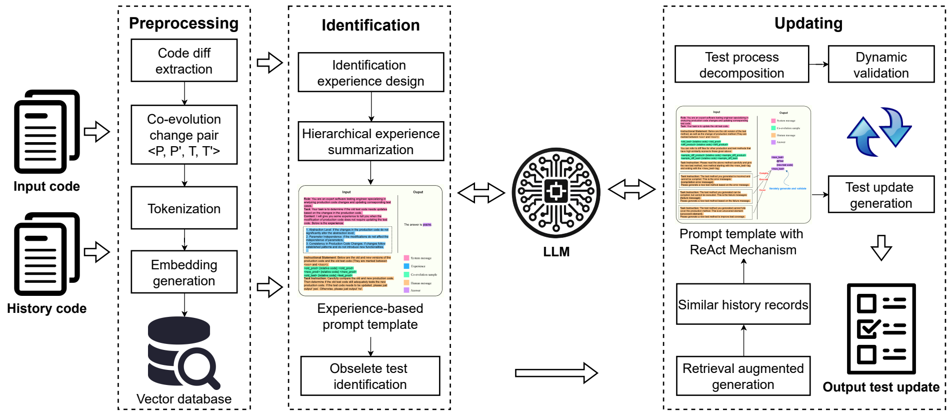

Preprocessing
Code diff extraction
Co-evolution change pair <P, P', T, T'>
Tokenization
Embedding generation
Vector database
Input code
History code
Identification
Identification experience design
Hierarchical experience summarization
Experience-based prompt template
Obselete test identification
LLM
Updating
Test process decomposition
Dynamic validation
Test update generation
Prompt template with ReAct Mechanism
Similar history records
Retrieval augmented generation
Output test update

Fig. 2. Overview of ReAccept's workflow

to `uniformCdf`, and also renamed the utility function `Uniform` to `uniform`. When the production code changes were presented as context to a LLM (e.g., ChatGPT), the model failed to differentiate between the two distinct `uniformCdf` methods. Consequently, it generated an incorrect patch for the test code that mistakenly invokes `ContinuousDistributions.uniformCdf` with two integer arguments, resulting in a compilation error in the test case `testUniformCdf`. This incorrect test case cannot be directly integrated into the repository without further corrections. Therefore, we provided the error messages to the LLM and requested it to retry. By analyzing these error messages, the LLM revised its output and generated a valid test case, successfully resolving the discrepancies between the two `uniformCdf` methods.

3 Our Method

In this section, we introduce the workflow of ReAccept and describe the details of its key components, including preprocessing, identifying obsolete test code, and applying the necessary updates to the code base.

3.1 Overall Workflow

The ultimate of ReAccept is to automate the PT co-evolution completely, i.e., automatically identify and update obsolete test code. While numerous novel approaches have been proposed in recent years [13, 22, 28, 32–34, 48, 51, 52, 54], most focus primarily on the identification problem, with the task of updating outdated test code remaining cumbersome. In addition, limited research targeting automatic test code updates suffers from low accuracy, rendering them impractical for real-world software [22].

Given LLM's impressive capability to process and understand structured text [11, 12, 20, 27, 29, 37–39, 59, 63], ReAccept relies on LLMs and Retrieval Augmentation Generation (RAG), together with dynamic validation, to identify and update obsolete test code precisely. In Figure 2, we illustrate the overall workflow of ReAccept, which consists of three phases: preprocessing, identification, and updating. The preprocessing phase scrutinizes each project's commit history while extracting per-method code diffs between adjacent commits. These code diffs are stored in a local vector knowledge base, facilitating prompt construction for subsequent phases. In the identification phase, ReAccept relies on an LLM with pre-defined hierarchical experience to determine whether the input code demands updating. Finally, in the update phase, ReAccept combines LLM prompting, dynamic validation, and RAG techniques to fix detected obsolete test cases with high accuracy.

3.2 Preprocessing

At this phase, a collection of code commits from real-world projects is organized into structured data, and code change pairs are extracted to construct a local vector knowledge base.

3.2.1 Data Collection. We collect samples from 537 Java projects of the Apache Foundation, as well as popular Java projects from GitHub. All of these projects are managed by Maven, with well-organized repository structures and clear naming conventions. These features make it easier to mine production and test code based on naming conventions.

ReAccept represents each per-method code diff as a *Change Pair* (CP), comprising four components: *group* (the project team), *project* (the project name), $change_p$ (changes in the production code), and $change_t$ (changes in the test code).

$$CP = (group, project, change_p, change_t) \tag{1}$$

$change_p$ and $change_t$ are both 5-tuples of metadata.

$$change_p / change_t = (version, module, package, class, type) \tag{2}$$

These tuples represent the commit hash ID (i.e., SHA1 in Git), the module name, the package name, the class name, and the type of change applied to a single file (either *CREATE*, *DELETE*, or *EDIT*), respectively. The values of *group*, *project*, and *version* together form a unique identifier for the project, while "$[module]/[package].[class]$" specifies the relative path of the source file in the project.

3.2.2 Knowledge Base Construction. For each code diff, ReAccept systematically mines the implicit correlation between:

- the original method, $prod_{old}$, and its updated version, $prod_{new}$,
- the original test case, $test_{old}$, and its co-evolved version, $test_{new}$.

According to prior studies [52, 54], the most common combination of change types is *EDIT-EDIT* (EE), where developers modify both the production and test code. Therefore, $change_p$ and $change_t$ are commonly set to *EDIT* in most CPs.

ReAccept employs RAG techniques to retrieve pertinent knowledge from the knowledge base, utilizing a few relevant samples (few-shot) to guide the identification and updating phases. RAG enables AI applications to transition from the previous pre-training+fine-tuning mode to the pre-training+prompt mode, greatly simplifying the workload of training models for different demands. A crucial task in constructing the knowledge base is representing code changes. We aim to reduce redundant information in code changes, shorten prompt words, and improve the performance of LLMs on long sequences. If sequence editing is used, it is necessary to explain the meaning of different operations in the editing script to LLMs, which complicates the understanding process. Consequently, we differentiate code changes and use code diffs instead of editing scripts [16] as input. To achieve better retrieval results, these diffs are segmented, embedded, and stored in a vector database during the knowledge base construction.

Tokenization: For each diff file of the production code, ReAccept leverages LangChain's code segmentation tool [6] to partition it into token sequences. During the tokenization, changes to keywords are separated from those to identifiers and converted into distinct tokens, enabling the LLM to learn the code diff characteristics more efficiently. ReAccept employs a block size of 50 with no block overlap. Additionally, the segmentation tool does not differentiate between code and comments. Source codes and comments were combined without separating comments from codes. We do not separate codes and comments for the following reasons:

- comments in the code also provide useful description for corresponding code changes;

- refactoring commits commonly revise a large number of comments, making the description consistent with the code changes.

Code Embedding: After tokenization, ReAccept vectorizes these token sequences and uses LangChain's code embedding module to embed such sequences. For text embedding, ReAccept applies OpenAI's text embedding model, ada-002 [8]. This vector embedding process captures the semantic features of code elements, transforming them into high-dimensional vector representations. These vectors will then be utilized to find knowledge base samples that closely match the target production code changes.

Vector Storage: The vector data, obtained from code embedding and collaborative modification code pairs, is stored in the vector database Chroma [2] as a dictionary. That is, it is used as an external expert knowledge base. The constructed vector database supports efficient similarity search and retrieval. It is subsequently used for extracting the most similar sample and guiding the LLMs for code generation.

Knowledge Retrieval: When a software project receives a change, ReAccept retrieves the most similar sample from the database. Then, ReAccept applies this change to the production code and outputs the diff file. The input production code will be transformed into a vector representation. ReAccept relies on *cosine similarity* for sample comparison, which considers the angle between the vectors generated by code changes, and regularizes the code length. The usage of cosine similarity can avoid distance bias due to different code lengths. Compared to Euclidean distance, cosine similarity can better capture the semantic information of code changes and thus reflect the semantic similarity of code changes. The cosine similarity is calculated using the following formula.

$$\boldsymbol{c} \cdot \boldsymbol{s} = \|\boldsymbol{c}\| \|\boldsymbol{s}\| \cos\theta \tag{3}$$

$$\text{similarity} = \cos\theta = \frac{\boldsymbol{c} \cdot \boldsymbol{s}}{\|\boldsymbol{c}\| \|\boldsymbol{s}\|} = \frac{\sum_{i=1}^{n} c_i s_i}{\sqrt{\sum_{i=1}^{n} c_i^2}\sqrt{\sum_{i=1}^{n} s_i^2}} \tag{4}$$

Given the production code change $\boldsymbol{c}$ and the high similarity sample $\boldsymbol{s}$, the cosine similarity is calculated by the dot product and the vector length.

3.3 Identification of Obsolete Test Code

In this phase, the preprocessed code change pairs will be sent to ReAccept's identifier to determine whether an obsolete test update is needed. Let p and t denote the production code and the test code before the update, with p' and t' after the update, respectively. The task of obsolete test identification can be formulated as a function Identify(p, p', t) such that

$$\text{Identify}(p, p', t) = \begin{cases} 1 & \text{if } t \neq t' \text{ i.e., } t \text{ must be updated.} \\ 0 & otherwise \end{cases} \tag{5}$$

Identifying obsolete tests can be regarded as a specialized text classification task, which ReAccept addresses through prompt engineering and RAG techniques.

When the production code undergoes changes and a repair patch is generated, the production code pair $< p, p' >$ and the original test code t are fed into the LLM. The model then attempts to understand and capture the semantic relationships in the code modifications, thereby determining the necessity of updating t. If the update is indispensable, ReAccept compares and searches for the most similar update record in the vector knowledge base, to guide the subsequent obsolete test update.

The overall process of obsolete test identification is made up of four steps, including *identification experience design*, *experience summarization*, *prompt template design* and *identification result generation.*

3.3.1 Identification Experience Design. In a preliminary experiment, we directly ask the LLM to identify the obsolete test code only based on $prod_{new}$, $prod_{old}$ and $test_{old}$. The performance proves to be unsatisfactory. The LLM exhibits a cognitive bias when analyzing the test code, presuming that any modification to the production code would render the correlated test cases obsolete. There might be two probable reasons.

- **Data and corpus deviation**. LLMs are typically trained on vast amounts of text data, which often includes extensive discussions about code updates and maintenance. In many real-world development scenarios, modifications to the production code frequently accompany updates to the test code. This pattern may dominate a significant portion of the training data, causing the model to develop such a bias.
- **Lack of domain-specific knowledge**. Despite possessing a broad spectrum of general knowledge, LLMs may lack expertise and experience in specific domains. There is a tendency to rely on general knowledge, potentially overlooking the nuance between real-world projects.

To address this issue, we propose an experience-guided approach to construct more effective prompts, helping the LLMs more accurately determine whether modifications to production code require updates to test code. Algorithm 1 introduces the details of the experienced-guided approach. The term "experience" refers to a set of natural-language descriptions of domain-specific principles in PT co-evolutions, denoted as $E = \{e_1, e_2, ...e_m\}$, where each e_i consists of a principle's title and a detailed explanation.

For instance, e_i could be "Abstraction Level: if the changes in the production code do not significantly alter the abstraction level". The set of experience E will be incorporated into the prompt, guiding the LLM in more accurately determining whether modifications to the production code necessitate updates to the test code.

Algorithm 1 Identification Experience Extraction Process

```
1:  Input: Training dataset $D = \{d_1, \ldots, d_n\}$(code changes and update labels)
2:  Output: A set of refined experience $E = \{e_1, e_2, \ldots, e_m\}$.
3:  Step 1: Experience Extraction
4:  for each sample $d_i$ in $D$ do
5:      Extract experience $e_i$ from $d_i$ by identifying key patterns of code changes and updates.
6:      Add $e_i$ to the initial set of experience $E_{init}$.
7:  Step 2: Experience Embedding and Aggregation
8:  Vectorization: Use a pre-trained embedding model to vectorize experience in $E_{init}$.
9:  PCA: Apply PCA to reduce the dimensions of the embeddings.
10: Clustering: Apply mini-batch k-means clustering to group similar experience.
11: Step 3: Clusters Summarization
12: for each cluster $C_i$ in $\{C_1, \ldots, C_n\}$ do
13:     Split $C_i$ into chunks.
14:     Summarize each chunk $Ch_i$ by using an LLM to get chunk-level summarization $s_i$.
15:     Add $s_i$ into relevant cluster-level summarization $S_c$.
16:     Add experience $S_c$ into the set $E_{sum}$.
17: Step 4: Semantic Deduplication
18: Semantic Deduplication: Remove redundant $S_c$ in $E_{sum}$.
19: Output: Refined experience set $E$
```

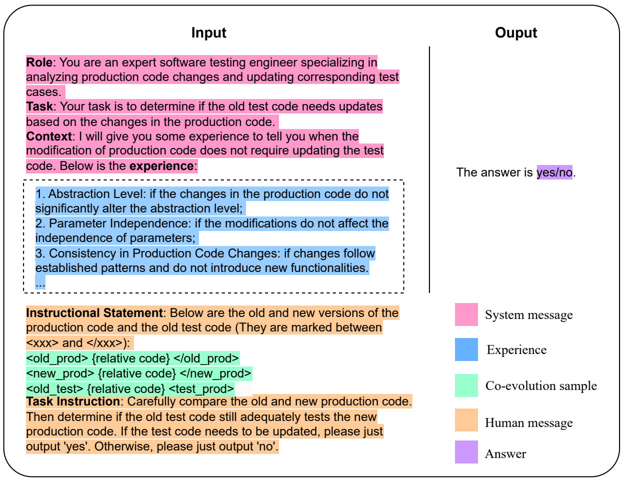
3.3.2 Experience Summarization. We build the experience set in four steps, as outlined in Algorithm 1. In lines 3-6, each training sample d_i from dataset D is refined into an experience piece e_i, preserving only key patterns in code changes and their corresponding test updates. These patterns include syntactic modifications, dependency shifts, and semantic alignments between the production code and test code. By aggregating extracted experience, we get an initial set E_{init}.

In lines 7-10 of Algorithm 1, we undergo a refinement process through embedding and aggregation. First, we employ a pre-trained embedding model to convert each experience $e_i \in E_{init}$ into a high-dimensional vector. This vector captures semantic and syntactic nuances of code changes and test updates. To mitigate dimensionality challenges, we apply Principal Component Analysis (PCA) to the embeddings, retaining most of the variance while compressing features into a lower-dimensional space. Subsequently, we cluster these reduced embeddings using mini-batch k-means, which efficiently groups a large amount of similar experience into the same category.

In lines 11-16 of Algorithm 1, we refine each cluster C_i into concise principles. Large clusters are split into smaller chunks Ch_i to accommodate the context length limitation in LLMs. For each chunk Ch_i, we prompt the LLM with a task-specific template: "summarize experience into key principles in a structured format: $\{num\}.\{AbstractPrinciple\} : \{Explanation\}$". This template produces chunk-level summarization s_i, which are further refined into the cluster-level experience S_c within the set E_{sum}.

Finally, in lines 17-18 of Algorithm 1, we perform semantic deduplication on E_{sum}. Using cosine similarity between embeddings of all summary pairs, we merge or remove redundancies where similarity exceeds the threshold. This yields the refined experience set E, which helps mitigate model hallucinations and thus improves recognition accuracy.

3.3.3 Prompt Template Design. Prompt engineering plays a crucial role in identifying obsolete test code. By carefully designing the prompts, the LLM can make more accurate judgments. Figure 3 illustrates an example of prompt template designed for the obsolete test identification task. The prompt template is primarily composed of two parts: *system message* and *human message*. The system message defines the role the LLM should assume and its general behavior. The human message, also known as the user message, includes prompts from the user and may provide examples, historical prompts, or specific instructions for the assistant. The generalized experience retrieved through Algorithm 1 is also included in the human message.



Role: You are an expert software testing engineer specializing in analyzing production code changes and updating corresponding test cases.
Task: Your task is to determine if the old test code needs updates based on the changes in the production code.
Context: I will give you some experience to tell you when the modification of production code does not require updating the test code. Below is the **experience**:

1. Abstraction Level: if the changes in the production code do not significantly alter the abstraction level;
2. Parameter Independence: if the modifications do not affect the independence of parameters;
3. Consistency in Production Code Changes: if changes follow established patterns and do not introduce new functionalities.
...

Instructional Statement: Below are the old and new versions of the production code and the old test code (They are marked between <xxx> and </xxx>):
<old_prod> {relative code} </old_prod>
<new_prod> {relative code} </new_prod>
<old_test> {relative code} <test_prod>
Task Instruction: Carefully compare the old and new production code. Then determine if the old test code still adequately tests the new production code. If the test code needs to be updated, please just output 'yes'. Otherwise, please just output 'no'.

The answer is yes/no.

Fig. 3. The prompt template of the identification task

3.3.4 Identification Result Generation. After the identification experience extraction process, our prompt design can now accurately guide the LLM in identifying the obsolete test code. During the process of generating identification results, the LLM analyzes the given production and test code based on the experience and guidance in the keywords, and provides a judgment on whether

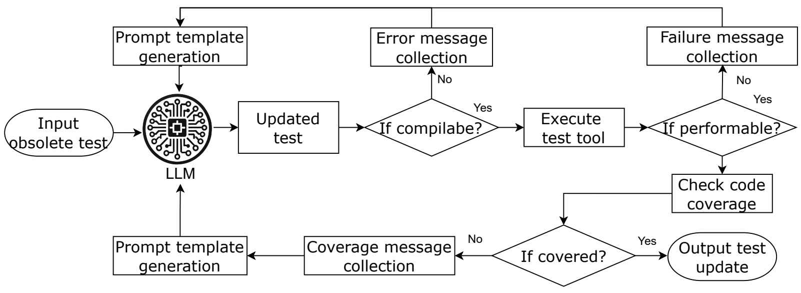


Fig. 4. Test process decomposition

the test code needs to be updated. Additionally, the LLM will offer detailed explanations to help developers understand the reasoning behind its judgments.

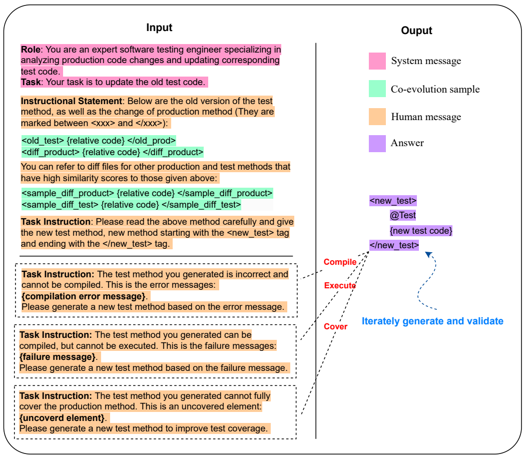
3.4 Obsolete Test Code Update

In this phase, ReAccept updates the detected obsolete test code. The task of obsolete test update can be described as a function $\text{Update}(p, p', t) = t'$ which returns a test t' that is compilable, testable, and covering updated code in p. In our approach, ReAccept solves this task in two steps: *test update generation* and *test update validation*. Although ReAccept focuses on Java applications, the techniques behind ReAccept are applicable to other programming languages.

3.4.1 The Decomposition of Obsolete Test Update. As illustrated in Figure 4, the identified obsolete test code will be sent to ReAccept's updater. The updater combines the obsolete test code with the retrieved most similar historical records to form a few-shot prompt, thereby generating the new test code through the LLM. Using dynamic validation, the updater adjusts the output from the LLM to make sure the updated test code satisfies some pre-defined requirements. First, the test code must be correctly compiled by the Javac compiler [4]. In addition, ReAccept reruns the test code and acquires their coverage to the production code through JaCoCo [3]. As a valid update, the generated test code must cover the modified statements in the production code while not arising any runtime errors [25].

The decomposition of the update phase facilitates the construction of prompt templates for different scenarios, assisting the LLM in fixing erroneous snippets in the test code. Due to the inherent variability of testing effects, ReAccept measures testing code quality using four levels: *compilation failure*, *test failure*, *coverage failure*, and *satisfying all testing requirements*. This also requires ReAccept to parse the results of the dynamic validation and interact with the LLM for future test code generation.

3.4.2 Test Update Generation. The prompt templates play a pivotal role


Input

Role: You are an expert software testing engineer specializing in analyzing production code changes and updating corresponding test code.
Task: Your task is to update the old test code.

Instructional Statement: Below are the old version of the test method, as well as the change of production method (They are marked between <xxx> and </xxx>):

<old_test> {relative code} </old_prod>
<diff_product> {relative code} </diff_product>

You can refer to diff files for other production and test methods that have high similarity scores to those given above:

<sample_diff_product> {relative code} </sample_diff_product>
<sample_diff_test> {relative code} </sample_diff_test>

Task Instruction: Please read the above method carefully and give the new test method, new method starting with the <new_test> tag and ending with the </new_test> tag.

Task Instruction: The test method you generated is incorrect and cannot be compiled. This is the error messages:
{compilation error message}.
Please generate a new test method based on the error message.

Task Instruction: The test method you generated can be compiled, but cannot be executed. This is the failure messages:
{failure message}.
Please generate a new test method based on the failure message.

Task Instruction: The test method you generated cannot fully cover the production method. This is an uncovered element:
{uncoverd element}.
Please generate a new test method to improve test coverage.

Ouput

🟥	System message
🟩	Co-evolution sample
🟧	Human message
🟪	Answer

```
<new_test>
    @Test
    {new test code}
</new_test>
```

Fig. 5. The prompt template of the updating task

Algorithm 2 Generate and Validate Updated Test Code

```
Input: Diff for production code: $prod_{diff}$, obsolete test code: $test_{old}$
Output: Updated test code: $test_{gen}$
    procedure GENERATE($prod_{diff}$, $test_{old}$)
2:      Retrieve $sample$ from knowledge base, build $prompt$ with $prod_{diff}, test_{old}, sample$
        Get $test_{gen}$ from LLM with input $prompt$, update $test_{old}$ in environment with $test_{gen}$
4:      Compile and get $result, messages$
        if $result = SUCCESS$ then
6:          Unit test and get $result, messages$
            if $result = PASS$ then
8:              Coverage test and get $result, messages$
        while $result \neq COVER$ do
10:         Build $prompt$ with $messages$
            Get $test_{gen}$ from LLM with input $prompt$, update $test_{last}$ in environment with $test_{gen}$
12:         Compile and get $result, messages$
            if $result = SUCCESS$ then
14:             Unit test and get $result, messages$
                if $result = PASS$ then
16:                 Coverage test and get $result, messages$
        return $test_{gen}$
```

in guiding update tasks. It is also nec-
essary to optimize the prompt template to ensure that the LLM outputs well-formatted text [43].

As has been shown in Figure 5, the input parameters accepted by the prompt template for updating tasks are production code before and after the change ($prod_{old}$, $prod_{new}$), the original test code $test_{old}$, and retrieved samples with highest similarity ($prod_{samlpe}$, $test_{sample}$). We choose to embed a sample's code diff into the prompt templates rather than directly entering all related production and test code to the LLM. Since the diff file concentrates on the code changes, it is instrumental in compressing the prompt length and avoiding potential performance downgrade when the LLM encounters a lengthy prompt.

If the test code does not meet the testing requirements, ReAccept uses the test results obtained from feedback information. Figure 5 illustrates the three types of feedback templates we construct. These templates will be integrated into the human message to build a new prompt, which will then be input into the LLM.

3.4.3 *Applying ReAct Mechanism.*

After generating prompt templates, ReAccept initializes the LLM with a conversation memory, to build an agent following the ReAct mechanism [57].

The agent generates inference paths and specific task behaviors in an interleaved manner. The inference paths facilitate the deduction of behavior plans and the handling of corner cases. Meanwhile, the task behaviors allow the agent to access external sources, such as the knowledge base to obtain knowledge related to the current task, i.e., high-similarity samples. The ReAct mechanism can coordinate LLM to obtain testing information from the actual project environment, which is used to guide LLM to correct compilation errors, testing failures, or low coverage of testing code.

3.4.4 *Test Update Validation.*

After constructing the agent, the change patterns will be input into the agent to predict the updated test code. We will also retrieve the most similar sample from the knowledge base to construct a prompt. This prompt will then be input into the LLM, allowing us to extract the updated test code from the LLM's output. The updated test code will be used to test the modified production code in the environment, generating test results and messages.

ReAccept will revise and dynamically validate the output code through the following process. All error messages encountered during dynamic validation will be analyzed to inform future test code generation.

(1) Update the test code in the original project environment.
(2) Compile the test code to check if the test class file is compilable.
(3) Rerun the test code, examining whether the execution raises any failures or runtime errors.
(4) Apply JaCoCo to check the coverage at the statement level. The updated test code should cover the changes revealed in diffs of the production code.

The dynamic validation process will keep executing the process shown in Figure 4 and Algorithm 2 until generating a valid test code or reaching a pre-defined cutoff (currently, we empirically set the maximum rounds of iteration to 8). When the test code passes the checks of compilation, re-execution, and test coverage, ReAccept will stop further interaction with the LLM and output the updated test code.

4 Experiment

Our method ReAccept has been implemented using the LangChain framework, with OpenAI's gpt-4-0125-preview as the underlying LLM. The LLM's default settings for PT co-evolution are: temperature set to 0, top-p sampling with $p = 1$, frequency penalty set to 0, and a conversation buffer enabled with a window size of 3.

We chose to use these default settings based on the following insights:

- Lowering the temperature helps improve the stability of the LLM output. We observed that with the temperature set to 0, the LLM output for updating test code exhibited less variability.
- When the top-p value is low, LLM reserves fewer candidates for output, which may result in the loss of necessary keywords or identifiers in the test code. Therefore, a higher top-p value is required to reserve enough candidate tokens.
- We evaluated two context management strategies available in LangChain: conversation-buffer-window and conversation-token-buffer. The former retains the last few conversation turns, while the latter maintains only the most recent messages within a specified token limit. Applying the conversation-buffer-window strategy avoids a conversation being truncated and helps us monitor the LLM's performance in different iterations.
- Setting the window size to 3 ensures that all input remains within the LLM's token limit. We found that with a window size of 4, 5, or 6, some samples in the dataset exceeded the LLM's input capacity.

In this section, we design multiple experiments to evaluate the effectiveness of ReAccept. We start by outlining the research questions that guided the evaluation design, followed by a description of the experimental setup and an in-depth discussion of the results.

4.1 Research Questions

We aim to answer the following research questions:

- **RQ1:** How effective is ReAccept on solving the PT co-evolution problem, i.e., identify and update obsolete test code?
- **RQ2:** How effective is ReAccept on identifying obsolete tests?
- **RQ3:** How effective is ReAccept on updating obsolete tests?
- **RQ4:** What are the factors that can impact the performance of ReAccept?

Table 1. Data Set Details

Data Set	Pos.	Neg.	Total
Train	4676	16496	21172
Test	520	1771	2231
Total	5196	18267	23403

Table 2. Benchmark Details for Dynamic Evaluation

Project	Commits	Samples
springside4	2/2	3/3
commons-lang	5/5	8/8
dddlib	2/4	8/10
datumbox	7/7	9/9
openmrs-core	28/34	29/37
basex	30/42	46/63
Total	74/94	103/130

RQ1 aims to provide an overall evaluation of our method, treating the identification and update phases as two parts of a holistic process. RQ2 and RQ3 then analyze the performance of our method in each phase separately. At last, RQ4 conducts multiple ablation studies to evaluate the impact of different design choices and parameters on ReAccept's performance.

4.2 Experiment Settings

4.2.1 Dataset Construction. In order to build a comprehensive PT co-evolution benchmark dataset, we expanded the datasets from previous work [22, 52, 54] to facilitate our evaluations and future research on PT co-evolution. The dataset primarily consists of Java projects from the Apache Foundation, along with the top 1000 highly-starred Java projects on GitHub. All of these projects are managed using Maven and follow a good structure and naming convention, making it easier to mine paired production and test code based on these conventions. In total, the dataset includes 537 projects. We selected CPs from these projects according to the change types in the production and test code. We only included CPs that edit both production and test code (e.g., `Edit`), while excluding other change types (e.g., `Create` and `Delete`). The detailed information of the dataset is shown in Table 1.

We split the dataset into a 90% training set for the LLMs to gain experience and a 10% test set to assess the effectiveness of our PT co-evolution approach. The co-evolution samples in the dataset were labeled as either positive or negative. Positive samples indicate that the corresponding test code requires updating, while negative samples do not require any changes. Consequently, only the positive samples from the training set were included in the knowledge base, which can be retrieved using the RAG technique.

Note that ReAccept conducts dynamic validation in the test code updating phase, which requires setting up the correct test environment (e.g., JDK, third-party packages) for all samples. Thus, to evaluate the effectiveness of updating obsolete test code, we scrutinized six popular Java projects from our dataset that are managed by Maven, examining the test environment of each collected commit. In order to make an apple-to-apple comparison, we only report the results on these six projects, which prior work can also build and produce valid results for.

Details of these projects are presented in Table 2. The first two columns record each project's name and the number of collected commits, respectively. The third column is the number of collected samples, indicating that a single commit may contain multiple samples. The number of commits and samples are expressed as ratios, with the numerator representing the count of successfully built and runnable instances. Overall, our evaluations utilized 74 commits, which included 103 co-evolution samples.

4.2.2 Baselines. For baseline comparisons, we compared ReAccept with general methods such as Random Guess (RG), K-Nearest Neighbor (KNN), Neural Machine Translation (NMT), as well as state-of-the-art approaches including SITAR [54], CHOSEN [52], and CEPROT [22]. Since RG, SITAR, and CHOSEN are only capable of identifying the obsolete test code (RQ2), we omitted them when comparing the overall performance (RQ1) and the effectiveness of updating the obsolete test code (RQ3).

4.2.3 Evaluation Metrics. When evaluating obsolete test code identification, we measured each approach's accuracy, precision, recall, and F1 score on our dataset. We selected these metrics because the task can be modeled as a binary classification problem, and these metrics are commonly used to assess the performance of classification methods.

To evaluate the effect of updating obsolete test code, we need to compare the generated code, by different approaches, with the ground truth. We applied three distinct categories of metrics for the comparison, including:

- Syntactic metrics: We used Levenshtein Distance to evaluate the syntactic performance of updated test code.
- Semantic metrics: We used Statement Coverage to evaluate the semantic performence of the updated test code.
- Comprehensive metrics: We adopted CodeBLEU [46] as the evaluation metric.

As an extension to BLEU [42], CodeBLEU is a commonly used metric for evaluating code generation tasks. Prior work [46] has revealed that BLEU does not take into account the correctness of programming language syntax and logic, likely to result in syntax or logic errors with high n-gram accuracy. Therefore, CodeBLEU is more suitable for our evaluations.

However, CodeBLEU still cannot fully reflect the effectiveness of the updated test code in practice. In [22], Hu *et al.* show that even test code with a high CodeBLEU score may fail to execute correctly in real projects. The compilation success rate is only 48%, and the actual correctness rate of the update is only 12.3%. To address the problem, we further complemented CodeBLEU with the following three metrics measured based on dynamic evaluation, when comparing ReAccept with those baseline approaches.

- Compile Success Rate (CSR): the percentage of test code that is successfully compiled (after building the corresponding project version).
- Test Pass Rate (TPS): the percentage of updated test code that successfully runs and passes.
- Update Coverage Rate (UCR): the percentage of updated test code that successfully runs, passes, and covers the changes in the product code.

Note that for all rates, higher values indicate better performance.

At last, to evaluate the overall effectiveness of each technique for PT co-evolution, we measured the percentage of correctly identified test code that successfully compile, run, and cover the updated production code.

4.3 Experimental Results

4.3.1 RQ1: The effectiveness on solving the PT co-evolution problem. We conducted experiments on 130 samples from Table 2 to evaluate the overall effectiveness of different approaches. These 130 samples can be categorized into two types: configurable samples, which consist of successfully generated code validated by the environment, and configuration-failed samples, which fail to compile due to missing dependencies or syntax errors. During the experiment all samples identified as positive (requiring an update) entered ReAccept's repair phase. When calculating the joint accuracy of the identification and update phases, we applied the formula $Acc = TPC/TP$. TP are the total number of true positive samples which ReAccept correctly identified as requiring an update. TPC refers to the number of samples where the generated code is successfully compiled and functionally validated.

Figure 6 shows the results when ReAccept and baseline approaches aim to solve the PT co-evolution problem completely, i.e., identifying and updating the tests automatically. We can observe that ReAccept outperforms all baseline approaches, showing significant improvements across all metrics. ReAccept achieves an update accuracy of 60.16% in correctly identifying the obsolete test code, surpassing KNN's 8.51%, NMT's 19.83%, and CEPROT's 31.62% by 607%, 203%, and 90%, respectively. In terms of syntactic and semantic performance, ReAccept is also more outstanding than the baseline method. The Levenshtein distance of ReAccept is 240, obviously less than CEPROT's 312, KNN's 524 and NMT's 610, which means it can achieve better performance than other methods with fewer edits. ReAccept has a statement coverage of 41.1%, which is 20%, 119% and 154% higher than CEPROT's 34.2%, KNN's 18.8% and NMT's 16.2%. The dynamic validation of ReAccept ensures the quality of the test code, resulting in better statement coverage than other methods. Regarding the CodeBLUE metric, ReAccept also outperforms other approaches with a score of 82.03%. This yields improvements of 118.7%, 154.4%, and 30.4% over KNN's 37.63%, NMT's 32.35%, and CEPROT's 63.11%, respectively.

Specifically, in the identification phase, ReAccept identifies more obsolete test code compared to KNN, NMT, and CHOSEN classifiers. Although CEPROT surpasses ReAccept in the obsolete test identification phase, ReAccept demonstrates significantly better performance in the obsolete test update phase, ultimately securing a notable overall advantage. Note that both phases of ReAccept are flexible and independent, allowing for the integration of more effective obsolete test identification techniques to further enhance ReAccept's performance. Overall, the results suggest that ReAccept is a useful tool that can significantly reduce the required effort for the PT co-evolution problem.

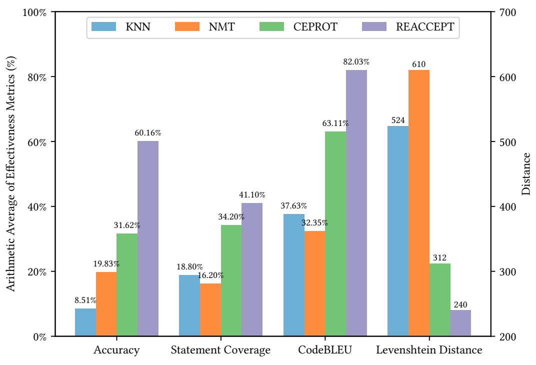


Fig. 6. The overall effectiveness of different approaches

4.3.2 *RQ2: The effectiveness of the obsolete test identification.* The above results suggest that ReAccept is the best approach overall. Next, we evaluated the performance of ReAccept in the two phases separately and compared it to the corresponding state-of-the-art approaches. Table 3 provides detailed statistics on ReAccept's performance during the obsolete test identification phase, including True Positives, False Positives, True Negatives, and False Negatives. Next, Table 4 highlights the performance difference between ReAccept and other baseline approaches (RG, KNN, SITAR, NMT, CHOSEN, and CEPROT) using those predefined metrics, such as precision, recall, and F1-score, for both positive and negative samples. We carried out experiments with 3-fold cross-validation and calculated the average of the metrics as the experimental result to demonstrate the reliability of ReAccept. Note that accuracy reflects the overall correctness of an approach's predictions, irrespective of class labels, and is therefore reported as a single value in Table 4.

It can be observed that ReAccept significantly outperforms RG, KNN, NMT, and SITAR, marginally outperforms CHOSEN, and shows comparable performance to CEPROT. Notably, ReAccept excels in the negative class, with a precision of 99.00%, recall of 97.92%, and F1 score of 98.46%. We found that misidentification mainly results from a lack of comprehensive experience covering all complex scenarios. Improving the quantity and quality of the training set can help mitigate such misidentification. Additionally, utilizing more advanced LLM models could further enhance ReAccept's

Table 3. ReAccept's identification result of 2231 Samples

Identification Labe	Prediction		Total
	Pos.	Neg.	
Positive	496	24	520
Negative	53	1658	1711
Total	549	1682	2231

Table 4. Effectiveness of different techniques on the obsolete test identification task

Method	Acc.	Positive			Negative		
		Prec.	Rec.	F1.	Prec.	Rec.	F1.
RG	48.68%	46.39%	49.72%	48.00%	51.08%	47.74%	49.35%
KNN	74.73%	83.40%	72.30%	77.50%	90.56%	75.41%	82.29%
SITAR	84.17%	78.30%	38.90%	52.00%	84.83%	96.89%	90.38%
NMT	91.50%	82.00%	78.85%	80.39%	94.05%	95.09%	94.51%
CHOSEN	92.89%	89.29%	96.69%	92.84%	96.74%	89.45%	92.95%
CEPROT	97.50%	98.30%	90.00%	94.00%	97.20%	99.60%	98.40%
ReAccept	97.51%	91.69%	95.98%	93.79%	99.00%	97.92%	98.46%

identification performance. We would like to point out that, unlike ReAccept, which is designed to fully address the PT co-evolution problem, some approaches (such as SITAR and CHOSEN) only partially address the issue and can not automatically update the test code. Extending these approaches to incorporate update functionality is also cumbersome due to their complex software infrastructure. Furthermore, while CEPROT performs well in identifying the obsolete test code, it turns out to be ineffective in updating those test cases, as we will show in the following research question. Overall, the results show that ReAccept can effectively identify the obsolete test code.

4.3.3 RQ3: The effectiveness of the obsolete test update. Apart from the identification phase, we also evaluated ReAccept's performance in terms of updating the identified obsolete test code. The corresponding evaluation results are shown in Table 5. We can observe that ReAccept significantly outperforms all the baselines, in terms of all three metrics. The CSR of ReAccept is 85.44%, which is substantially higher than the best baseline's 49.52%. Furthermore, the updated test code generated by ReAccept have a 71.84% chance of covering the updated product code, considerably higher than the best baseline's 35.92%. These results indicate that most of the test code generated by ReAccept are not only syntactically correct but also appropriately validate the production code changes, while existing techniques struggle to effectively update the test code. Overall, ReAccept achieved an improvement of 825%, 222%, and 100% over KNN, NMT, and CEPROT, respectively.

It is apparent that prior work has largely overlooked the challenge of updating the obsolete test code, possibly due to its complexity. KNN and NMT are general-purpose methods that can be applied to various tasks, but their performance in PT co-evolution is not satisfactory enough. CEPROT exhibits reduced accuracy when tackling test code containing numerous statements, as the lengthy test code must be split into multiple smaller code blocks and updated separately. The underlying issue, as noted by the authors, is that their model is limited in handling long sequence tasks [22]. On the other hand, our approach ReAccept demonstrates significant improvements over the state-of-the-art approaches when tackling such lengthy test code.

Performance Fluctuation: Due to the diverse outputs of the underlying LLM, we conducted five repetitions of the experiment for each sample, analyzing the performance fluctuation when updating the test code. On average, ReAccept achieved a CSR of 79.81%, a TPS of 69.13%, and a UCR of 65.63%. The mean absolute deviations for these metrics are 4.50%, 6.06%, and 6.14%, with variances of 0.0021642, 0.0046904, and 0.0043988, respectively.

4.3.4 RQ4: Ablation studies. We conducted multiple ablation studies to evaluate the impact of the following parameters on ReAccept.

Table 5. Dynamic performance of different techniques on the obsolete test update task

Approach	KNN			NMT			CEPROT			ReAccept		
	CSR	TPS	UCR	CSR	TPS	UCR	CSR	TPS	UCR	CSR	TPS	UCR
springside4	66.67%	66.67%	0.00%	33.33%	33.33%	33.33%	66.67%	66.67%	66.67%	100.00%	100.00%	100.00%
commons-lang	62.50%	50.00%	12.50%	37.50%	25.00%	25.00%	62.50%	62.50%	62.50%	87.50%	87.50%	87.50%
dddlib	37.50%	37.50%	12.50%	12.50%	12.50%	12.50%	37.50%	37.50%	37.50%	25.00%	25.00%	25.00%
datumbox	22.22%	0.00%	0.00%	22.22%	22.22%	22.22%	44.44%	33.33%	33.33%	77.78%	77.78%	77.78%
openmrs-core	37.93%	24.14%	6.90%	31.03%	24.14%	20.69%	44.83%	31.03%	24.14%	82.76%	65.52%	65.52%
basex	47.83%	34.78%	8.70%	26.09%	26.09%	23.91%	52.17%	41.30%	36.96%	97.83%	84.78%	78.26%
Average	43.69%	31.07%	7.77%	30.10%	24.27%	22.33%	49.52%	39.82%	35.92%	**85.44**%	**74.76**%	**71.84**%

- Choice of LLMs: We evaluated the performance of ReAccept with gpt-3.5-turbo-0125, gpt-4-0125-preview, gpt-4-turbo-2024-04-09 and gpt-4o-2024-05-13 to understand the impact of different LLMs.
- Temperature: We evaluated the performance of ReAccept with different temperatures (ranging from 0 to 1), which controls the randomness of the generated response from the LLMs.
- Top-p: We evaluated the performance of ReAccept with different values for nucleus sampling (ranging from 0 to 1) that is used to control the diversity of the generated text. Note that a top-p value of x causes the LLMs to generate text using tokens with probabilities higher than x.
- RAG and DV(dynamic validation): We compared two variants without RAG, i.e., zero-shot, and without dynamic validation.
- Embedding model: We evaluated the performance of ReAccept with three different embedding models, including 1) text-embedding-3-large, 2) text-embedding-3-small, and 3) text-embedding-ada-002. to compare their impact on PT co-evolution tasks.
- Iteration number for the update phase: The dynamic validation process will keep executing until generating valid test code or reaching a pre-defined cutoff. By default, we limited the iteration number to 8. We aimed to evaluate whether this threshold has a significant impact on ReAccept.

Figure 7 presents the results of our ablation study, from which several conclusions can be drawn.

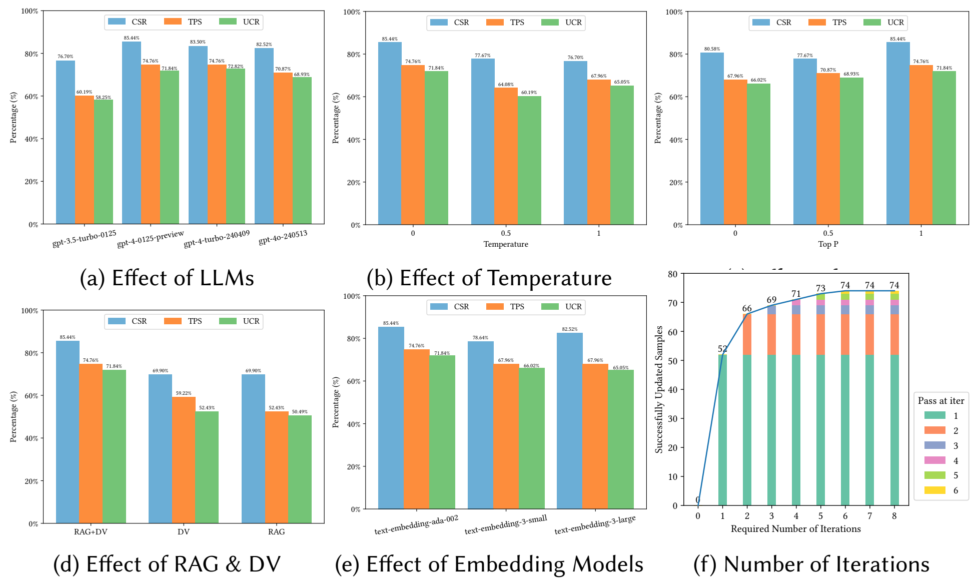
Different LLMs. All three GPT-4 versions outperformed GPT-3.5, and the differences among the GPT-4 versions were minimal. Specifically, gpt-4-0125-preview improves CSR, TPS and UCR by 8.74%, 14.57% and 13.59% over gpt-3.5-turbo-0125. The performance of gpt-4-0125-preview and gpt-4-turbo-2024-04-09 is almost equal. Surprisingly, the latest model, gpt-4o-2024-05-13, showed a decrease of 2.92%, 3.89%, and 2.91% in CSR, TPS, and UCR respectively, compared to gpt-4-0125-preview.

Temperature. The updated test code's CSR and TPS decline as the temperature increases, peaking at 0. In contrast, UCR reaches its lowest value at a temperature of 0.5.

Top-p. We observed that a higher top-p improves the performance of ReAccept. A low top-p value limits the number of output candidates, which may result in missing keywords or identifiers in the test code. Increasing the top-p value helps retain essential candidate tokens in LLMs.

RAG and DV. The use of RAG improves the relevance of its output to code changes, significantly improving CSR, TPS, and UCR. Furthermore, dynamic validation results guide LLMs in optimizing low-quality test code, leading to substantial improvements in these metrics.

Embedding model. When using text-embedding-ada-002 for tokenization, ReAccept successfully fixed the most samples compared to the other two embedding models. Although released earlier, text-embedding-ada-002 achieved the highest scores on all three metrics.



(a) Effect of LLMs (b) Effect of Temperature

(d) Effect of RAG & DV (e) Effect of Embedding Models (f) Number of Iterations

Fig. 7. Ablation study results

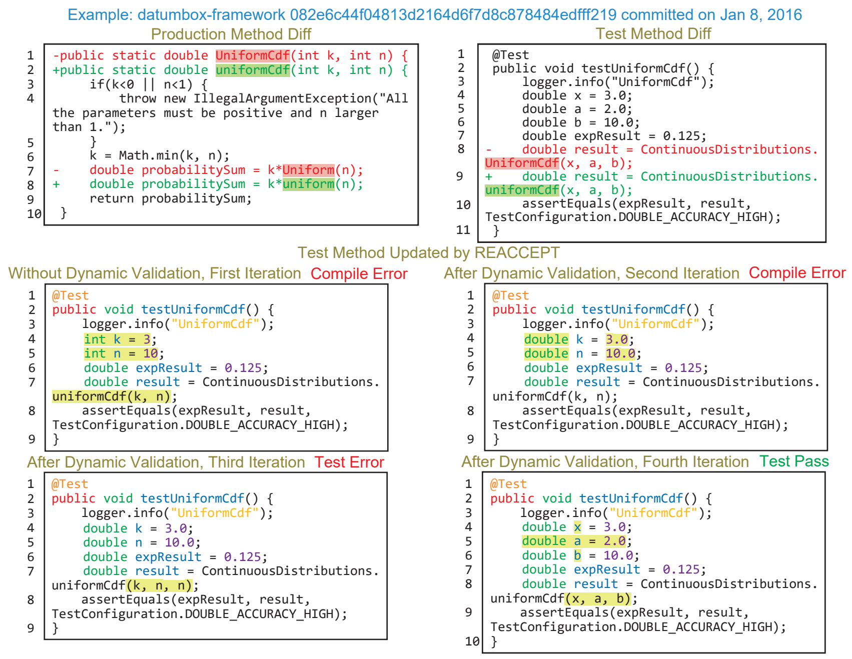
- CSR: text-embedding-ada-002 achieved the highest rate at 85.44%, surpassing text-embedding-3-small (78.64%) by 6.80% and text-embedding-3-large (82.52%) by 2.92%.
- TPS: text-embedding-ada-002 reached 74.76%, outperforming text-embedding-3-small (67.96%) by 6.80% and text-embedding-3-large (72.82%) by 1.94%.
- UCR: text-embedding-ada-002 recorded 71.84%, improving upon text-embedding-3-small (66.02%) by 5.82% and text-embedding-3-large (68.93%) by 2.91%.

Number of iterations. To determine the necessary iterations for PT co-evolution, we analyzed all 103 samples, with ReAccept successfully updating 74 of them. Figure 7f illustrates the first occurrence of a correct patch when ReAccept tackles a sample. We observed that 50.49% samples (UCR) were successfully updated in the first iteration. CSR and TPS reach 52.42% and 67.96% in the first iteration. Five samples required three or more iterations for a correct update, and all samples were successfully updated within six iterations. These results highlight the importance of having a sufficient number of iterations to complete the updating process.

4.4 Discussion

It was observed that ReAccept's performance on *dddlib* is not as strong as expected. After an in-depth investigation on *dddlib*, it turns out that the root cause of such performance downgrade lies in the unsuccessful updated samples. The *dddlib* project contains two successfully configured commits with four samples each. In commit *32f7a3d1d6d33c31e154fe893ab8ea3b73b32389*, out of the four samples, only the first sample is successfully updated, while the remaining three contain compilation errors. Upon analyzing these samples, we found that all samples have similar modified structures but different identifiers, i.e., variable names and method names used in the test cases, leading to the retrieval of a suboptimal sample. A similar result occurs when ReAccept tackling the other commit *566855c10bb0d875e493ef0c5d4d075c80c20a78*, in which one out of the four samples is correctly updated. Therefore, in the RAG process for the update task, leveraging similarities in code and modification structures may be more effective than relying solely on textual similarity.

To better clarify the effect of dynamic validation in optimizing the test code generation, we conducted a case study on a successfully updated sample from *datumbox-framework*. As shown in


Example: datumbox-framework 082e6c44f04813d2164d6f7d8c878484edfff219 committed on Jan 8, 2016
Production Method Diff
Test Method Diff
Test Method Updated by REACCEPT
Without Dynamic Validation, First Iteration
Compile Error
After Dynamic Validation, Second Iteration
Compile Error
After Dynamic Validation, Third Iteration
Test Error
After Dynamic Validation, Fourth Iteration
Test Pass

```
-public static double UniformCdf(int k, int n) {
+public static double uniformCdf(int k, int n) {
     if(k<0 || n<1) {
         throw new IllegalArgumentException("All
the parameters must be positive and n larger
than 1.");
     }
     k = Math.min(k, n);
-    double probabilitySum = k*Uniform(n);
+    double probabilitySum = k*uniform(n);
     return probabilitySum;
 }
```

```
 @Test
 public void testUniformCdf() {
     logger.info("UniformCdf");
     double x = 3.0;
     double a = 2.0;
     double b = 10.0;
     double expResult = 0.125;
-    double result = ContinuousDistributions.
UniformCdf(x, a, b);
+    double result = ContinuousDistributions.
uniformCdf(x, a, b);
     assertEquals(expResult, result,
TestConfiguration.DOUBLE_ACCURACY_HIGH);
 }
```

```
@Test
public void testUniformCdf() {
    logger.info("UniformCdf");
    int k = 3;
    int n = 10;
    double expResult = 0.125;
    double result = ContinuousDistributions.
uniformCdf(k, n);
    assertEquals(expResult, result,
TestConfiguration.DOUBLE_ACCURACY_HIGH);
}
```

```
@Test
public void testUniformCdf() {
    logger.info("UniformCdf");
    double k = 3.0;
    double n = 10.0;
    double expResult = 0.125;
    double result = ContinuousDistributions.
uniformCdf(k, n);
    assertEquals(expResult, result,
TestConfiguration.DOUBLE_ACCURACY_HIGH);
}
```

```
@Test
public void testUniformCdf() {
    logger.info("UniformCdf");
    double k = 3.0;
    double n = 10.0;
    double expResult = 0.125;
    double result = ContinuousDistributions.
uniformCdf(k, n, n);
    assertEquals(expResult, result,
TestConfiguration.DOUBLE_ACCURACY_HIGH);
}
```

```
@Test
public void testUniformCdf() {
    logger.info("UniformCdf");
    double x = 3.0;
    double a = 2.0;
    double b = 10.0;
    double expResult = 0.125;
    double result = ContinuousDistributions.
uniformCdf(x, a, b);
    assertEquals(expResult, result,
TestConfiguration.DOUBLE_ACCURACY_HIGH);
}
```

Fig. 8. Impact of dynamic validation on updating obsolete test code

Figure 8, developers renamed the method in the production code, resulting in a compilation error when launching the existing test code. To fix such an error, the test code must be refactored to invoke the method through the correct method name *uniformCdf*. After ReAccept identified this obsolete test code, it proceeded to the updating task using the LLM.

In the first iteration, the LLM renamed the *UniformCdf* method called in the test code. However, due to the influence of the non-changed part of the production code, the LLM incorrectly called the *uniformCdf* method, reasoning a Java compilation error and inconsistent semantics compared to the original one. In the second iteration, the LLM accepted the results of the dynamic validation and modified the actual argument type of *uniformCdf*. But it didn't modify the length of the actual argument list, the compilation error still occurred. In the third iteration, the LLM attempted to pass an extra *n* into the actual argument list and the compilation was successful. However, when executing the test, a test error was raised for the formal argument *a* in the *uniformCdf* method needed to be less than *b*. In the fourth iteration, the LLM finally modified the formal argument when calling *uniformCdf* correctly so that the test code can be executed successfully.

This example highlights the advantages of using dynamic validation with an LLM to update test code, significantly enhancing both the quality and practical performance of the test code.

4.5 Threats to Validity

4.5.1 Internal Validity. Although we've summarized general experience to guide the LLM, its quantity and quality are still not comprehensive enough. In complex or unique cases, this experience may not cover all possible code change scenarios. Expanding and enhancing the quality of this experience, could further improve identification accuracy.

In addition, despite utilizing RAG and dynamic validation to guide LLM in updating the obsolete test code, there are still some update failures. We use code diff as the code change representation, which is LLM-friendly. However, during knowledge retrieval, focusing solely on textual similarity may overlook code structure and change types. Introducing a vector representation of AST differences for a more fine-grained, structured search could enhance the effectiveness.

Furthermore, during the identification phase, we observed that LLMs exhibited a cognitive bias toward the negative class. LLMs often overestimate the need for test code updates when assessing the effect of production code changes. To mitigate this bias, we increased the proportion of negative samples in the training set, improving ReAccept's performance during identification.

4.5.2 Data Validity. We collected co-evolution samples from SITAR, CHOSEN, and CEPROT, mined using heuristic rules. Upon reviewing some code change pairs, we found that the production and test code only have dependencies, with uncorrelated changes and mismatched class or method names. These production code changes are insufficient for the LLM to effectively update the obsolete test code, resulting in low test coverage.

4.5.3 External Validity. The LLM used in our experiments is gpt-4-0125-preview. While it has moderate NLP and code understanding capabilities, its performance is average in the current competitive LLM landscape. We believe that using more advanced models could significantly improve identification accuracy and effectiveness.

5 Related Work

Previous work has proposed various techniques to model correlations in the code base [10, 28, 35, 36, 44, 48, 51–53, 60]. Commonly, these methods define heuristic rules to create traceability links. Zaidman et al. [60] were among the first to study PT co-evolution patterns in two open-source projects, using software visualization techniques to illustrate co-evolution trends. White et al. [56] proposed TCtracer, which automatically establishes traceability links between production and test code at the method and class levels. Huang et al. [23] introduced a new approach that gauges the likelihood of co-evolution through extracted code changes, code complexity, and certain semantic features.

In recent years, numerous studies on identifying and updating PT co-evolution relations have emerged [40, 45, 49, 55]. Wang et al. [54] studied the factors influencing test code updates. The proposed SITAR model uses historical PT co-evolution data to identify obsolete test code. Liu [34] proposed a machine-learning-based method called Drift, which predicts obsolete test cases at the method level. Hu [22] introduced a novel deep-learning-based method, CEPROT, to identify obsolete test cases and automatically update them. Sun et al. [52] explored the validity of common assumptions for collecting and labeling PT co-evolution samples. They also proposed a two-stage strategy-based method, CHOSEN, for identifying PT co-evolution. Yaraghi et al. [58] proposed a method called TARGET that utilizes pre-trained models in conjunction with program testing to repair broken test cases. TARGET may be applicable to PT co-evolution problems, but we did not find an open-source implementation of TARGET to compare with ReAccept. In addition, TARGET only partially validates the generated test cases by compiling and executing them, while our approach considers more metrics related to the quality of test code.

6 Conclusion

In this work, we propose ReAccept, a novel approach that leverages LLMs and dynamic validation to fully automate PT co-evolution. The evaluation results on a dataset of 537 collected Java projects show that ReAccept achieves significant improvements. In the future, we intend to investigate additional strategies to improve ReAccept further.

References

[1] Augest 2, 2024. REACCEPT: REasoning-Action mechanism and Code dynamic validation assisted Co-Evolution of Production and Test code. https://github.com/Timiyang-ai/REACCEPT.git.

[2] July 30, 2024. Chroma - the open-source embedding database. https://github.com/chroma-core/chroma.

[3] July 30, 2024. JaCoCo Java Code Coverage Library. https://www.jacoco.org/jacoco/.

[4] July 30, 2024. javac - Java programming language compiler. https://docs.oracle.com/javase/7/docs/technotes/tools/solaris/javac.html.

[5] July 30, 2024. JUnit 4. https://junit.org/junit4/.

[6] July 30, 2024. Langchain Framework. https://www.langchain.com/.

[7] July 30, 2024. Machine Learning Platform - Text Analysis Service | Datumbox. https://www.datumbox.com/.

[8] July 30, 2024. OpenAI Embeddings. https://platform.openai.com/docs/guides/embeddings.

[9] Georg Buchgeher, Christian Ernstbrunner, Rudolf Ramler, and Michael Lusser. 2013. Towards tool-support for test case selection in manual regression testing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 74–79. doi:10.1109/ICSTW.2013.16

[10] Yufan Cai, Yun Lin, Chenyan Liu, Jinglian Wu, Yifan Zhang, Yiming Liu, Yeyun Gong, and Jin Song Dong. 2024. On-the-fly adapting code summarization on trainable cost-effective language models. *Advances in Neural Information Processing Systems* 36 (2024).

[11] Saikat Chakraborty, Shuvendu K Lahiri, Sarah Fakhoury, Madanlal Musuvathi, Akash Lal, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2023. Ranking llm-generated loop invariants for program verification. *arXiv preprint arXiv:2310.09342* (2023). doi:10.48550/arXiv.2310.09342

[12] Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. 2024. CodeR: Issue Resolving with Multi-Agent and Task Graphs. *arXiv preprint arXiv:2406.01304* (2024). doi:10.48550/arXiv.2406.01304

[13] Jianlei Chi, Yu Qu, Ting Liu, Qinghua Zheng, and Heng Yin. 2022. Seqtrans: automatic vulnerability fix via sequence to sequence learning. *IEEE Transactions on Software Engineering* 49, 2 (2022), 564–585. doi:10.1109/TSE.2022.3156637

[14] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639219

[15] Emelie Engström and Per Runeson. 2010. A qualitative survey of regression testing practices. In *Product-Focused Software Process Improvement: 11th International Conference, PROFES 2010, Limerick, Ireland, June 21-23, 2010. Proceedings 11*. Springer, 3–16. doi:10.1007/978-3-642-13792-1_3

[16] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 313–324. doi:10.1145/2642937.2642982

[17] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*. 935–947. doi:10.1145/3540250.3549098

[18] Mats Grindal, Jeff Offutt, and Jonas Mellin. 2006. On the testing maturity of software producing organizations. In *Testing: Academic & Industrial Conference-Practice And Research Techniques (TAIC PART'06)*. IEEE, 171–180.

[19] Qiuhan Gu. 2023. Llm-based code generation method for golang compiler testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2201–2203. doi:10.1145/3611643.3617850

[20] Qi Guo, Xiaohong Li, Xiaofei Xie, Shangqing Liu, Ze Tang, Ruitao Feng, Junjie Wang, Jidong Ge, and Lei Bu. 2024. FT2Ra: A Fine-Tuning-Inspired Approach to Retrieval-Augmented Code Completion. *arXiv preprint arXiv:2404.01554* (2024). doi:10.1145/3650212.3652130

[21] Yuejun Guo, Qiang Hu, Xiaofei Xie, Maxime Cordy, Mike Papadakis, and Yves Le Traon. 2023. KAPE: k NN-based performance testing for deep code search. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–24. doi:10.1145/3624735

[22] Xing Hu, Zhuang Liu, Xin Xia, Zhongxin Liu, Tongtong Xu, and Xiaohu Yang. 2023. Identify and Update Test Cases When Production Code Changes: A Transformer-Based Approach. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1111–1122. doi:10.1109/ASE56229.2023.00165

[23] Yuan Huang, Zhicao Tang, Xiangping Chen, and Xiaocong Zhou. 2024. Towards automatically identifying the co-change of production and test code. *Software Testing, Verification and Reliability* 34, 3 (2024), e1870. doi:10.1002/stvr.1870

[24] Victor Hurdugaci and Andy Zaidman. 2012. Aiding software developers to maintain developer tests. In *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 11–20. doi:10.1109/CSMR.2012.12

[25] Marko Ivankovic, Goran Petrovic, Yana Kulizhskaya, Mateusz Lewko, Luka Kalinovcic, René Just, and Gordon Fraser. 2024. Productive Coverage: Improving the Actionability of Code Coverage. In *Proceedings of the 46th International*

Conference on Software Engineering: Software Engineering in Practice. 58–68. doi:10.1145/3639477.3639733

[26] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173. doi:10.1109/ICSE43902.2021.00107

[27] Kailun Jin, Chung-Yu Wang, Hung Viet Pham, and Hadi Hemmati. 2024. Can ChatGPT Support Developers? An Empirical Evaluation of Large Language Models for Code Generation. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 167–171. doi:10.1145/3643991.3645074

[28] Tenma Kitai, Hirohisa Aman, Sousuke Amasaki, Tomoyuki Yokogawa, and Minoru Kawahara. 2022. Have Java Production Methods Co-Evolved With Test Methods Properly?: A Fine-Grained Repository-Based Co-Evolution Analysis. In *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 120–124. doi:10.1109/SEAA56994.2022.00027

[29] Bonan Kou, Shengmai Chen, Zhijie Wang, Lei Ma, and Tianyi Zhang. 2023. Is model attention aligned with human attention? an empirical study on large language models for code generation. *arXiv preprint arXiv:2306.01220* (2023). doi:10.48550/arXiv.2306.01220

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

[31] Xiangyu Li, Marcelo d'Amorim, and Alessandro Orso. 2019. Intent-preserving test repair. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 217–227. doi:10.1109/ICST.2019.00030

[32] Xiaoli Lian, Shuaisong Wang, Jieping Ma, Xin Tan, Fang Liu, Lin Shi, Cuiyun Gao, and Li Zhang. 2024. Imperfect Code Generation: Uncovering Weaknesses in Automatic Code Generation by Large Language Models. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 422–423. doi:10.1145/3639478.3643081

[33] Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. 2024. Guiding ChatGPT for Better Code Generation: An Empirical Study. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 102–113. doi:10.1109/SANER60148.2024.00018

[34] Lei Liu, Sinan Wang, Yepang Liu, Jinliang Deng, and Sicen Liu. 2023. Drift: Fine-Grained Prediction of the Co-Evolution of Production and Test Code via Machine Learning. In *Proceedings of the 14th Asia-Pacific Symposium on Internetware*. 227–237. doi:10.1145/3609437.3609449

[35] Zeeger Lubsen, Andy Zaidman, and Martin Pinzger. 2009. Using association rules to study the co-evolution of production & test code. In *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 151–154. doi:10.1109/MSR.2009.5069493

[36] Cosmin Marsavina, Daniele Romano, and Andy Zaidman. 2014. Studying fine-grained co-evolution patterns of production and test code. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 195–204. doi:10.1109/SCAM.2014.28

[37] Simone Mezzaro, Alessio Gambi, and Gordon Fraser. 2024. An Empirical Study on How Large Language Models Impact Software Testing Learning. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 555–564. doi:10.1145/3661167.3661273

[38] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. 2024. ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2332–2354. doi:10.1145/3660810

[39] Wendkûuni C Ouédraogo, Kader Kaboré, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawendé F Bissyandé. 2024. Large-scale, Independent and Comprehensive study of the power of LLMs for test case generation. *arXiv preprint arXiv:2407.00225* (2024). doi:10.48550/arXiv.2407.00225

[40] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2016. Automatic test case generation: What if test code quality matters?. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 130–141. doi:10.1145/2931037.2931057

[41] Jiantao Pan. 1999. Software testing. *Dependable Embedded Systems* 5, 2006 (1999), 1.

[42] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318. doi:10.3115/1073083.1073135

[43] Yun Peng, Shuzheng Gao, Cuiyun Gao, Yintong Huo, and Michael Lyu. 2024. Domain knowledge matters: Improving prompts with fix templates for repairing python type errors. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3608132

[44] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2012. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering*. 1–11. doi:10.1145/2393596.2393634

[45] Sanyogita Piya and Allison Sullivan. 2023. LLM4TDD: Best Practices for Test Driven Development Using Large Language Models. *arXiv preprint arXiv:2312.04687* (2023). doi:10.48550/arXiv.2312.04687

[46] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020). doi:10.48550/arXiv.2009.10297

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

[48] Samiha Shimmi and Mona Rahimi. 2022. Leveraging code-test co-evolution patterns for automated test case recommendation. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*. 65–76. doi:10.1145/3524481.3527222

[49] Jiho Shin, Sepehr Hashtroudi, Hadi Hemmati, and Song Wang. 2024. Domain Adaptation for Code Model-based Unit Test Case Generation. doi:10.48550/arXiv.2308.08033

[50] Mats Skoglund and Per Runeson. 2004. A case study on regression test suite maintenance in system evolution. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 438–442. doi:10.1109/ICSM.2004.1357831

[51] Jeongju Sohn and Mike Papadakis. 2022. Using Evolutionary Coupling to Establish Relevance Links Between Tests and Code Units. A case study on fault localization. *arXiv preprint arXiv:2203.11343* (2022). doi:10.48550/arXiv.2203.11343

[52] Weifeng Sun, Meng Yan, Zhongxin Liu, Xin Xia, Yan Lei, and David Lo. 2023. Revisiting the Identification of the Co-evolution of Production and Test Code. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–37. doi:10.1145/3607183

[53] Bart Van Rompaey and Serge Demeyer. 2009. Establishing traceability links between unit test cases and units under test. In *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 209–218. doi:10.1109/CSMR.2009.39

[54] Sinan Wang, Ming Wen, Yepang Liu, Ying Wang, and Rongxin Wu. 2021. Understanding and facilitating the co-evolution of production and test code. In *2021 IEEE International conference on software analysis, evolution and reengineering (SANER)*. IEEE, 272–283. doi:10.1109/SANER50967.2021.00033

[55] Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2024. TESTEVAL: Benchmarking Large Language Models for Test Case Generation. *arXiv preprint arXiv:2406.04531* (2024). doi:10.48550/arXiv.2406.04531

[56] Robert White, Jens Krinke, and Raymond Tan. 2020. Establishing multilevel test-to-code traceability links. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 861–872. doi:10.1145/3377811.3380921

[57] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629* (2022). doi:10.48550/arXiv.2210.03629

[58] Ahmadreza Saboor Yaraghi, Darren Holden, Nafiseh Kahani, and Lionel Briand. 2024. Automated Test Case Repair Using Language Models. *arXiv preprint arXiv:2401.06765* (2024). doi:10.48550/arXiv.2401.06765

[59] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and Improving ChatGPT for Unit Test Generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1703–1726. doi:10.1145/3660783

[60] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie Van Deursen. 2008. Mining software repositories to study co-evolution of production & test code. In *2008 1st international conference on software testing, verification, and validation*. IEEE, 220–229. doi:10.1109/ICST.2008.47

[61] Andy Zaidman, Bart Van Rompaey, Arie Van Deursen, and Serge Demeyer. 2011. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering* 16 (2011), 325–364. doi:10.1007/s10664-010-9143-7

[62] Yuqi Zhu, Jia Li, Ge Li, YunFei Zhao, Zhi Jin, and Hong Mei. 2024. Hot or Cold? Adaptive Temperature Sampling for Code Generation with Large Language Models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 437–445. doi:10.1609/aaai.v38i1.27798

[63] Daniel Zimmermann and Anne Koziolek. 2023. Automating gui-based software testing with gpt-3. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 62–65. doi:10.1109/ICSTW58534.2023.00022

Received 2025-02-27; accepted 2025-03-31