Revisiting the Identification of the Co-Evolution of Production and Test Code

WEIFENG SUN, Chongqing University, China
MENG YAN*, Chongqing University, China
ZHONGXIN LIU, Zhejiang University, China
XIN XIA, Zhejiang University, China
YAN LEI, Chongqing University, China
DAVID LO, Singapore Management University, Singapore

Many software processes advocate that the test code should *co-evolve* with the production code. Prior work usually studies such co-evolution based on *production-test co-evolution samples* mined from software repositories. A production-test co-evolution sample refers to a pair of a test code change and a production code change where the test code change triggers or is triggered by the production code change. The quality of the mined samples is critical to the reliability of research conclusions. Existing studies mined production-test co-evolution samples based on the following assumption: **if a test class and its associated production class change together in one commit, or a test class changes immediately after the changes of associated production class within a short time interval, this *change pair* should be a *production-test co-evolution sample***. However, the validity of this assumption has never been investigated.

To fill this gap, we present an empirical study, investigating the reasons for test code updates occurring after the associated production code changes, and revealing the pervasive existence of noise in the production-test co-evolution samples identified based on the aforementioned assumption by existing works. We define a taxonomy of such noise, including 6 categories (i.e., adaptive maintenance, perfective maintenance, corrective maintenance, indirectly related production code update, indirectly related test code update, and other reasons). Guided by the empirical findings, we propose **CHOSEN** (an identifi**C**ation met**H**od **O**f production-te**S**t co-**E**volutio**N**) based on a two-stage strategy. **CHOSEN** takes a test code change and its associated production code change as input, aiming to determine whether the production-test change pair is a production-test co-evolution sample. Such identified samples are the basis of or are useful for various downstream tasks. We conduct a series of experiments to evaluate our method. Results show that: 1) **CHOSEN** achieves an AUC of 0.931 and an F1-score of 0.928, significantly outperforming existing identification methods. 2) **CHOSEN** can help researchers and practitioners draw more accurate conclusions on studies related to the co-evolution of production and test code. For the task of Just-In-Time (JIT) obsolete test code detection, which can help detect whether a piece of test code should be updated when developers modify the production code, the test set constructed by **CHOSEN** can help measure the detection method's performance more accurately, only leading to 0.76% of average error compared with ground truth. In addition, the dataset constructed by **CHOSEN** can be used to train a better obsolete test code detection model, of which the average improvements on accuracy, precision, recall, and F1-score are 12.00%, 17.35%, 8.75%, and 13.50% respectively.

*corresponding author.

Authors' addresses: Weifeng Sun, weifeng.sun@cqu.edu.cn, Chongqing University, China; Meng Yan, Chongqing University, China, mengy@cqu.edu.cn; Zhongxin Liu, Zhejiang University, China, liu_zx@zju.edu.cn; Xin Xia, Zhejiang University, China, xin.xia@acm.org; Yan Lei, Chongqing University, China, yanlei@cqu.edu.cn; David Lo, Singapore Management University, Singapore, davidlo@smu.edu.sg.

1049-331X/2023/1-ART1 $15.00
https://doi.org/10.1145/3607183

CCS Concepts: • **Software and its engineering → Software evolution**; **Empirical software validation**; **Software version control**.

Additional Key Words and Phrases: Empirical software engineering, Mining software repositories, Software evolution, Software testing

ACM Reference Format:
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 Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2023), 38 pages. https://doi.org/10.1145/3607183

1 INTRODUCTION

A software system must evolve or it will become less useful over time [42, 55, 74, 88]. During evolution, the production code is constantly modified and updated to address new requirements or possible issues that may arise. Test code, as an essential artifact, should *co-evolve* with the production code to ensure that the associated production code meets specification, which is referred to as *co-evolution of production and test code* [81], hereon *PT co-evolution*. Despite the importance of *PT co-evolution*, developers may forget or ignore updating test code, which brings in *obsolete test code* [81], thereby increasing the cost of development and maintenance [50, 65].

PT co-evolution has been attracting continued interests from both academia and industry. On the one hand, researchers [42, 55, 88] have proposed various visualization techniques to display production and test file changes over time (*PT co-evolution*) and group files that change together. Such visualizations can help analysts recognize different *PT co-evolution* scenarios, obtaining relevant observations. Further, Wang et al. have proposed a state-of-the-art method for Just-In-Time (JIT) obsolete test code detection (for short, OTCD), named SITAR [81]. SITAR enables learning a model from the historical data of *PT co-evolution* in a project and help detect whether a piece of test code should be updated when developers modify the production code.

Identifying and mining *production-test co-evolution samples* is the basis of existing studies related to *PT co-evolution*, such as *PT co-evolution* visualization [42, 55, 88] and JIT obsolete test code detection [81]. A *production-test co-evolution sample*, hereon *PT co-evolution sample*, refers to a pair of a test code change and a production code change where the test one triggers or is triggered by the production one. Unfortunately, given a test code change and a production code change, it is hard to automatically determine whether they co-evolve or not, because: 1) Associating a test code snippet with a production code snippet is already a non-trivial task if there are no explicit links between them (e.g., their names follow some conventions). 2) Precisely determining whether a test code co-evolves with its associated production code requires comprehending their changes and understanding their relationship, which is also difficult.

To enable the mining of *PT co-evolution samples*, existing work related to *PT co-evolution* widely use the following strategies to identify *PT co-evolution samples*: 1) Focusing on unit test classes and taking advantage of naming conventions to ease the association of test and production classes. For example, given a unit test class `FooTest`, only its tested production class (i.e., `Foo`) will be considered when constructing *PT co-evolution samples*. For convenience, hereon, we refer to such tested production code as *associated production code*. 2) Collecting *PT co-evolution samples* based on the following assumption [42, 55, 81, 88]:

Assumption .1

If a test class and its associated production class change together in one commit [42, 55, 88], or a test class changes immediately after the changes of associated production class within a short time interval (denoted by T), this change pair should be a production-test co-evolution sample [81].

For example, in Wang et al.'s work [81], if a test class is updated within 48 hours after the change of its associated production class, such *change pair* is considered as a *PT co-evolution sample*.

However, we find that even if we only focus on unit test classes and associate test and production classes based on naming conventions, the widely used assumption (shown in **Assumption .1**) does not always hold. For example, Figure 1 shows a commit collected from the Easy-rules [24] project, where a test class (i.e., JexlActionTest) and its associated production class (i.e., JexlAction) change simultaneously. In the production class (Listing 1), a developer modified the logger message by updating "evaluate" to "execute" to help users understand the JexlException. In Listing 2, the test code encapsulated the technical facts (i.e., "System.out") into the new Jexl namespaces. Note that, *PT co-evolution* implies that production code modifications and test code modifications are necessarily correlated couplings, i.e., the test code change triggers or is triggered by the production one. For the abovementioned example, although the production code and test code are modified with a similar purpose: to facilitate future project maintenance, the content of their modifications

Fig. 1. False *co-evolution* example in `easy-rules` project.

Listing 1. `JexlAction.java` change (simplified, commit `1a06601`).

```
index ca2bf9b..84fd15c 100644
-    public void execute(Facts facts) throws Exception {
+    public void execute(Facts facts) {
         Objects.requireNonNull(facts, "facts cannot be null");
         MapContext ctx = new MapContext(facts.asMap());
         try {
             compiledScript.execute(ctx);
         } catch (JexlException e) {
-            LOGGER.error("Unable to evaluate expression:'"+expression+"' on facts: "+facts, e);
+            LOGGER.error("Unable to execute expression: '"+expression+"' on facts: "+facts, e);
             throw e;
         }
     }
```

Listing 2. `JexlActionTest.java` change (simplified, commit `1a06601`).

```
index 7b99721..d35142c 100644
-    public void testJexlFunctionExecution() throws Exception {
+    public void testJexlActionExecutionWithCustomFunction() throws Exception {
-        Action printAction = new JexlAction("var hello = function() {
-                System.out.println(\"Hello from JEXL!\"); }; hello();");
+        PrintStream originalStream = System.out;
+        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
+        System.setOut(new PrintStream(outputStream));
+        Map<String, Object> namespaces = new HashMap<>();
+        namespaces.put("sout", System.out);
+        JexlEngine jexlEngine = new JexlBuilder().namespaces(namespaces).create();
+        Action printAction = new JexlAction("var hello = function() {
+                sout:println(\"Hello from JEXL!\"); }; hello();", jexlEngine);
```

Fig. 2. False *co-evolution* example in `java-faker` project.

Listing 3. `Hacker.java` change (simplified, commit `f574317`).

```
index 3a84b23..2ec2597 100644
 public class Hacker {
+    private final Faker faker;
-    private final FakeValuesServiceInterface fakeValuesService;
-    public Hacker(FakeValuesServiceInterface fakeValuesService) {
-        this.fakeValuesService = fakeValuesService;
+    Hacker(Faker faker) {
+        this.faker = faker;
     }
}
```

Listing 4. `HackerTest.java` change (simplified, commit `f574317`).

```
index 2af4d21..8639213 100644
-   public class HackerTest {
-       private Faker faker;
-       @Before
-       public void before() {
-           faker = new Faker();
-       }
+   public class HackerTest  extends AbstractFakerTest{
        ...
    }
```

is not related. In other words, the inference that the production code modified the logger message of the JexlException must cause the test code to encapsulate the technical facts does not hold. Figure 2 shows another example. Specifically, in the production code (Listing 3), the developer made two changes: they replaced the private field `fakeValuesService` with `faker` and modified the constructor to initialize `faker`. Meanwhile, in Listing 4, the test code extracted the method with the @Before annotation[1] into a base test class and inherited from that base class. This modification of the test code is a code refactoring activity, which aims to improve test code readability, reduce code duplication, and facilitate future maintenance of the test cases' common behavior. In contrast, the changes made to the production code involve the updates of a private property and a method, which should not result in the test code removing `faker`, or `before` method. Therefore, the changes made to the test code are independent of the changes made to the production code. Overall, we can see that in each example, the test code change is not triggered by the production code change.

These examples mean that the *PT co-evolution samples* collected by prior work based on the **Assumption .1** may contain noise, or in other words, false positives. Like any data-driven task, existing studies related to *PT co-evolution* are highly data-dependent. For example, for JIT obsolete test code detection [81], in order to learn complex features of production code changes, we require large *production-test change pairs* that have been labeled as either *PT co-evolution* or *PT non-co-evolution*. As for *PT co-evolution* visualization [42, 55, 88], developers and test engineers rely on different *PT co-evolution* scenarios mined from *PT co-evolution samples*. Moreover, some linking methods use *PT co-evolution samples* to construct the traceability between test code and production code (hereafter called test-to-code traceability) by assuming that test code and their associated production code usually co-evolve together throughout time [66, 83]. However, the noise might potentially impact the data quality and subsequently pose a threat to prior work's findings

[1]Annotating a `public void` method with @Before causes that method to be run before the test method

and conclusions. Data quality is an integral component of any data-driven system. The misinformation or noise can make benchmark performance results misleading [37, 70]. This can cause obsolete test code detection models to fail to generalize to real-world scenarios if they have not been trained with realistic, high-quality data. Furthermore, false positives may make linking methods incorrectly construct test-to-code traceability or confuse developers when identifying *co-evolution* scenarios in *PT co-evolution* visualization. Motivated by this, this work revisits the identification of the *co-evolution of production and test code*. To keep in line with prior work, we also focus on unit test classes and use naming conventions to associate test and production code. First, we investigate to what extent the assumption shown in **Assumption .1** is satisfied through an empirical study. Specifically, we carefully select 44 projects with good testing efforts [62] from GitHub, construct a total of 33,567 *PT co-evolution samples* based on the assumption, and manually inspect a statistically significant sample made up of 380 *PT co-evolution candidates*, representing the population with 95% confidence and 5% margin of error. The results show that depending on the selected time interval T, 11.34%-96.40% of the constructed *PT co-evolution samples* are false positives, i.e., the test code change is not triggered by the associated production code change. Then, based on our inspection, we define a taxonomy of the cases where the assumption shown in **Assumption .1** does not hold. The taxonomy consists of 6 categories, including adaptive maintenance, perfective maintenance, corrective maintenance, indirectly related production code update, indirectly related test code update, and other reasons.

The empirical study shows that the assumption of prior studies does not always hold in practice and the methods used by prior work to identify *PT co-evolution samples* may introduce noise. This prompted us to investigate what is the impact of such noise to the conclusion of the downstream tasks. We select the JIT obsolete test code detection (for short, OTCD) task [81] as the example, because this task can help demonstrate the impact of noise in a quantitative manner. OTCD can remind developers to update obsolete tests just after they change the production code, which can help developers 1) find the obsolete tests that would not fail and 2) find the obsolete tests that would fail without executing them. Specifically, we manually re-label 741 *production-test change pairs* collected and labeled by the identification method of SITAR (the state-out-the-art JIT OTCD method), and investigate the impacts of the mislabeled pairs on prior work's conclusions. From the results, the *production-test change pairs* mislabeled by the existing method significantly impacts the detection performance. The maximum precision decline is 13.33%. To reduce such noise, we propose an identifi**C**ation met**H**od **O**f production-te**S**t co-**E**volutio**N**, namely **CHOSEN**. Besides time intervals often adopted by previous works, **CHOSEN** attempts to understand the relationship between production and test code changes. Specifically, given a test code change $change_t$, **CHOSEN** first constructs its *production-test change pair* $\langle change_p, change_t \rangle$, where $change_p$ is the latest change of the associated production code that occurs in the same commit of or before $change_t$. Then, **CHOSEN** determines whether $change_t$ is triggered by $change_p$ based on the time interval between $change_t$ and $change_p$ and 5 fine-grained strategies learned from our empirical study (e.g., *production-test change pair* with non-semantic relevance is not co-evolving).

We evaluate the effectiveness of **CHOSEN** on *PT co-evolution* identification by comparing the *PT co-evolution samples* identified by **CHOSEN** with manually labeled ground truth (another set of 380 samples re-selected and labelled). The results show that **CHOSEN** achieves an F1-score of 0.928 and an AUC (area under the precision-recall curve) of 0.931, significantly better than existing identification methods. Moreover, we demonstrate how **CHOSEN** can be used to facilitate the downstream tasks related to *co-evolution of production and test code*. We select the OTCD task as the representative. Specifically, we first leverage **CHOSEN** to re-label 741 production-test change pairs collected and labeled by the identification method of SITAR and investigate the usefulness of **CHOSEN** in the OTCD task. Experimental results show that: 1) Measuring the OTCD model

on the test set labeled by existing identification methods leads to average performance errors of 6.01%, 8.40%, 6.96%, and 3.61% in terms of accuracy, precision, recall, and F1-score, respectively. In contrast, those errors are only 0.68%, 0.99%, 0.62%, and 0.76% on the test set labeled by **CHOSEN**. 2) After training the OTCD model on the training set labeled by **CHOSEN**, the model's precision, recall, and F1-score is improved by 17.35%, 8.75%, and 13.50%, respectively.

In summary, this paper makes the following contributions:

(1) We conduct an empirical study to investigate the validity of the assumption that is widely used by prior work [42, 55, 81, 88] (i.e., **if a test class changes together or immediately after the changes of associated production class within a short time interval, this change pair should be a production-test co-evolution sample**.) to collect and label *production-test co-evolution samples*. Results show that 11.34%-96.40% of the *PT co-evolution samples* constructed by such assumption are false positives.

(2) We perform an in-depth analysis on the cases where the assumption used by prior work does not hold and summarize 6 types of test code change patterns which are not triggered by the changes of the associated production code. To the best of our knowledge, this is the first work that investigates such test code change patterns.

(3) We propose a method, namely **CHOSEN**, based on our empirical study's findings to identify *production-test co-evolution samples*. Experimental results show that, compared to existing identification methods, **CHOSEN** achieves significantly better F1-score and AUC. Moreover, for the task of JIT obsolete test code detection, the test set labeled by **CHOSEN** can help measure the detection model's performance more accurately, and the training set labeled by **CHOSEN** can help improve the performance of the detection model.

(4) Finally, we built a dataset with over 46k *production-test change pairs*, including more than 1k manually labeled pairs (380 pairs for empirical studies, 380 pairs to verify **CHOSEN** effectiveness, and 741 pairs to demonstrate the usefulness of **CHOSEN** for downstream tasks). We open these data and our scripts for follow-up work[2].

The paper is organized as follows: In Section 2, we describe the empirical study that is performed to explore to what extent the **Assumption .1** is satisfied. We demonstrate the impacts of mislabeled pairs on prior work's conclusions and elaborate on our *co-evolution sample* identification method in Section 3. Section 4 evaluates the effectiveness of our method on *co-evolution* identification and the usefulness of **CHOSEN** on downstream tasks related to *production-test co-evolution*. In Section 5, we discuss the limitations of this paper, the implications for researchers and practitioners, and the threats to validity. Section 6 presents a brief review of related work. Finally, Section 7 concludes the paper and mentions future work.

2 EMPIRICAL STUDY

The goal of the empirical study is to investigate the validity of assumption that is test code and associated production code are co-evolving if a test class and its associated production class change together in one commit [42, 55, 88], or a test class changes immediately after the changes of associated production class within a short time interval (denoted by T). We first build a dataset of *PT co-evolution samples* based on the assumption. Then, some samples are randomly selected and manually labeled to check whether the test code update is attributable to the change of the associated production code. For the samples where the test code change is not triggered by the associated production code change, we further analyze the reasons behind the test code change. This empirical study helps us answer the following two research questions:

[2]https://anonymous.4open.science/r/CHOSEN-DE81

RQ1.1: *Are production-test change pairs whose test class changes immediately after the changes of associated production class always PT co-evolution samples?* This RQ aims to investigate to what extent the assumption (shown in **Assumption .1**) is satisfied.

RQ1.2: *What types of test code updates are not triggered by the changes of their associated production code.* This research question aims to characterize the cases where the assumption (shown in **Assumption .1**) does not hold.

2.1 Data Collection and Cleaning

2.1.1 Project Selection. In terms of project selection, we first follow and clone the available projects used by Wang et al. [34, 81]. These projects are all Apache Software Foundation (ASF) Java projects. Considering that only using the ASF projects to construct a dataset may introduce data bias, thereby missing some *PT co-evolution* practices that do not appear in the ASF projects, we further supplement 1,500 Java projects used by Wen et al. [82]. After filtering out duplicate projects, we obtain a total of 1,952 projects. Next, we select the project using the following criteria: 1) *Project Directory Layout*. We only consider Java projects managed by Maven. Maven projects follow the standard directory layout [31], which allows us to conveniently find its associated production code using *naming convention*; 2) *Popularity*. The number of stars [27] of a repository reflects its popularity on GitHub. Following previous research [82], we select projects with more than ten stars to avoid possible irrelevant/toy projects; 3) *Long change history*. Following previous work [56–58, 75], we exclude projects with a commit history of fewer than three years to ensure that the selected projects are well-maintained; 4) *Compilability*. The selected projects can be compiled, and all test cases can be run successfully, allowing us to obtain the project's test coverage and perform subsequent testing analysis. In order to run the test cases, we need to manually restore the compilation environment of the project, which is time-consuming and resource-intensive. Therefore, we select one project from 1,952 projects in a random manner and verify whether the project satisfies the aforementioned criteria. The project that meets the criteria is retained; otherwise, it is discarded. We utilized a server with two 32-core processors and 256GB memory to compile the projects efficiently. We hired three experienced master students to perform compilation. During the project selection process, we attempted to compile a total of 487 projects but successfully compiled only 150. The entire compilation process, including setting up the environment, debugging errors, configuring test coverage tools, running all unit test cases, and collecting coverage information, required approximately three weeks of effort in total. Despite the considerable effort, applying rigorous project selection criteria is necessary to guarantee the reliability and validity of our empirical study.

Table 1. Statistics of evaluation results for 150 projects.

Metrics	Per Project		
	Mean	**Median**	**Std. Dev.**
Branch Coverage	54.18%	54.00%	25.63%
LOC_{prod}	77980339	10514949	202116309
LOC_{test}	21080109	3894575	49098123
$\frac{LOC_{test}}{LOC_{total}}$	28.69%	27.42%	17.56%
$Change_{test}$	6584	2988	9606
$Change_{prod}$	1448	652	2095
$\frac{Change_{test}}{Change_{total}}$	21.39%	21.13%	12.03%

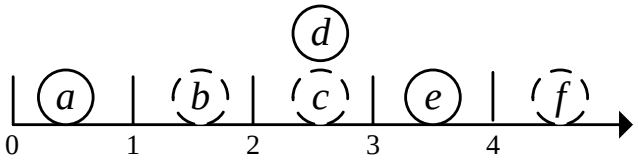

Fig. 3. An example to illustrate production-test change link.

We expect to build a dataset from projects with the best testing efforts [62]. To this end, a preliminary analysis is conducted to evaluate how well these projects are tested. Based on previous studies [62], a project can be considered extensively tested if it meets the following conditions: 1) ratio of changes occurring in test and production code is over 20%; 2) ratio between the amount of test and production code is over 25%; 3) branch coverage determined with Jacoco [29] is over 67%. Due to test failure and compilation errors, we record branch coverage on each project's final version rather than on all versions. The results of three metrics are presented in Table 1. Finally, we retain 44 projects satisfying the above threshold among 150 projects.

2.1.2 Dataset Construction. Following previous work [81], we utilize the following strategies to associate test and production classes: 1) *File Path Matching*. The mined project needs to follow the standard directory layout [31], where `src/main` stores production files and `src/test` stores test files. 2) *File Name Matching*. The naming convention is used to identify the associated production class by removing the "Test" prefix/suffix of test class's name. Subsequently, we construct the *PT co-evolution samples* based on the assumption shown in **Assumption .1**. For a test code change *test*, the production code change that occurs in the same commit of or before *test* will be paired with *test*. Note that previous studies adopt different settings for the time interval T between production code change and test code change. Thus, we relax the restriction on T to explore the amount of noise introduced depending on different time intervals.

Here we introduce an example to illustrate the sample construction. On the timeline (as shown in Figure 3), each circle represents a file modification, and the changes that occur in the same commit are put in the same vertical line. Among these file changes, suppose $\{a, d, e\}$ (solid circles) denotes the updates of production class `Foo`, while $\{b, c, f\}$ (dashed circles) denotes the changes of test class `FooTest`. For b, its updates are traced back to a, due to that b immediately follows a. The change c would be paired with d since production and test modifications are involved in the same commit. Before the change f, there are three production code modifications, i.e, $\{a, d, e\}$, while only e would be paired with f. This is because f is committed after e, which is a later modification of `Foo`. As a result, e should be considered a co-evolved production change of f, rather than a and d.

To ensure that the test code is a unit test class, all file names over the history are stored to check whether the production file exists in the project's historical structure. We keep the code changes that involve at least one *Abstract Syntax Tree* (AST) operation using GumTreeDiff [43] to filter out the changes which only modify comments or formats. Finally, from the 44 projects, we extract 33,567 *co-evolution samples* and analyze 380 randomly sampled *change pairs*, representing the population with 95% confidence and 5% margin of error.

2.2 Analysis of Production-Test Change Pairs

We invite three volunteers with research experiences related to software evolution and 5-years of Java programming experience, including one ph.D student and two master students. Then, the 380

samples are distributed to these three volunteers respectively. During the analysis of *production-test change pairs*, each volunteer needs to manually review 380 samples, summarize the reasons for test code changes, and ultimately agree on the reasons. For each sample, we provide commit messages, code diffs, and changed files to volunteers and allow them to search the corresponding Git repository. We provide instructions to guide volunteers to label each sample as follows: 1) reviewing the commit messages and changed files to understand the change intention and the context of the sample, respectively, 2) checking code diffs and analyzing the correlation between the production-test changes, 3) according to such correlation analysis, judging whether the production changes trigger the test changes, if so, label this sample as positive, otherwise negative, 4) for each negative sample in our empirical study, summarizing the reason of its test change and defining a tag to represent the reason. When analyzing a change pair, each volunteer first defines the tag independently. Subsequently, the volunteer compares it with the shared tags. If similar tags exist, the volunteer adopts the tags that have been defined, otherwise adds the newly-defined tag into the shared tags. Fleiss Kappa [45] is used to measure the overall agreement between these tags. The Kappa value is 0.78, indicating substantial agreement. Once the volunteers did not agree on the tags of a change pair, the pair would be discussed to reach a common decision. In previous empirical works [79, 80], when the manual analysis work is nearing its end, researchers claim that the collected cards/tags reach saturation when new cards/tags do not appear anymore. We found that after analyzing approximately 80% of the change pairs, the volunteers did not add new tags. Thus, motivated by previous works [79, 80], we believe our manual analysis has reached saturation.

After having tagged all samples, we organize the tags into several groups through card sorting [54] to form a taxonomy of test code change patterns. We apply open card sort [73] to sort index cards into categories. Specifically, each card has a title (name of the category). We carefully read the title and the tag (concrete reasons of test code update) to determine whether the tag belongs to this title. All the groups with low-level subcategories will be aggregated into high-level subcategories. In the end, all analytical work was completed in two weeks.

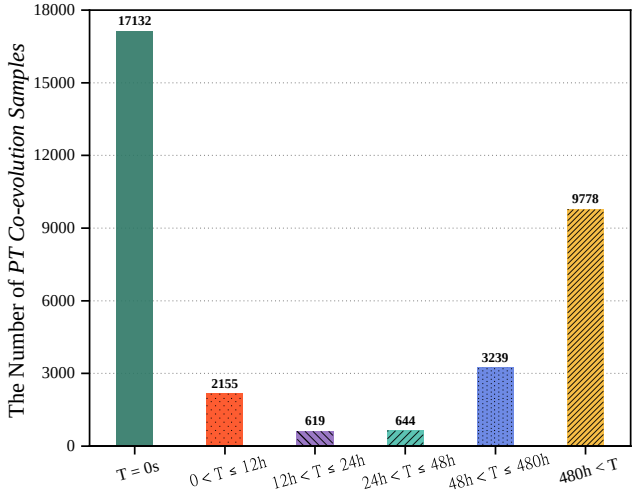

Fig. 4. The distribution of *PT co-evolution samples* constructed based on **Assumption .1**.

Table 2. Sample distributions according to time interval.

Time Interval (T)	Total Samples (N)	Co-evolution (N_{co})	Non-co-evolution (N_{non_co})	N_{co}/N	N_{non_co}/N	$N_{co}/Sum(N_{co})$
$T = 0$	194	172	22	88.66%	11.34%	89.12%
$0 < T \leq 12h$	24	11	13	45.83%	54.17%	5.70%
$12h < T \leq 24h$	7	1	6	14.29%	85.71%	0.52%
$24h < T \leq 48h$	7	1	6	14.29%	85.71%	0.52%
$48h < T \leq 480h$	37	4	33	10.81 %	89.19%	2.07%
$T > 480h$	111	4	107	3.60%	96.40%	2.07%
Sum	380	193	187	50.79%	49.21%	——

2.3 RQ1: Categories of Test Code Update

In this section, we provide the results of our empirical study to answer RQ1.1 and RQ1.2.

First, we count the distribution of 33,567 *PT co-evolution samples* constructed based on **Assumption .1**. The results are shown in Figure 4, where the x-axis represents the time interval T between production code change and test code change, and the y-axis is the number of *PT co-evolution samples*. From Figure 4, we can observe that: consistent with the previous observations [81], when $T = 0s$, the constructed *PT co-evolution samples* are the most. As T increases, the number of postponed *PT co-evolution samples* decreases. In addition, there are some *PT co-evolution candidates* in which the time interval between production changes and corresponding test changes is more than four days. Overall, most test changes occur within a short time interval when the production code changes.

Then, according to constructed 33,567 *PT co-evolution samples*, we adopt a stratified sampling way to randomly select 380 *change pairs* (representing the population with 95% confidence and 5% margin of error). Specifically, we calculate the ratio of samples grouped by the time intervals T to the total samples, and randomly select *change pairs* based on the ratio to ensure that the time distribution properties of the 380 samples are consistent with the original set. The subsequent empirical study is performed based on the selected 380 samples.

RQ1.1: Are production-test change pairs whose test class changes immediately after the changes of associated production class always PT co-evolution samples?

Table 2 provides the number of true positive and false positive samples and shows the distributions of *PT co-evolution samples* (Last column). The *change pairs* are grouped by the time intervals T between the production code change and the test code change. T is set as $0s$, $12h$, $24h$, $48h$, and $480h$.

From Table 2, we can find that: 1) The false positive rates range from 11.34% to 96.40% for different time intervals. 2) When T becomes large, the percentage of *PT co-evolution samples* decreases. Additionally, most *PT co-evolution* occurs within a small time interval (i.e., $T < 12h$). When T exceeds 480 hours, a *production-test change pair* is not likely to be a *PT co-evolution sample*.

Overall, although *PT co-evolution* is common, many test code changes are not triggered by the production code changes. In addition, when production code and test code are modified in a small time interval, they are more likely to construct a *PT co-evolution sample*. However, Table 2 reveals a *production-test change pair* for which the T is small is not necessarily a *PT co-evolution sample*. For example, even for $T = 0s$, the false positive rate is more than 11%. We perform a closer inspection on the *PT co-evolution samples* with $T > 480h$, and find that most of them add or delete files.

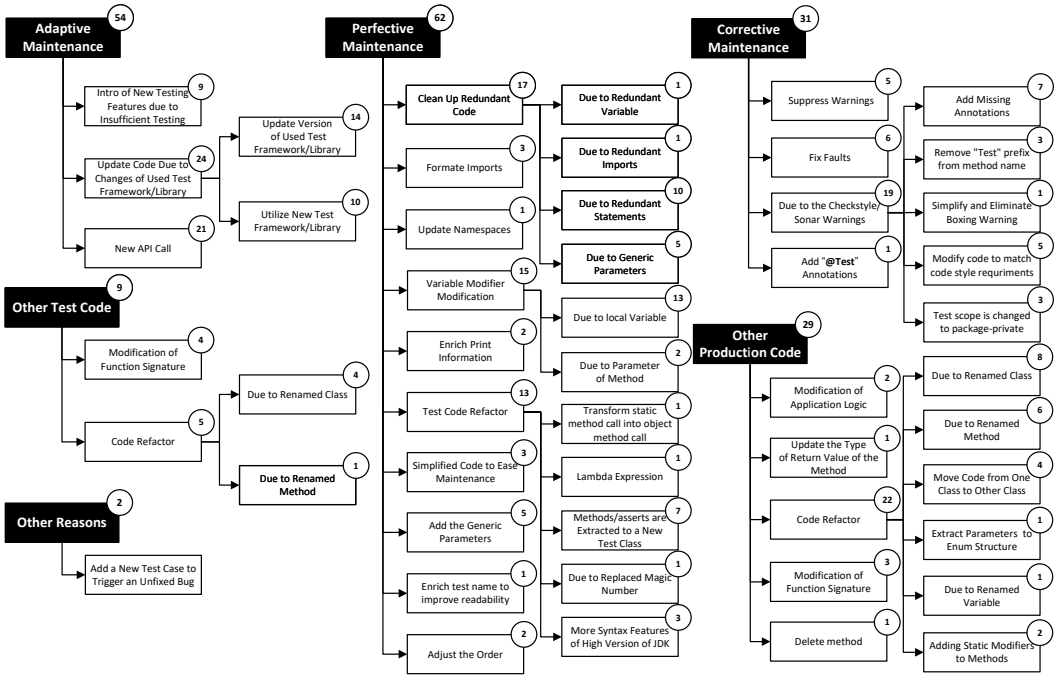

Fig. 5. RQ1.2: Taxonomy of test update unrelated to corresponding production modification.

☛ Depending on the selected time interval T, 11.34%-96.40% of the constructed *PT co-evolution samples* are false positives, i.e., their test code changes are not triggered by their associated production code changes.

RQ1.2: What types of test code updates are not triggered by the changes of their associated production code?

RQ1.1 has confirmed that there exist some *change pairs* where the test code change is not attributable to the change of their associated production code. For these *change pairs*, the assumption (shown in **Assumption .1**) does not hold. To answer RQ1.2, we further analyse these *change pairs* and define a taxonomy, which summarizes the change patterns of these *change pairs* accounting for 6 categories.

As shown in Figure 5, the taxonomy involves **maintenance** [39], **indirectly related production code updates**, **indirectly related test code updates**, and **others**, where maintenance activities are further subdivided into 1) **adaptive maintenance**, which introduces new features into the system and keep software usable in a changed or changing environment; 2) **perfective maintenance**, which improves the performance or maintainability of the system; 3) **corrective maintenance**, which fixes discovered bugs and errors in a program. These three categories are consistent with previous studies, such as [59, 64]. More concrete change patterns are denoted either as intermediate nodes or leaves. The intermediate nodes or leaves are always located to the right of the parent node. For each change pattern, we present the specific number (immediately following the pattern name), along with the descriptions of its representative examples.

1) Adaptive Maintenance (54): The maximum number of test code updates in this category are related to **changes of used test framework/library**, accounting for 44.44%=24/54. One example is from the `Cactoos` project, in which for the test code `CycledTest` [19], the developer updated the

package imports, replacing "junit.Test" with "junit.jupiter.api.Test", and removed the public modifier for each test method. *JUnit Jupiter*, as the critical component of JUnit 5, does not require test methods to be `public` [30]. As a result, the developers modified the test code to meet the feature of the new testing framework. In contrast, the corresponding production code `Cycled` [14] carried out an API replacement due to refactoring activities in the other production code `IterableOf`. Interestingly, we also notice that some test code changes introduce new test methods which are not caused by the associated production code. For example, the test code `EntryStreamTest` [26] added a new test case, while the production code `EntryStream` [25] just update `import` statements. We further checked the commit message and found that the test methods are added to **improve the branch coverage of the project**.

2) Perfective Maintenance (62): In this category, test code is updated to facilitate the future maintenance of test code. One of the change patterns is **cleaning up redundant test code**. In the `Commons-collections` project [20], the production code changes and test code changes were added in the same commit with the message "Remove redundant generic type arguments". However, the test code and the production code modified different objects. For example, the production code `StaticHasher` modified a `TreeSet` type variable, while the test code `StaticHasherTest` modified an `ArrayList` type variable. *PT co-evolution* needs to meet the condition: the test code change triggers or is triggered by the production code change. For the above example, there is no coupling correlation between the variable modified by the test code and that of the production code. In other words, we cannot infer that the modification made by `StaticHasher` to remove the generic parameters of `TreeSet` caused the modification made by `StaticHasherTest` to remove the generic parameters of `ArrayList`. Thus, such production-test change is not an example of *PT co-evolution*, but rather a cleanup activity by developers for code units containing redundant generic type parameters. A more frequent test code change pattern is **modifier changes to local variables and method parameters**. For this pattern, the most common action is adding the "final" modifier. Final variables are read-only by design. When changing a final variable's value, the compiler will throw an error. For example, the test code `ValidateTest` [23] added the "final" modifier to the custom variable, but the corresponding production code `Validate` [16] underwent code refactoring: using `Objects.requireNonNull()` instead of custom conditional statements checking whether an object is empty.

In 13 commits, test code is updated due to **code refactoring activities**, such as, the test code `TestDefaultParametersManager` [15] utilized the Lambda expression to replace anonymous internal classes, making the code more concise and compact. Yet, the function parameters of `DefaultParametersManager` [11] added the final modifier. When multiple test classes contain the same property (e.g., test cases), developers usually extract commonalities and form a new test class inherited or imported by the test classes. For instance, a commit [17] extracted common divide-by-fraction test cases to `CommonTestCases`. Another test code change [22] attempted to encapsulate commonly used asserts (e.g., assertFalse) in a test helper class. Then, `Transform2STest` directly called the encapsulated method rather than writing redundant test asserts. In contrast, the production code `Transform2S` [21] perform a perfective maintenance activity: removing redundant local variables. In addition, this category involves **adding generic type arguments**, **formatting imports**, **reordering test cases**, **enriching print information**, and **enriching test name to improve readability**, etc.

3) Corrective Maintenance (31): The test code updates fixing functional and non-functional faults lie in this category. Overall, many test code modifications are due to Checkstyle [28] or SonarQube [33] warnings. It is worth noting that **Suppress Warnings** is different from the **Fix Checkstyle/SonarQube Warnings** pattern. For example, the test code `TestBinaryAnd` added the annotation of "@SuppressWarnings("unchecked")" to ignore all unchecked warnings coming from

that class [3]. However, the source of warning remains. The corresponding production code [1] added the missing annotation "@Override".

4) Indirectly Related Production Code Updates (29): For a test code `FooTest`, we indicate the other production class that is depended by `FooTest` but is not its associated production code (i.e., class `Foo`) as *indirectly related production code*. This category groups test code changes triggered by the modifications of their indirectly related production code. We observed that test code usually calls methods in non-associated production code to complete testing state initialization. This category includes the cases where the test code updates are triggered by the changes of such methods. For example, `TimeSeries` changed the **function signature**: from `valueOf` to `numOf`, resulting in an update to the test code `InSlopeRuleTest` [13]. The production code `InSlopeRule` located in the same commit [12] modified the import information by replacing "org.ta4j.core.Num.AbstractNum.NaN" with "org.ta4j.core.Num.NaN.NaN".

An example of this category comes from the `Joda-time` project. A commit [4] was created to fix a bug of `Days.daysBetween`. However, the commit did not make any changes to `Days`; instead, the `BaseSingleFieldPeriod` was updated. In the test code `TestDays`, the developers added more test cases for the target method `Days.daysBetween`. We carefully check the implementation details of the target method and observe that it directly called the `BaseSingleFieldPeriod.between`, and then performed a type conversion of the returned result. In essence, the application logic of the target method is realized by `between`. Therefore, the updates of `BaseSingleFieldPeriod` have probabilities to introduce the modifications of `TestDays`. Another example is from the `Commons-csv` project [2], where a commit modified both the production code and the test code. For the production class `CSVParser`, developers only deleted some obsolete code. In contrast, a lot of contents that are related to the `CSVFormat` in the test class are modified. After a closer inspection of `CSVFormat`, we find that it performed a code refactoring activity in this commit, resulting in updates to all code that used this production class.

5) Indirectly Related Test Code Updates (9): The test code that is dependent by the target test code is denoted as *indirectly related test code*. Similar to category 4, test code updates due to the **refactoring activities** of other test code account for the major part, especially for test code with inheritance/dependent relationships. For instance, a commit [8] modified the function signature, leading to the updates of all the test code (such as `GraggBulirschStoerStepInterpolatorTest`) that uses this function. The corresponding production code `GraggBulirschStoerStepInterpolator` [7] changed the function access rights from "protected" to "public".

6) Others (2): The test code updates that are designed to trigger production bugs/errors are placed in this category. In `commons-compress` project, a test method `testCompress264` [6] was added with the commit message "add a (failing) Unit Test". As the comments show, the "COMPRESS-264" issue would be triggered by the added test case. In the next commit, the developer fixed this issue and updated the name of the test case. Obviously, the test code update in this case is not caused by the modifications of associated production code; instead, it would trigger subsequent production code changes. However, the corresponding production code `ZipFile` [5] extracted the custom exception code to a new helper class.

☛ For the *change pairs* where the test code changes are not triggered by the changes of their associated production code, the reasons for their test code updates are varied, from which perfective maintenance accounts for the major part. Besides, other file changes (e.g., indirectly related test and production code) may also cause test code updates. **Assumption .1** attributes test code changes to updates of associated production code without comprehending their changes and understanding their relationship, thus introducing false positives.

3 CO-EVOLUTION IDENTIFICATION METHOD

Our empirical study has confirmed that the assumption of prior studies does not always hold in practice and the methods used by prior work to identify *PT co-evolution samples* may introduce noise. This motivates us to explore whether the introduced noise has a significant impact on the conclusions of the downstream tasks. As opposed to visualisation, the detection task allows us to measure the impact of noise quantitatively and will therefore be adopted as the experimental object. Obviously, conducting studies based on a manually labeled dataset can mitigate this threat. But this requires expensive human resources and time costs. In addition, it is not feasible to rely only on manually labeled data for machine learning and deep learning based methods, which usually require a large number of samples to train good models. Based on the findings from our empirical study, we propose a method to identify *PT co-evolution samples* accurately. We refer to this approach as **CHOSEN**, an identifi**C**ation met**H**od **O**f production-te**S**t co **E**volutio**N**.

3.1 RQ2: Impact of Noise on JIT Detection Methods

We select the JIT obsolete test code detection task (for short, OTCD) [81] as the example task. Prior work showed that developers may forget/ignore updating test code [50, 81]. OTCD can remind developers to update obsolete tests just after they change the production code, which can help developers 1) find the obsolete tests that would not fail and 2) find the obsolete tests that would fail without executing them. Thus, OTCD can be beneficial for improving test coverage and reducing test costs. Our empirical study has confirmed that SITAR's identification method [81] would introduce some noise. Hence, we perform a preliminary experiment to answer the following questions:

RQ2: What is the impact of noise on existing JIT OTCD methods?

Experiment Settings. Since SITAR has no publicly available tools or source code, we implemented SITAR based on their descriptions and obtained similar detection performance on the dataset provided by Wang et al. Following Wang et al. [81], we adopt random forest [53] with default hyper-parameters set by the scikit-learn as the detection model. We make our re-implementation publicly available for further inspection[3].

Data Collection. Since the experiment requires manually labeled datasets, we select three projects on which SITAR achieves good, moderate, and poor detection performance, namely `Flink`, `Log4j2`, and `jsoup` respectively. This selection way can verify that the effect of noise is universal, regardless of the detection method performs well or poorly. Moreover, we observe that most test modification time remains in the initial phase of the project history for the provided dataset [34]. Then, we construct the *product-test change pairs* for each latest version of the project, strictly following the steps in the paper [81].

Data Splitting. We divide the *production-test change pairs* into positive and negative samples using the SITAR's identification method. Unlike SITAR, which randomly splits *change pairs* to obtain the training set and test set, we sort samples in the ascending order of production change commit and put the first 80% samples into the training set; meanwhile, shuffle the remaining 20% samples into the test set. This way ensures all production code updates in the training set occurred before those in the test set, similar to prior studies [49, 87]. Subsequently, we randomly sample 132, 262, and 347 pairs from the test set for `jsoup`, `log4j2`, and `Flink` respectively, which represent the population with a 95% confidence level and 5% margin of error. Given that our empirical study has confirmed that SITAR's identification method introduces some noise, we refer to these sampled test sets as **noise test sets**. To establish the ground truth, we manually re-label the sampled noise test set and refer to the resulting test sets as the **annotated test set**.

[3]https://anonymous.4open.science/r/CHOSEN-DE81

Table 3. Performances of detection method trained on noise training set, tested on Noise *vs.* Annotated test set.

Projects	**Metrics**	Annotated test set (x)	Noise test set (y)	*Error* ($\lvert y-x \rvert$)	*p-value* (y *vs.* x)
Jsoup	Acc.	58.64%	54.55%	4.09%	< 0.05
	Prec.	38.63%	51.96%	13.33%	< 0.001
	Rec.	50.24%	45.81%	4.43%	< 0.05
	F1.	43.68%	48.69%	5.01%	< 0.001
Log4j2	Acc.	60.73%	57.29%	3.44%	< 0.05
	Prec.	51.11%	57.94%	6.83%	< 0.05
	Rec.	67.17%	60.22%	6.95%	< 0.05
	F1.	58.05%	59.06%	1.01%	< 0.001
Flink	Acc.	66.34%	55.85%	10.49%	< 0.05
	Prec.	72.52%	77.56%	5.04%	< 0.001
	Rec.	57.69%	48.20%	9.49%	< 0.001
	F1.	64.26%	59.45%	4.81%	< 0.001

Evaluation Metrics. Following the previous work [81], we compare SITAR's performance on the noise test set and the annotated test set in terms of accuracy, precision, and recall, and F1-score to quantify the effect of noise on the detection method. Considering the randomness of random forest, we ran each experiment 10 times to obtain averages for analysis. In addition, we compute *p-value* (probability value) examine the significance of the performance differences.

Results. Table 3 presents the average performances of SITAR on the noise/annotated test set, where the last column gives the calculated p-values. On the annotated test sets of the three projects, SITAR's precision decreases by at most 13.33% and recall increases. The p-values are less than 0.05 in terms of all evaluation metrics, implying that the differences in detection performance are significant on these test sets. The above observations can be explained as follows. The precision metric is computed by $\frac{TP}{P}$, where TP and P refer to the number of true positives and the total number of positive predictions, respectively. Since the classifier stays unchanged, P is equal for both test sets. After relabeling the test set (i.e., removing noise), the size of positive samples would become smaller than before, leading to a decrease in terms of precision.

Explanation. It is important to note that there is a discrepancy between our obtained detection performance and published results [81]. This is because: 1) the sizes of our used datasets are different. Our newly constructed dataset contain the latest commits of each project, and thus contains more samples. 2) SITAR split their dataset randomly, which may lead to information leakage, whereas we use the production modification time to split our dataset.

👉 The noise causes detection performance on the testing set to be significantly different from its actual performance, with an average difference of 6.01%, 8.40%, 6.96%, 3.61% in accuracy, precision, recall, and F1-score respectively.

3.2 Proposed Approach

3.2.1 Problem Formulation. Given a *production-test change pair* $\langle change_p, change_t \rangle$, where $change_t$ is the test code change and $change_p$ is the latest change of the associated production code that occurs in the same commit of or before $change_t$, then, **CHOSEN** determines whether $change_t$ is triggered by $change_p$, i.e., a positive/*co-evolution sample*.

3.2.2 Usage Scenario. Practitioners and researchers can use **CHOSEN** to construct *PT co-evolution samples* accurately. Such samples are the basis of or are useful for various downstream tasks.

JIT obsolete test code detection [81]. CHOSEN can be used to assist developers in performing JIT obsolete test code detection [81]. The test set built from **CHOSEN** can measure the performance of models more accurately. The dataset constructed by **CHOSEN** can train a better detection model.

Mining co-evolution scenarios [42, 55, 88]. Based on the *PT co-evolution samples* constructed by **CHOSEN**, developers and test engineers can explore different *PT co-evolution* scenarios, including synchronous and phased, to gain insight into the testing process. For detailed instances and explanations of synchronous and phased *PT co-evolution*, please refer to the literature [88].

Fault localization [72]. Considering the bug fixing cases, these often occur with additions/-modifications of tests (to validate the repair action). Thus, when a test execution fails, **CHOSEN** can roughly locate the class-level fault production code by identifying the production code that often evolves in concert with the failure test case.

Linking production and test code [83]. CHOSEN has the potential to establish relevance links between tests and code units. Given a series of candidate links $\{\langle t, p_1\rangle, \langle t, p_2\rangle, ..., \langle t, p_n\rangle\}$, where t refers to the test code and p_i represents various production code, developers can utilize **CHOSEN** to mine the amount of co-evolution of production and test code for each candidate link. Intuitively, the greater the number of co-evolution, the greater the likelihood of correlation, as the co-evolution of tests and code units reflects a probable link between them. The rationale is that the *PT co-evolution* sample refers to a pair of a test code change and a production code change where the test code change triggers or is triggered by the production code change, which means that there is a relevance coupling between them. For example, when developers fix bugs in production code, they usually introduce or modify test code to evaluate the fixed parts. Similarly, when developers implement a new feature or update an existing one, they may introduce or update test code related to that feature simultaneously or later. Even if identified *PT co-evolution* by **CHOSEN** is irrelevant (i.e., false positives), these should be eliminated by the number of *PT co-evolution* as it is unlikely that the same irrelevant changes will be repeated. Thus, the more frequently t and p_i co-evolve, the stronger the relevance coupling between them. In other words, p_i is more likely to be the associated production code of t.

3.2.3 Approach. Our method includes two stages, namely ***initial label estimation*** and ***multi-strategy-based label determination***.

Stage 1-Initial label estimation: Based on our empirical observations, when a test class is updated within 12 hours (94.82% *PT co-evolution samples* are less than $12h$) after modifying its associated production code, **CHOSEN** regard it as an initial positive sample. Otherwise, it is an initial negative sample.

Stage 2-Multi-strategy based label determination: Through stage 1, an initial label is assigned to each *production-test change pair*. Subsequently, we propose five strategies to adjust the label of *change pair* based on the experience gained from our empirical studies.

Strategy 1: *The type of the associated production code change or the test code change is non-modification type and there are no production/test changes between their commits* $\longrightarrow$ *'POSITIVE'*. Intuitively, the deletion and addition of the test file should be closely related to the associated production file. Hence the non-modification type (e.g., `ADD` or `DELETE`) of *production-test change pairs* are likely to be *PT co-evolution samples*. Our empirical investigation also confirms this intuition: Among the four non-modification change pairs with $T > 12h$, all the change pairs are

PT co-evolution samples. For more details, please refer to our empirical study dataset[4]. Besides, we have added a constraint with there are no production/test changes between their commits

Strategy 2: ***There are additional production code modifications between production code change commit and test code change commit ⟶ 'NEGATIVE'.*** As shown in Table 2, the Ts of 94.82% *co-evolution samples* are committed less than 12 hours apart. Once there are multiple production code modifications before the test updates, it is reasonable to consider that earlier production change is insufficient to introduce test modifications.

Strategy 3: ***The changes of the production or test code involve only import changes, and the intersection of import modification is empty ⟶ 'NEGATIVE'.*** As shown in Figure 5, the test code change patterns involve some modifications related to import activities, such as **updates of used test libraries/frameworks**, **removal of redundant imports**, and **formatting imports**. We observe that a commit might updates the import statements in multiple files. However, these modifications are not related to each other, such as [18]. Hence, for Strategy 3, we aim to adjust the label of *production-test change pairs* containing changes of import statements. **CHOSEN** needs first to ensure the production or test code only modifies the import statements. Once the intersection regarding import changes is empty, **CHOSEN** identify this *production-test change pair* as a negative sample.

Strategy 4: ***There is no semantic relevance between the changes of the production and test code.⟶ 'NEGATIVE'.*** Generally, test code is responsible for validating whether the production code behaves as expected, which means that test code should be semantically related to the associated production code. For example, adding a new test method to cover a production method or updating test asserts to validate production code changes. Hence, we consider the initial positive *change pairs* with no semantic relevance as *non-co-evolution samples*.

To measure the semantic relevance between the test and the production code changes, we collect the changed contents of the class body, remove the changes of comments or format as well as the import statement, and tokenize collected changed contents by spaces and punctuation. **CHOSEN** regards a *production-test change pair* as a negative sample, when the tokenized change set of production and test code have nothing in common.

Strategy 5: ***The type of modification involves annotations, modifiers, refactoring operations ⟶ 'NEGATIVE'.*** The empirical study shows that certain modification activities should not cause *PT co-evolution*. Among 22 false positives with $T = 0s$, 10 (45.5%) samples are the change pattern of "**Variable Modifier Modification**" (as shown in Figure 5). Additionally, the maintenance activities can update the annotations in the test code, such as **adding some missing annotations** and **suppressing warnings**. Moonen et al. have indicated that even though refactorings are behavior preserving, they potentially invalidate tests [65]. Hence, we consider a *change pair* as a negative sample when the test code only contains refactoring actions. We utilize the RefactoringMiner [77, 78] to mine refactorings of two given revisions of the project. Compared to other refactoring detection tools (such as RefDiff [71]), RefactoringMiner has some unique features: 1) it does not depend on the similarity threshold of the code, 2) it supports low-level refactoring that occurs within the method body. 3) it can detect nested refactoring operations in a single commit. The experimental results also show that RefactoringMiner achieves the highest average precision and recall in identifying refactoring operations [77]. For the test code change, we first extract its modification list. If the modification list is empty after removing all the modifications related to refactoring operations, we classify the change pair as a negative sample.

Algorithm 1 provides details of Stage 2. We suppose a *production-test change pair* $\{pro_i, test_i, tag_i\}$, where the elements in the triple refer to the associated production code change, test code

[4]https://anonymous.4open.science/r/CHOSEN-DE81

Algorithm 1: IDENTIFICATION METHOD

Input: *Samples* /*The dataset to be filtered*/
δ /*An predefined parameter*/
Output: pos_{sample}, neg_{sample}={}, {}

1: **for** (each element $\langle pro_i, test_i, tag_i \rangle$, where $i \in [1, n]$) **do**
2: **if** ($tag_i \in$ {NEGATIVE} **and** $type_{pro}$ or $type_{test} \notin$ {MODIFY} **and** HasNonProBetween (pro_i,$test_i$) **then**
3: $pos_{sample} \leftarrow \langle prod_i, test_i,$'POSITVE'$\rangle$
4: **end if**
5: **if** ($tag_i \in$ {POSITVE}) **then**
6: **if** (HasProBetween(pro_i,$test_i$)) **then**
7: $neg_{sample} \leftarrow \langle pro_i, test_i,$ 'NEGATIVE'$\rangle$
8: **end if**
9: **if** (getChangeType(pro_i) or getChangeType($test_i$) $\in$ {Import}) **then**
10: $Import_{pro} \leftarrow$ getDiffToken(pro_i)
11: $Import_{test} \leftarrow$ getDiffToken($test_i$)
12: **if** getIntersectionRatio($Import_{pro}, Import_{test}$) == 0 **then**
13: $neg_{sample} \leftarrow \langle pro_i, test_i,$'NEGATIVE'$\rangle$
14: **end if**
15: **end if**
16: $Content_{pro} \leftarrow$ getDiffToken(pro_i)
17: $Content_{test} \leftarrow$ getDiffToken($test_i$)
18: **if** (getIntersectionRatio($Content_{pro}, Content_{test}$) $\leq \delta$) **then**
19: $neg_{sample} \leftarrow \langle pro_i, test_i,$'NEGATIVE'$\rangle$
20: **end if**
21: **if** (getChangeType(pro_i) or getChangeType($test_i$) $\in$ {Modifier, Annotation} **and** getChangeType($test_i$)) $\in$ {Refactoring}) **then**
22: $neg_{sample} \leftarrow \langle pro_i, test_i,$'NEGATIVE'$\rangle$
23: **end if**
24: **end if**
25: **end for**
26: **Return** pos_{sample}, neg_{sample}

change, and the label estimated by Stage 1, respectively. We collect the changed lines for $test_i$, indicated by $change_{test}$. When the tag_i is NEGATIVE, **CHOSEN** checks the file change type of pro_i and $test_i$. Once one of the change types is not the MODIFY and the *change pair* has no other production/test changes between their commits, it is identified as a *PT co-evolution sample* (line 2-4). We consider those initial positive samples satisfying Strategy 2 as negative samples (line 6-8). The remaining positive samples would be applied to Strategy 3, Strategy 4, and Strategy 5. Specifically, we use the GumTree to extract the edit actions of pro_i and $test_i$, denoted $action_{pro}$ and $action_{test}$ respectively. When the $action_{pro}$ or $action_{test}$ only involves import edits and the intersection of import edits is empty, we identify the *change pair* as a negative sample (line 9-15). We exclude from the *co-evolution* set those samples owing the non-semantic relevance using Strategy 4 (line 16-20). When $action_{pro}$ or $action_{test}$ is caused by annotation or modifier activities, we adjust the *change pair*'s label. Furthermore, RefactoringMiner [77, 78] is utilized to mine refactorings of two given revisions of a project. RefactoringMiner can obtain the set of changed rows (denoted Ref_{test})

due to refactoring. If $change_{test} \subseteq Ref_{test}$, the *change pair* can be considered a negative sample (line 21-23).

4 EVALUATION

We conduct a series of experiments to evaluate the performance of our proposed identification method. This section provides details about the experimental settings and results.

4.1 Research Questions

Given a *production-test change pair*, we expect the **CHOSEN** can precisely determine whether the test code co-evolves with the production code. Thus, it is required to examine the effectiveness of **CHOSEN** in identifying PT co-evolution samples. To demonstrate that our proposed method remains available for downstream tasks, we further explore the performance of our approach in other tasks, e.g., JIT OTCD. The experimental studies help us answer the following four research questions:

- *RQ3: How effective is CHOSEN at identifying PT co-evolution?*
- *RQ4: How useful is CHOSEN in JIT OTCD?*
  - ◇ *RQ4.1: Could the test set constructed by **CHOSEN** be used to measure the performance of the detection method more accurately?*
  - ◇ *RQ4.2: Could the dataset constructed by **CHOSEN** be used to train a better OTCD model?*
  - ◇ *RQ4.3: How does CHOSEN perform on other projects?*

4.2 RQ3: The Effectiveness of CHOSEN

4.2.1 Baselines. We implement two baselines to compare with our method:

Random guess (RG). Random guess is a particular baseline that directly employs the data distribution obtained from our empirical study to identify *PT co-evolution samples*. This baseline is usually used when there is no previous method for the research question [84]. Specifically, the ratio of co-evolution samples in our empirical study is 51.05%, and then, the RG baseline will randomly select approximately 51.05% of *production-test change pairs* as positive samples. Note that the AUC of RG is always 0.5 [84].To reduce the bias of randomly selecting, we run each experiment 10 times and report the average results.

SITAR's identification method (SITAR_IM), following the Assumption .1. Recently, Wang et.al [81] proposed SITAR, a machine-learning-based approach, to detect whether a piece of test code should be updated when developers modify the production code. SITAR consists of two components: 1) the construction of *PT co-evolution samples* (positives) and *PT non-co-evolution samples* (negatives); 2) designing several structural code features and training a detector to detect obsolete test code, based on the mined samples. SITAR uses the following strategies to identify *PT co-evolution samples*: 1) positive sample, if a test class is updated within 48 hours since its associated production class is changed; 2) negative sample, the test code has not changed within 480 hours. Since Wang et.al [81] does not name the component of identifying *PT co-evolution samples*, we name it as *SITAR_IM* for the sake of description. In this paper, unless otherwise specified, SITAR refers to the obsolete test code detection method, and *SITAR_IM* refers to the method for identifying *PT co-evolution samples* in SITAR.

4.2.2 Experimental Setup. The design and settings of the experiments are described in this section.

Data Collection. We propose **CHOSEN** based on 380 labeled samples used in empirical study. Evaluating **CHOSEN** on these observed samples would lead to overfitting, which is undesirable. Thus, following the Section 2.1, we re-select and label another set of 380 samples from the 33,567 *production-test change pairs* while ensuring that there are no overlap or duplicate samples between

Table 4. Effectiveness of **CHOSEN** *vs.* RG and SITAR_IM.

Method	Acc.	Positive			Negative			AUC
		Prec.	Rec.	F1.	Prec.	Rec.	F1.	
RG	48.68%	46.39%	49.72%	48.00%	51.08%	47.74%	49.35%	50.00%
CHOSEN	92.89%	89.29%	96.69%	92.84%	96.74%	89.45%	92.95%	93.07%
Improvement	90.81%	92.46%	94.44%	93.41%	89.41%	87.37%	88.35%	86.13%
SITAR_IM	81.63%	73.95%	99.44%	84.82%	99.05%	62.65%	76.75%	81.04%
CHOSEN	92.71%	89.18%	97.74%	93.26%	97.32%	87.35%	92.06%	92.54%
Improvement	13.57%	20.59%	-1.70%	9.95%	-1.75%	39.42%	19.95%	14.19%

Table 5. The confusion matrix of **CHOSEN** vs. RG and SITAR_IM.

Actual Classification of 380 samples	**CHOSEN** Prediction		RG Prediction	
	Positive	Negative	Positive	Negative
Positive	175	6	90	91
Negative	21	178	104	95
Actual Classification of 343 samples	**CHOSEN** Prediction		SITAR_IM Prediction	
	Positive	Negative	Positive	Negative
Positive	173	4	176	1
Negative	21	145	62	104

these 380 samples and that of our empirical study. These 380 samples will be used as ground truth to verify the effectiveness of **CHOSEN** and RG. Notably, SITAR_IM does not provide an identification strategy for *change pairs* with $48h < T < 480h$. Therefore, we filter the 380 samples, retaining only the *change pairs* that satisfy SITAR_IM's identification requirements. In summary, we re-collect and label 380 samples for RQ3. We compare the effectiveness of **CHOSEN** and RT on the 380 samples while comparing the effectiveness of **CHOSEN** and SITAR_IM on the filtered dataset consisting of 343 samples.

Evaluation Metrics We use accuracy, precision, recall, F1-score, and area under the precision-recall curve (AUC) [69] as evaluation measures. The accuracy directly reflects the amount of noise introduced. The precision and recall allow us to evaluate the proportion of true positives among all positive predictions and all positive examples retrieved, respectively. Since precision and recall are trade-offs, the F1-score is a valuable metric as it enables us to figure out which methods handle the trade-off best. Finally, we employ the area under the precision-recall curve (AUC) since it provides a threshold-independent perspective of each technique's performance.

4.2.3 Answer to RQ3. ***How effective is CHOSEN at identifying PT co-evolution?*** Table 4 provides the performance comparison of **CHOSEN** and the baseline methods, including RG and SITAR_IM, when determining whether a change pair is a *PT co-evolution sample* (*Positive*: co-evolution pair, *Negative*: non-co-evolution pair). Table 5 reports the confusion matrix of **CHOSEN**, RG and SITAR_IM. In Table 4, we display the precision, recall, F1-score of compared methods on both positive and negative classes. Since accuracy and AUC have identical values for both categories, we present them in a separate column. Accuracy measures the overall correctness of the classifier's predictions, regardless of the class label, and therefore is the same for both classes [76]. Additionally, to calculate AUC [48] metrics for different identification methods, we assigned a

prediction probability of 1.0 to positive predictions and 0.0 to negative predictions, as the identification methods, including **CHOSEN**, do not provide prediction probabilities for the identification results. Using the assigned prediction probabilities and information in the confusion matrix (Table 5), we determined AUC values for each identification method. Our results indicate that AUC values are the same for both positive and negative categories in this paper. In terms of identifying *PT co-evolution samples*, results show that **CHOSEN** achieves 93.07% of AUC and improves over RG by 86.13% and 93.41% in terms of AUC and F1-score, respectively. Because SITAR_IM does not identify *production-test change pairs* with $48h < T < 480h$, we filter out such *change pairs* from the 380 samples to compare with SITAR_IM. Table 4 shows that, although **CHOSEN** performs slight worse on recall, the F1-score and AUC of **CHOSEN** are 93.26% and 92.54%, respectively, improving by 9.95% and 14.19%. The above results can be explained as follows. SITAR_IM identifies all *PT co-evolution samples* that occur within 48 hours as positive samples, thereby obtaining a high recall value. Its disadvantage is that it does not distinguish *PT co-evolution samples* and *non-co-evolution samples* when the time interval is short, introducing noise. Since **CHOSEN** identify *PT co-evolution samples* case by case, it can achieve higher precision by better distinguishing *PT co-evolution samples* and *non-co-evolution samples* when the time interval is less than 48h. In terms of identifying *PT non-co-evolution samples*, results show that **CHOSEN** improves over RG by 88.35% in terms of F1-score. Although **CHOSEN** performs slight worse on precision than SITAR_IM, the recall and F1-score of **CHOSEN** are 87.35% and 92.06%, respectively, improving by 39.42% and 19.95%. The above results can be explained as follows. SITAR_IM directly identifies the production-test change pair with $T > 480h$ as *PT non-co-evolution sample*. Our empirical study has confirmed that when the T exceeds 480 hours, the change pair has a high probability of being *PT non-co-evolution sample* (more than 95%), so SITAR_IM has a high precision value. In contrast, the recall value of SITAR_IM is much lower than that of **CHOSEN**, since SITAR_IM does not consider the negative samples that occurred within 48 hours.

☛ Our **CHOSEN** can effectively identify *PT co-evolution*. It improves SITAR_IM and RG by 13.57% and 90.81% respectively in terms of accuracy, by 9.95% and 93.41% respectively in terms of F1-score. Moreover, **CHOSEN** can effectively identify *PT non-co-evolution*. It improves SITAR_IM and RG by 39.42% and 87.37% respectively in terms of recall, by 19.95% and 88.35% respectively in terms of F1-score.

4.3 RQ4: The Usefulness of CHOSEN in JIT OTCD

4.3.1 Experimental Setup. We provide more details of the design and settings of the experiments in this section.

Data Collection. To answer RQ4.1 and RQ4.2, we have followed the dataset used in RQ2 (more descriptions in **Data Splitting** of Section 3.1). As for RQ4.3, we select the 14 projects widely used in *PT co-evolution* related research studies [35, 81, 83]. Table 6 summarizes the detailed information of these 14 projects. We construct *production-test change pairs* for each project and undersample the majority class to balance the sample label distributions. We adopt the same way of splitting data as mentioned in Section 3.1. We adopt SITAR_IM to divide dataset to obtain ***noise training set*** and ***noise test set***. Then, the ***noise training set*** and ***noise test set*** after applying our proposed method are indicated as ***CHOSEN training set*** and ***CHOSEN test set***, respectively. A dataset with the prefix ***annotated*** means that the dataset has been relabeled manually.

Evaluation Metrics. Following the previous work [81], we compare SITAR's performance in terms of accuracy, precision, recall, and F1-score to quantify the usefulness of identification methods. Considering the randomness of random forest, we ran each experiment 10 times to obtain averages for analysis. In addition, we compute both *p-value* (probability value) and *effect size* to

examine the significance of the performance differences. Specifically, we utilize the paired Mann Whitney-Wilcoxon test [36] to verify whether there are statistically significant differences among the investigated methods. The p-value is less than 0.05, implying a significant difference between the two compared methods. Meanwhile, the non-parametric Cliff's delta effect size is used to evaluate the amount of the difference[5] between the two approaches.

4.3.2 Answer to RQ4.1. ***Could the test set constructed by CHOSEN be used to measure the performance of the detection method more accurately?*** We evaluate SITAR's detection performance on the noise/**CHOSEN** test set. The detection performance on the annotated test set is considered as the ground truth. Table 7 shows that the noise test set leads to average performance errors of 6.01%, 8.40%, 6.96%, and 3.61% in terms of accuracy, precision, recall, and F1-score, respectively. In contrast, those errors are only 0.68%, 0.99%, 0.62%, and 0.76% on the test set labeled by **CHOSEN**. Statistical results show the p-value < 0.001 and the *cliff's delta* $= 0.94$, meaning that the performance errors caused by the noise test set are significantly greater than those caused by the **CHOSEN** test set. Therefore, compared to SITAR_IM, **CHOSEN** can be used to construct better test set, which can more accurately measure the performance of OTCD models.

4.3.3 Answer to RQ4.2. ***Could the dataset constructed by CHOSEN be used to train a better OTCD model?*** We trained the detection model on the noise/**CHOSEN** training set, respectively, and tested them on the annotated test set. Table 8 lists the accuracy, precision, recall, and F1-score in percentage. The improvements in terms of accuracy range from 1.30% to 18.03%, and the average improvement is 12.00%. The range of precision improvements is between -1.71% and 30.39%. Meanwhile, START obtains the 13.50% average improvement in terms of F1-score. Overall, the **CHOSEN** training set can be used to train a better detection model than the noise training set. One possible reason is that the distribution of the dataset constructed by **CHOSEN** is more consistent with the true distribution. For `Flink`, **CHOSEN** may introduce more noise when identifying *change pairs* of $12h < T \leq 48h$, resulting in a slight decrease in precision.

[5]We use the following mapping for the values of the delta that are less than 0.147, between 0.147 and 0.33, between 0.33 and 0.474 and above 0.474 as "*Negligible (N)*", "*Small (S)*", "*Medium (M)*", "*Large (L)*" effect size, respectively [40].

Table 6. Selected projects for evaluation

Project	From	#Commits	#Files	#KLoC
ActiveMQ	[81]	10951	4322	408.5
BioJava	[35, 81]	6540	1298	145.1
CloudStack	[81]	34641	5811	683.0
Math	[81]	6622	1165	149.7
dnsjava	[81, 83]	2031	280	33.1
Geode	[81]	10735	9018	1381.1
Gson	[62, 81]	1485	208	25.3
James	[81]	12939	5447	446.6
JRuby	[81]	50346	1750	264.0
PMD	[62, 81]	17768	3289	348.5
Storm	[81]	10462	2434	300.6
Usergrid	[81]	10954	2097	175.2
IzPack	[35, 81]	5667	1112	105.3
Zeppelin	[81]	5012	916	158.0

Table 7. Performances of detection method trained on noise training set, tested on Noise *vs.* **CHOSEN** test set.

Projects	**Metrics**	Annotated test set (x)	Noise test set (y)	**CHOSEN** test set (z)	*Error* ($\|y-x\|$)	*Error* ($\|z-x\|$)	*p-value* (y *vs.* x)
Jsoup	Acc.	58.64%	54.55%	59.32%	4.09%	0.68%	< 0.05
	Prec.	38.63%	51.96%	38.61%	13.33%	0.02%	< 0.001
	Rec.	50.24%	45.81%	51.71%	4.43%	1.47%	< 0.05
	F1.	43.68%	48.69%	44.21%	5.01%	0.53%	< 0.001
Log4j2	Acc.	60.73%	57.29%	59.47%	3.44%	1.26%	< 0.05
	Prec.	51.11%	57.94%	48.52%	6.83%	2.59%	< 0.05
	Rec.	67.17%	60.22%	67.25%	6.95%	0.08%	< 0.05
	F1.	58.05%	59.06%	56.37%	1.01%	1.68%	< 0.001
Flink	Acc.	66.34%	55.85%	66.25%	10.49%	0.09%	< 0.05
	Prec.	72.52%	77.56%	72.87%	5.04%	0.35%	< 0.001
	Rec.	57.69%	48.20%	57.38%	9.49%	0.31%	< 0.001
	F1.	64.26%	59.45%	64.20%	4.81%	0.06%	< 0.001
Average					**6.24%**	**0.76%**	

Table 8. Performances of detection method trained on Noise *vs.* **CHOSEN** training set, tested on annotated test set.

Projects	**Metrics**	Noise training set	**CHOSEN** training set	Improvement
Jsoup	Acc.	58.64%	68.41%	16.66%
	Prec.	38.63%	50.37%	30.39%
	Rec.	50.24%	56.19%	11.84%
	F1.	43.68%	53.12%	21.62%
Log4j2	Acc.	60.73%	71.68%	18.03%
	Prec.	51.11%	63.05%	23.36%
	Rec.	67.17%	70.95%	5.63%
	F1.	58.05%	66.77%	15.02%
Flink	Acc.	66.34%	67.20%	1.30%
	Prec.	72.52%	71.28%	-1.71%
	Rec.	57.69%	62.75%	8.77%
	F1.	64.26%	66.74%	3.86%

4.3.4 Answer to RQ4.3. ***How does CHOSEN perform on other projects?*** In this section, we present the **CHOSEN**'s performance on other projects. Since it is too costly to manually label a test set for each project and Section 4.3.2 has demonstrated that **CHOSEN** only lead to few performance errors ($< 1\%$), we use **CHOSEN** to label the test set for each project.

Table 9 reports the accuracy, precision, recall, and F1-score for all projects. We also provide the performance improvements in terms of each metric, where positive improvements are marked in grey. Results show that the detection models trained on **CHOSEN** training sets achieve the average improvements of 8.32%, 13.85%, and 9.31% in terms of accuracy, precision, and F1-score, respectively. Compared with the noise training set, the **CHOSEN** training set can help the detection model achieves better recall in 11 out of the 14 projects. The statistical analysis displays that

the p-values are less than 0.05, meaning that the **CHOSEN** training set is significantly better than the noise training set on all metrics. The effect-size results show that the dataset constructed by **CHOSEN** brings large performance differences on accuracy, medium performance differences in precision and small performance in F1-score.

Table 9. Performances of **CHOSEN** in 14 programs

Project	Noise training set (x) *vs.* **CHOSEN** training set (y)	**Acc.**	**Prec.**	**Rec.**	**F1.**
ActiveMQ	x	57.45%	55.10%	49.13%	51.94%
	y	67.82%	68.75%	57.48%	62.61%
	Improvement	**18.05%**	**24.77%**	**17.00%**	**20.54%**
BioJava	x	71.59%	46.92%	43.13%	44.95%
	y	77.41%	63.36%	37.95%	47.47%
	Improvement	**8.13%**	**35.04%**	**-12.01%**	**5.61%**
CloudStack	x	71.44%	61.98%	60.00%	60.97%
	y	71.56%	62.06%	60.65%	61.35%
	Improvement	**0.17%**	**0.13%**	**1.08%**	**0.61%**
Math	x	73.33%	62.80%	68.68%	65.61%
	y	74.19%	64.19%	68.83%	66.43%
	Improvement	**1.17%**	**2.21%**	**0.22%**	**1.25%**
dnsjava	x	64.42%	18.25%	43.33%	25.68%
	y	70.96%	23.99%	47.33%	31.84%
	Improvement	**10.15%**	**31.45%**	**9.23%**	**23.98%**
Geode	x	67.92%	78.01%	68.37%	72.87%
	y	69.94%	81.43%	67.77%	73.97%
	Improvement	**2.97%**	**4.38%**	**-0.88%**	**1.51%**
Gson	x	55.27%	51.99%	46.54%	49.11%
	y	66.25%	68.80%	50.38%	58.17%
	Improvement	**19.87%**	**32.33%**	**8.25%**	**18.43%**
IzPack	x	62.37%	58.20%	59.71%	58.95%
	y	65.79%	61.95%	63.88%	62.90%
	Improvement	**5.48%**	**6.44%**	**6.98%**	**6.71%**
JRuby	x	48.33%	28.50%	36.36%	31.95%
	y	48.79%	30.46%	42.27%	35.41%
	Improvement	**0.95%**	**6.88%**	**16.25%**	**10.80%**
PMD	x	64.13%	64.79%	56.79%	60.53%
	y	67.03%	68.63%	58.76%	63.31%
	Improvement	**4.52%**	**5.93%**	**3.47%**	**4.60%**
Storm	x	62.50%	57.78%	53.39%	55.50%
	y	72.50%	69.30%	66.61%	67.93%
	Improvement	**16.00%**	**19.94%**	**24.76%**	**22.40%**
Usergrid	x	65.57%	41.57%	67.37%	51.41%
	y	74.76%	52.67%	65.62%	58.44%
	Improvement	**14.02%**	**26.70%**	**-2.60%**	**13.66%**
Zeppelin	x	60.22%	42.55%	63.37%	50.91%
	y	69.59%	52.71%	64.31%	57.94%
	Improvement	**15.56%**	**23.88%**	**1.48%**	**13.79%**
James	x	70.18%	69.71%	70.00%	69.85%
	y	72.59%	72.12%	72.52%	72.32%
	Improvement	**3.43%**	**3.46%**	**3.60%**	**3.53%**
Avg.	x	63.91%	52.73%	56.16%	53.59%
	y	69.23%	60.03%	58.88%	58.58%
	Improvement	**8.32%**	**13.85%**	**4.86%**	**9.31%**
Statistical results	p-value	6.1035×10^{-5}	6.1035×10^{-5}	0.0148	5.4697×10^{-4}
	cliff's delta	0.51	0.34	0.14	0.28
	Effect-size	*L*	*M*	*N*	*S*

☛ *The test set constructed by **CHOSEN** measures the performance of OTCD models more accurately with an average error of 0.76%. The dataset constructed by **CHOSEN** can train a better detection model, with performance improvements of 12.00%, 17.35%, 8.75%, and 13.50% in terms of accuracy, precision, recall, and F1-score, respectively.*
☛ *The dataset constructed by **CHOSEN** improves the performance of the detection model on other projects, where the average accuracy, precision, recall, and F1-score of the model improve by 8.32%, 13.85%, 4.86%, and 9.31% respectively.*

5 DISCUSSION

In this section, we further discuss the definition of **Assumption .1**, the limitation of empirical study, the limitation of our approach, the implications for practitioners and researchers, and the threats to the validity of this work.

5.1 The definition of Assumption .1

The definition of **Assumption .1** is based on prior research [42, 55, 81, 88] that employs time-based commit analysis to identify *PT co-evolution samples*. Existing work argues that *PT co-evolution* should occur together in one commit [42, 55, 88], or in a short time interval [81]. Actually, **Assumption .1** was formulated to be able to mine *PT co-evolution samples*, since automatically determining *PT co-evolution* is a non-trivial task that requires precisely comprehending production and test code changes. Furthermore, **Assumption .1** is reasonable because it aligns with the idea that if the modification of the production code results in obsolete test code, the obsolete test code should be promptly updated accordingly, since obsolete test code would increase the cost of development and maintenance [50, 65]. On the other hand, if the time interval between a production code change and a test code change is too long, their modifications are likely to be uncorrelated. Careful readers may wonder if the definition of **Assumption .1** is a bit too strict. However, since the identified *PT co-evolution samples* typically serve as the foundation for downstream tasks like JIT obsolete test code detection and fault localization [85, 86], etc., the strict assumption can reduce noise and is favorable.

5.2 Limitation of Empirical Study

To enable the mining of *PT co-evolution samples*, existing *PT co-evolution-related* works widely use the following strategies to identify *PT co-evolution samples*: 1) Focusing on unit test classes and taking advantage of naming conventions to ease the association of test and production classes. 2) Collecting *PT co-evolution samples* based on the **Assumption .1**, i.e., if a test class and its associated production class change together in one commit, or a test class changes immediately after the changes of the associated production class within a short time interval (denoted by T), this change pair should be a *PT co-evolution sample*. We conduct the empirical study with the goal of investigating whether the production-test change pairs that satisfy the assumption are always *PT co-evolution samples*. Therefore, we need to follow the abovementioned assumption to construct the *PT co-evolution samples*. During the construction process, we identify all test code changes for each commit in the repository's main branch. For each test code change, denoted as $change_{test}$, we iterate through the commits before (and including) the $change_{test}$ to find the latest production code changes. Note that there may be a situation where a set of changes applied to the production class in different commits results in a change in the corresponding test class. In such case, only the pair of the last production code change and the test code change is regarded as our *PT co-evolution candidate* following this assumption. Other pairs of production code changes and test code changes fall outside the scope of the **Assumption .1**. It is worth mentioning that, these change pairs do not

threaten the validity of our empirical findings, as the purpose of our empirical study is to investigate whether the *PT co-evolution samples* that satisfy **Assumption .1** introduce noise. We plan to explore other pairs in our future work.

5.3 Limitation of Our Approach

5.3.1 The Selection of Test-to-Code Traceability Link Method. **CHOSEN** accepts a production-test change pair $\langle change_p, change_t \rangle$ as input. To construct a production-test change pair, **CHOSEN** needs to associate a production code snippet and a test code snippet. However, all existing test-to-code traceability link techniques have weaknesses that make them unsuitable for use as a general solution [83]. Therefore, following previous studies [81], we utilize a most common technique, i.e., naming convention. The specific conventions may vary between projects. However, the standard convention is that a test class should share the same name as the associated production class, with *test* prepended or appended [61, 68]. The naming convention can ensure that, for each test code change, once a production code is found by removing the *test* prefix or suffix of the test code, this production code must be the associated production code. Since Maven projects follow the standard directory layout, we only consider the projects written in Java language and managed by Maven in this paper.

However, it does not mean that our approach only applies to Java projects. For example, mainstream python testing frameworks (e.g., pytest [32]) also follow naming conventions, where test methods/functions or test classes are expected to match the "test_*.py" or "*_test.py" pattern. As for projects that do not adhere to these conventions, we can leverage other test-to-code traceability link techniques [52, 67, 83]. For example, based on the assumption that the test code should be similar to the associated production code, Csuvik et al. [41] use word embeddings to create traceability links between the test classes and associated production classes. Therefore, **CHOSEN** can be easily migrated to other languages/types of projects by accurately modeling the traceability relationship between test and production code.

5.3.2 Interpretation of the Identification Strategies. Given a *production-test change pair*, we propose a two-stage identification strategy to determine whether the production code change triggers the test code change. The proposed method is learned from the findings of the empirical study. In this section, we will discuss the intuition and interpretation of some identification strategies.

Table 10. *PT co-evolution* identification performance under different time intervals in 380 samples constructed in the empirical study.

Time Interval (T)	Acc.	Prec.	Rec.	F1.
$T = 0$	88.68%	88.66%	89.12%	88.89%
$T = 12h$	88.16%	83.94%	94.82%	89.05%
$T = 24h$	86.84%	81.78%	95.34%	88.04%
$T = 48h$	85.53%	79.74%	95.85%	87.06%
$T = 480h$	77.89%	70.26%	97.93%	81.82%

Threshold in Stage 1. The Stage 1 of **CHOSEN** uses the time interval to initially estimate the label of *product-test change pair*. When the time interval between the test code change and the production code change is less than a threshold T (in this paper, $T = 12h$), **CHOSEN** will treat it as an initial positive sample. Otherwise, it is an initial negative sample. The determination of the T is based on our empirical study, in which we set different values of T, i.e., $T = \{0, 12h, 24h, 48h\}$ and

compare the identification performance of *PT co-evolution* using these values in terms of accuracy, precision, recall, and F1-score, based on 380 labeled samples. The results of the comparison are presented in Table 10. From Table 10, we can observe that as T increases, the recall value gradually increases while the precision value gradually decreases. On the one hand, when $T > 12h$, increasing the T can bring slight recall improvement; but dramatically reduces the precision. Although $T = 0$ gets the best precision performance, it is not optimal for the F1-score. On the contrary, the best F1-score result is achieved at $T = 12h$. Although $T = 12h$ does not obtain satisfactory precision results, **CHOSEN** can further improve the precision performance through the Stage 2's identification strategies.

Strategy 1 in Stage 2. The determination of Strategy 1 is based on the findings of our empirical study. We retain the *production-test change pairs* constructed in the empirical study satisfying Strategy 1 and obtain a total of 6,253 non-modification change pairs (i.e., the production changes or test changes are non-modification types). We observe some interesting facts regarding non-modification change pairs. We find that most change pairs involve the same non-modification type of change for both production and test code, with 5,185 samples (82.92%). For the remaining 1,068

Fig. 6. *PT co-evolution* example of production code modifications leading to test class deletion in `commons-compress` project.

Listing 5. `FramedLZ4CompressorOutputStream.java` change (simplified, commit 176cd18).

```
index e0622e10..f50aa579 100644
-    private static final List<Integer> BLOCK_SIZES = Arrays.asList(64 * 1024, 256 * 1024,
-                                1024 * 1024, 4096 * 1024);
+    private final Parameters params;
+    public enum BlockSize {
+        K64(64 * 1024, 0), K256(256 * 1024, 1), M1(1024 * 1024, 2), M4(1024 * 1024, 4);
     }
+    public static class Parameters {
+        private final BlockSize blockSize;
+        public static Parameters DEFAULT = new Parameters(BlockSize.M4);
+        public Parameters(BlockSize blockSize) {
+            this.blockSize = blockSize;
+        }
+    }
-    public FramedLZ4CompressorOutputStream(OutputStream out, int blockSize)
-                                                                    throws IOException {
-        if (!BLOCK_SIZES.contains(blockSize)) {
-            throw new IllegalArgumentException("Unsupported block size");
-        }
+    public FramedLZ4CompressorOutputStream(OutputStream out, Parameters params)
                                                                     throws IOException {
+        this.params = params;
}
```

Listing 6. `FramedLZ4CompressorOutputStreamTest.java` change (simplified, commit 176cd18).

```
index 5f2802e2..00000000
-    package org.apache.commons.compress.compressors.lz4;
-    import java.io.ByteArrayOutputStream;
-    import java.io.IOException;
-    import org.junit.Test;
-    public final class FramedLZ4CompressorOutputStreamTest {
-        @Test(expected = IllegalArgumentException.class)
-        public void illegalBlockSize() throws IOException {
-            new FramedLZ4CompressorOutputStream(new ByteArrayOutputStream(), 32 * 1024);
-        }
-    }
```

samples, we randomly select 10 change pairs. By manually checking and analyzing test change reasons, such selected samples are all *PT co-evolution samples*. Obviously, considering only the change pairs of which the non-modification type change must occur and must be the same on both production changes and test changes as *PT co-evolution samples*, we may ignore some positives.

We further analyze the abovementioned observations. On the one hand, some change pairs of different non-modification types are due to the modification type of production or test change being `RENAME` and `COPY`. Github offers five types of file change, including `MODIFY`, `ADD`, `DELETE`, `RENAME`, and `COPY`. `RENAME` is usually caused by changes to the file path, which can be seen as the deletion of files located in the old path and the addition of files located in the new path. Meanwhile, `COPY` operation adds a new file in the target file path. Secondly, some change pairs are due to developers modifying the production code resulting in the addition of a test class (which accounts for 678/1,068 cases). For example, in the ta4j project, the production code `XorRule` removed the `check` method and added a new `isSatisfied` method [9]. Three days later, the developers added a new test class `XorRuleTest` and a test case that tests the focal method `isSatisfied` [10]. In addition, we observe some uncommon cases where the developer simply removed the test class without removing the production class, accounting for 8.80% (55/6,253). As shown in Figure 6, before the code is committed, the production code defines the `BLOCK_SIZES` variable to store a set of block sizes. When the production code accepts a block size that does not fall within the specified range, an `IllegalArgumentException` is thrown, which is also tested by the corresponding test case. Subsequently, the developer removed the conditional statement that threw the exception and added the `Parameters` inner class and the `BlockSize` enumeration class to enforce the use of a predefined block size. The changes to the production code make the test case redundant, and the test class contains only one test case. As a result, the developer directly deleted the entire test class.

Strategy 2 in Stage 2. For Strategy 2, we define a production-test change pair as a *PT non-co-evolution sample* when there are additional production code modifications between the production code change commit and test code change commit. The definition of Strategy 2 is based on the following reasons: 1) ***Consistent with problem formulation (as described in Section 3.2.1)***. Strategy 2 allows us to focus on the co-evolution of the test changes with the most recent production changes; 2) ***Mapping to the PT co-evolution distribution of empirical study***. Our empirical study shows that *PT co-evolution* usually occurs in a short time interval (T), and the number of *PT non-co-evolution samples* increases significantly whenever T increases slightly (as shown in Table 2). Thus, when there are multiple production code modifications prior to the test changes, it is reasonable to consider that the earlier production modifications are associated with the test modifications with low confidence. Strategy 2 ensures that **CHOSEN** constructs the *PT co-evolution samples* as precisely as possible. However, there may be multiple consecutive production code changes that occur in different commits and are associated with a test code update. In such a case, the last production and test code change must be a *PT co-evolution sample*, but **CHOSEN** would identify other change pairs as negatives, which may sacrifice some recall results while ensuring high precision performance. Improving the recall performance of **CHOSEN** in identifying *PT co-evolution samples* will be part of our future work.

5.4 Limitation of CHOSEN Application

Following previous research [42, 55, 81, 88], **CHOSEN** focuses on unit test classes and identifies the *PT co-evolution samples* where test modifications are triggered by production modifications. Interestingly, in our empirical research, we observe some cases where test code changes trigger subsequent production code modifications (i.e., category 6 in the taxonomy). Although such cases are not many, accounting for only 0.53%, limited by our study scope, **CHOSEN** cannot identify them. However, our qualitative analysis can offer insight for future researchers to address this

limitation. As mentioned in Section 2.3, developers may, in order to fix a bug in production code, add a failed test case that triggers the bug [6]. Generally, with the addition of failed test cases, developers would explicitly describe modification intentions in commit messages and comments and assign an ID to denote the bug. When subsequent developers fix the production bug, they can refer to the previously defined bug ID in the corresponding commit message. This insight inspires us to identify test modifications that may trigger subsequent production modifications by extracting their comments and commit messages. Then, developers can use the assigned bug ID to traverse the commit history to search for production modifications.

5.5 Implications

Identifying *PT co-evolution samples* is crucial for helping developers understand the co-evolution process of a project (e.g., how source code entities co-evolve). Furthermore, the mined *PT co-evolution samples* can be used in downstream tasks to further facilitate co-evolution in a project, such as JIT OTCD. Our large-scale study in RQ1 confirms that the assumption used in the literature for identifying co-evolution samples is not always held, and even when production code changes and test code changes happen simultaneously, they are not necessarily *PT co-evolution samples*. In other words, production code changes do not always result in updates of corresponding test code. Therefore, we are not claiming it is a bad practice to not update test code as a consequence of production code changes. Additionally, our qualitative analysis reveals that the factors leading to test code updates are diverse, providing us with some lessons learned. In the following, we discuss the implications for researchers and/or practitioners from our findings.

5.5.1 Implications for Practitioners. **Diversity of test code updates.** Software engineers and developers should be aware that test code updates can be diverse and not always triggered by production code changes. Our empirical study has shown that a considerable number of test code changes are related to factors such as test framework updates, refactoring, or bug fixing within the test code itself. In addition to the test code updates due to adaptive, perfective, and corrective maintenance activities, there are also indirectly related production code updates and indirectly related test code updates. Therefore, it is important to not solely rely on production code changes as an indicator of when to update test code and to attribute test code changes simply to production code changes. Additionally, developers should consider monitoring and analyzing the test code changes to ensure that the tests remain relevant and effective in detecting defects. Overall, being proactive in maintaining and updating test code can help to catch issues early and reduce the cost and effort of fixing bugs later in the development cycle.

Diversity of software artifact co-evolution, including indirectly related production/test code. We observed a series of cases where test code updates are due to other software artifacts, e.g., indirectly related production code and indirectly related test code. For example, a function signature change of an indirectly related production code leads to the modifications of a series of test files that call the method [13]. Similarly, for a test class with an inheritance relationship, changes in the parent class may cause updates in the subclass [8]. This highlight that when developers update source code entities, they also need to consider the impact of other code entities that are associated or inherited. The implications of this finding for software engineers and developers are significant. It suggests that developers should broaden their scope when updating software artifacts and be mindful of the potential ripple effects on other code entities. Specifically, they need to consider how updates to indirectly related code artifacts may affect the behavior and functionality of the entire system. This broader perspective can improve the quality and stability of software systems and reduce the potential for unexpected bugs and errors. By recognizing the diversity of

co-evolution and adopting a more holistic approach to software updates, developers can enhance their software development skills and produce more robust and reliable software systems.

5.5.2 Implications for Researchers. **PT co-evolution identification is still an open problem.** Existing studies related to *PT co-evolution*, including this paper, have primarily focused on code changes at the class level. However, there are still significant challenges and gaps in understanding method-level *PT co-evolution*. Therefore, further exploration of method-level *PT co-evolution* is necessary to provide better solutions for developers. Researchers can work towards developing new techniques and tools that can help identify finer-grained co-evolution between production code and test code, as well as the impact of test code changes on production code and vice versa, ultimately facilitating the *PT co-evolution* process. As discussed in Section 1, a major research challenge in identifying and mining *PT co-evolution samples* is how to associate a test code snippet and a production code snippet. In this paper, we utilize the *naming convention* (NC) approach, which exhibits high precision, particularly in projects that strictly adhere to naming conventions. However, NC's performance is limited in projects that do not follow naming conventions. Therefore, there are many research opportunities here for researchers to further improve *PT co-evolution* identification by constructing more accurate test-to-code traceability links.

The dataset of other code entities co-evolution is necessary. Our empirical study shows that the assumption used in previous studies for constructing *PT co-evolution samples* does not always hold and may introduce noise. However, manually collecting *PT co-evolution samples* is time-consuming for researchers. Therefore, we propose a method for automatically identifying PT co-evolution samples (i.e., **CHOSEN**). **CHOSEN** achieves promising performances (accuracy: 92.89%) and has the potential to help researchers construct high-quality *PT co-evolution samples* from the large amount of historical data stored in GitHub and other version control systems. However, software is multi-dimensional, and so is the development process behind it. This multi-dimensionality lies in the fact that other artifacts are required to develop high-quality source code, such as requirements, documentation, etc., [63]. Our qualitative analysis has shown that there are various forms of co-evolution scenarios, such as the co-evolution of test code with indirectly related production code and the co-evolution of test code with indirectly related test code. Therefore, in the future, it is necessary to construct standard datasets of different forms of co-evolution to facilitate researchers to perform co-evolution-related work.

5.6 Threats to Validity

Despite our best efforts, we are aware of some threats to validity. In this section, we briefly discuss them, which are grouped into the following categories.

5.6.1 Threats to Internal Validity. Internal validity threats concern factors internal to our study that could influence our results.

Manual Analysis. The use of manual analysis could suffer from subjectivity bias for interpretation in determining the test update reasons. To mitigate this threat, we employed three volunteers to conduct a two-phase check. All test update reasons have a full inter-rater agreement.

Saturation Evaluation. To determine whether our manual analysis achieved saturation, we followed the common practice of previous studies [79, 80]: near the end of the manual analysis, the cards/tags collection can be considered saturation when new cards/tags no longer appeared. Similarly, we found that after analyzing about 80% of the changed pairs, volunteers did not add any new tags. Thus, we claimed that our manual analysis has reached saturation. Additionally, we randomly selected 10% of the PT non-co-evolution samples from the evaluation dataset constructed by RQ3. Upon the analysis of these test change reasons, we found that the test change patterns were

always found in the taxonomy, providing further evidence that our manual analysis has reached saturation.

Dependent Tools. The detection of refactoring operations relies entirely on off-the-shelf tools, and to reduce the risk of tools, we choose state-of-the-art detection tools, which have wide applications and high accuracy. Our method implementations involve the GumTree tool whose potential threat [44] may limit the effectiveness of our approach.

Test-to-Code Traceability Construction. The third threat to internal validity is that constructing the production-test change pair may introduce traceability noise, i.e., the production code that is not the associated production code of the test code. Following previous work [81, 88, 89], we adopt the naming convention, a very high precision traceability technique in class level [83], with the goal of containing minimal traceability noise, although this technique can have a poor recall. On the other hand, this technique is ineffective if the project does not adhere to the naming conventions. Notwithstanding these limitations, as discussed in Section 5.3.1, **CHOSEN** remains easily transferable to projects that use other programming languages and do not follow naming conventions.

Identification Strategies and the Mapping to the Taxonomy. Our empirical findings inform the five identification strategies we propose. Strategies 1 and 2 are determined based on the distribution of *PT co-evolution samples*, while strategies 3, 4, and 5 are derived from the summarized taxonomy. This paper focuses on the co-evolution of a test class and its associated production class. Consequently, we developed strategies to identify *PT non-co-evolution samples* based on the maintenance activities of test code in the taxonomy, including adaptive maintenance, perfective maintenance, and corrective maintenance. Other code artifacts, such as indirectly related production code as well as indirectly related test code, resulting in test code updates, are beyond the scope of this paper. RefactoringMiner can detect most test change patterns related to maintenance activities of test code in taxonomy, such as extracting methods, renaming methods/classes, and adding/removing annotations [77]. We define Strategy 4 to ensure that test and production changes are semantically relevant. However, some change patterns, such as "fix test code faults", are not covered by **CHOSEN**. Detecting and implementing these uncovered change patterns and other test update categories will be part of our future work. **CHOSEN** is limited in identifying cases where test modifications trigger production modifications, but we provide possible solutions in Section 5.4. Finally, a potential threat to the validity of our study is that we only consider the last change to a production code as a possible pair to a subsequent change to the related test class. This may result in overlooking potential *PT co-evolution* patterns that occur with multiple changes to the production code. Although **CHOSEN** may not capture the full extent of *PT co-evolution*, it ensures that the identified *PT co-evolution samples* are sufficiently accurate. To address the limitation, future work could explore more sophisticated techniques to identify and track co-evolution patterns across multiple changes to production code, which may involve analyzing code changes at a finer-grained level or using more advanced machine learning techniques.

5.6.2 Threats to External Validity. External validity is specifically about to what extent our experimental results can be generalized.

Subject Selection. The main threats to validity come from the subjects. An external threat is the representativeness of the projects since we have no clear evidence as to how representative the selected projects are. However, the adopted projects are widely used in co-evolution-related research, and the labeled sample size (more than 1k) is large enough to demonstrate our approach's generality. As explained by Kalliamvakou et al. [51], relying solely on the number of stars to select popular projects may not accurately reflect a project's true popularity, which can be influenced by factors other than project quality, such as social networks and marketing efforts. To mitigate

this potential threat, we applied multiple strict selection criteria, including a long change history and compilability, and manually verified if the star count of these projects increased within a short period of time. Moreover, the selected project list is obtained from previous studies [81, 82] and is widely used in the software engineering community.

Evaluation Dataset. To avoid overfitting, **CHOSEN** has been evaluated on a different set of 380 samples from the 33,567 production-test change pairs while ensuring that there are no overlap or duplicate samples between these 380 samples and that of our empirical study. Careful readers may wonder, even though the two datasets are different, they may represent similar information since both are statistically significant samples of the original dataset. However, in evaluating the **CHOSEN**'s usefulness in JIT OTCD, we select and label three projects, of which `Flink` and `Log4j2` are not among the 44 projects of the empirical study. For the `Flink` and `Log4j2` datasets (a total of 609 labeled change pairs), the experimental results further confirm the effectiveness of **CHOSEN** in identifying *PT co-evolution samples*: Table 7 shows that the test set constructed by **CHOSEN** more accurately measures the performance of the OTCD model with an average error of 0.80%, while SITAR_IM leads to an error of 6.00%. We also acknowledge the fact that the evaluation of **CHOSEN** requires more projects as well as a larger labeled dataset. We are actively working on addressing these limitations in our continuing work.

6 RELATED WORK

6.1 Establishing Traceability Links between Production Code and Test Code

Previous works have proposed methods to establish traceability between production and test code. Given a test class, Qusef et al. [67] exploited dynamic slicing to identify a set of candidate production classes. Then, external and internal textual information associated with the classes were used to identify the final set of tested classes. White et al. [83] utilized a wide range of techniques to establish links between tests and tested code, as well as between test classes and tested classes. At the method level, developers can utilize various tools and frameworks to annotate tests with links to the method-under-test. One such framework is EzUnit [38], which performs a static analysis and suggests methods called by a test for annotation. When an error occurs in the test, EzUnit highlights the linked methods. Ghafari et al. [47] also work at the method level, where they decompose test cases into sub-scenarios to identify the tested function, using static data flow analysis. These tools and techniques provide valuable insights for establishing traceability at the method level.

Other test-to-code traceability approaches assume that a test should be similar to the corresponding tested unit. Kicsi et al. [52] utilized Latent Semantic Indexing (LSI) on source code to establish traceability links between test classes and production classes. The experimental results show that the ground truth link is ranked top between 30% and 62% and suggests a low recall, with no investigation of precision. Csuvik et al. [41] used word embeddings instead of LSI in the same approach and achieved better precision. Although naming convention can achieve good traceability accuracy on the class level since developers usually follow naming conventions for test classes, on the method level, there are various guidelines for naming test methods, leading to challenges in achieving accurate traceability. Madeja et al. [61] investigated and found that only 49% of tests' name included the full name of the focal method, while 76% of tests' name contained a partial name of a focal method. This paper mainly focuses on the *PT co-evolution* at class level, so we adopt the naming convention with the goal of containing minimal traceability noise. Different from these traceability studies, our purpose is not to associate production code and test code, but to identify whether test code changes are triggered by the production code changes.

6.2 Co-evolution of Production and Test Code

Levin and Yehudai studied co-evolution with semantic changes [59]. They focused on the co-evolution caused by specific maintenance activities, using the number of different types of activities as change features to mine the pattern of *PT co-evolution*. Unlike their work, we explore test code updates occurring after the changes of the associated production code, composing six root categories. Moreover, Mamdouh et al. [35] have presented statistical evidence of a significant relationship between production code and its associated test suites. Specifically, they utilized several metrics used to determine the size of production code or tests, such as the number of classes, lines of code (LOC), number of methods, and number of packages, to understand and identify how test cases evolve during production code changes (releases) in terms of size and complexity. Instead, our study focuses on accurately identifying *production-test co-evolution samples*, where test code change triggers or is triggered by production code change. Previous researchers have investigated the nature of the co-evolution between production code and test code (i.e., synchronous or phased) [60, 88, 89]. In [60], Lubsen et al. used association rule mining to examine the *PT co-evolution* in two case studies: an open-source system and an industrial software system. In [89] and [88], the researchers proposed three views which are: change history view, growth history view, and test coverage evolution view, and combine them to study how production-test co-evolves over time. Further, they demonstrated and validated the use of such views on two open-source cases and one industrial case, drawing several relevant observations about the testing processes used in development. However, the abovementioned studies [60, 88, 89] are conducted based on identified and mined *PT co-evolution samples*. To this end, they directly identified those production-test change pairs where the test class changes and its associated production class changes together in one commit as *production-test co-evolution samples*. However, our empirical study has shown that such an identification method would introduce noise, which may pose a threat to their conclusions. To mitigate this threat, we propose **CHOSEN** to identify *PT co-evolution samples* accurately. We believe that **CHOSEN** can enhance the validity of existing works related to *PT co-evolution*.

Mars-avina et.al. [62] run each modified test case to identify all the entities from the production code covered by the test, thereby constructing *PT co-evolution samples* dynamically. Based on *PT co-evolution samples*, co-evolution patterns are mined by ChangeDistiller [46]. Intuitively, compared with static analysis, a dynamic solution may better identify *PT co-evolution samples*. However, dynamic methods need to run tests to collect coverage information, are therefore more expensive [50]. Additionally, dynamic methods can achieve high recall but may sacrifice precision since they identify all the entities in the production code accessed by the test [62]. In contrast, **CHOSEN** is a lightweight static approach that identifies *PT co-evolution samples* with high precision. We believe **CHOSEN** and dynamic methods can complement each other. Wang et al. [81] conducted a large-scale empirical study (including 975 open-source Java projects) to understand the practice of *PT co-evolution*. To ease test maintenance whenever the production code changes, they further proposed a machine-learning-based approach named SITAR for JIT obsolete test code detection. In their approach, SITAR focuses more on changing lines of code in programming language constructs. SITAR consists of two components: 1) the construction of *PT co-evolution samples* (positives) and *PT non-co-evolution samples* (negatives); 2) designing several structural code features and training detectors to detect obsolete test code based on the mined *PT co-evolution samples*. In Section 3.1, we have demonstrated that SITAR's identification method of *PT co-evolution samples* introduces noise, which causes SITAR's detection performance on the testing set to be significantly different from its actual performance. Compared with SITAR, our work focuses on constructing the *PT co-evolution samples*. Moreover, **CHOSEN** can measure the performance of

OTCD models more accurately, and the dataset constructed by **CHOSEN** can train a better detection model (as shown in Section 4.3).

7 CONCLUSION AND FUTURE WORK

This paper explores the reasonableness of the assumption which is widely used by prior works to collect and label *production-test co-evolution samples*: if a test class and its associated production class change together in one commit, or a test class changes immediately after the changes of associated production class within a short time interval, this *change pair* should be a *production-test co-evolution sample*. To this end, we conduct an empirical study and demonstrate that the assumption does not always hold in practice and the methods used by prior work to identify *PT co-evolution samples* would introduce noise. Therefore, we propose an identification method of *PT co-evolution*, namely **CHOSEN**. We evaluate the effectiveness of **CHOSEN** on a manually labeled dataset. The results show that **CHOSEN** can more accurately identify *PT co-evolution samples*. In addition, We demonstrate that **CHOSEN** can be used to facilitate the downstream tasks. For the obsolete test code detection task, the test set labeled by **CHOSEN** can help accurately measure the detection model's performance with only 0.76% errors, and the training set labeled by **CHOSEN** can help improve the performance of the detection model. When the cost of manual labels is unaffordable, we recommend that researchers construct *PT co-evolution samples* through our method, thereby improving the validity of the findings. Limited by our study scope, **CHOSEN** cannot identify the case where production modifications are triggered by test modifications (in our empirical study, two samples satisfy this pattern, accounting for 0.53%). Therefore, one important direction for future work is on the extension of **CHOSEN** to identify *PT co-evolution* where test modifications trigger production modifications. In addition, it would be very promising to adopt other state-of-the-out linking methods that associate production code and test code to construct production-test change pairs, thereby further improving the generalizability of **CHOSEN**.

8 ACKNOWLEDGEMENTS

This work was supported in part by the National Key Research and Development Project (No. 2021YFB1714200), the Fundamental Research Funds for the CentralUniversities (No. 2022CDJDX-005), the Chongqing Technology Innovation and Application Development Project (No. CSTB2022TIAD-STX0007 and No. CSTB2022TIAD-KPX0067), the National Natural Science Foundation of China (No. 62002034) and the Natural Science Foundation of Chongqing (No. cstc2021jcyj-msxmX0538).

REFERENCES

[1] 2011. A commit in Commons-functor project, `BinaryAnd.java`. https://github.com/apache/commons-functor/commit/cb42eacc6a82f98da131f32972915f6cde609fd9.
[2] 2012. A commit in Commons-csv project. https://github.com/apache/commons-csv/commit/6a34b823c807325bc251ef43c66c307adcd947b8.
[3] 2012. A commit in Commons-functor project, `TestBinaryAnd.java`. https://github.com/apache/commons-functor/commit/22ec28d8aabe5dcdf9f4723f573395822611f6b5.
[4] 2013. A commit in Joda-time project. https://github.com/JodaOrg/joda-time/commit/3a413d7844c22dc6ddd50bf5d0d55ff3589e47ac.
[5] 2014. A commit in Commons-compress project, `ZipFile.java`. https://github.com/apache/commons-compress/commit/885d2053f4fc29d986904c9b8cfe69bcfbe7b361.
[6] 2014. A commit in Commons-compress project, `ZipFileTest.java`. https://github.com/apache/commons-compress/commit/bc741b19e88749d66b03bf8dc292f3ae0fc74156.
[7] 2015. A commit in Commons-math project, `GraggBulirschStoerStepInterpolator.java`. https://github.com/apache/commons-math/commit/8e0b98bf6bd30713d94b72c7c410addb26c3c472.
[8] 2015. A commit in Commons-math project, `GraggBulirschStoerStepInterpolatorTest.java`. https://github.com/apache/commons-math/commit/bf803b119be94bfd71902ea5db06075aada82672.

[9] 2015. A commit in ta4j project, `XorRule.java`. https://github.com/ta4j/ta4j/commit/db5576521118459df4b36120ed9e7b7fae5aedca.
[10] 2015. A commit in ta4j project, `XorRuleTest.java`. https://github.com/ta4j/ta4j/commit/1b3f7949962d01e3d5724437d1fc4d301c124c3b.
[11] 2018. A commit in Commons-configuration project, `DefaultParametersManager.java`. https://github.com/apache/commons-configuration/commit/fa5dbfaf68973a204dc09acb42909ff5bd39ff70.
[12] 2018. A commit in ta4j project, `InSlopeRule.java`. https://github.com/ta4j/ta4j/commit/590cab635abe3fbdb30d2a0bed66bee8421c254e.
[13] 2018. A commit in ta4j project, `InSlopeRuleTest.java`. https://github.com/ta4j/ta4j/commit/590cab635abe3fbdb30d2a0bed66bee8421c254e.
[14] 2019. A commit in Cactoos project, `Cycled.java`. https://github.com/yegor256/cactoos/commit/b092754a6e18a39951e27733490be1961dadfeb0.
[15] 2019. A commit in Commons-configuration project, `TestDefaultParametersManager.java`. https://github.com/apache/commons-configuration/commit/63bb3e88d13e6447f64bac47fcfd71b60c5a4c3e.
[16] 2019. A commit in Commons-lang project, `Validate.java`. https://github.com/apache/commons-lang/commit/37442639705892348d2cd6d7717fff4d9841ca09.
[17] 2019. A commit in Commons-numbers project. https://github.com/apache/commons-numbers/commit/7427fd0639557f25a2d7274597be70882527ffd0.
[18] 2019. A commit in Dubbo project. https://github.com/apache/dubbo/commit/c91618b05f6137b291134ef10ebd28a918193ecd.
[19] 2020. A commit in Cactoos project, `CycledTest.java`. https://github.com/yegor256/cactoos/commit/97454478d07f360b68ff98504e803f26d6777ae9.
[20] 2020. A commit in Commons-collections project. https://github.com/apache/commons-collections/commit/1d26ffda9302433fda227c5724d2f5cd499b0148.
[21] 2020. A commit in Commons-geometry project, `Transform2S.java`. https://github.com/apache/commons-geometry/commit/38f25f8fe5eccdde5213555b0a97f46214b37277.
[22] 2020. A commit in Commons-geometry project, `Transform2STest.java`. https://github.com/apache/commons-geometry/commit/b36deb014b5c0a2332d225d871db14a58def5200.
[23] 2020. A commit in Commons-lang project, `ValidateTest.java`. https://github.com/apache/commons-lang/commit/485876f9c2d90b211b5776567086ec0700767f3c.
[24] 2020. A commit in Easy-rules project. https://github.com/j-easy/easy-rules/commit/1a0660140c6786458a92b28c4f650b1c5e0c40bc.
[25] 2020. A commit in Streamex project, `EntryStream.java`. https://github.com/amaembo/streamex/commit/836a2e5240321cdf2d6e54239110f62f94a540bb.
[26] 2020. A commit in Streamex project, `EntryStreamTest.java`. https://github.com/amaembo/streamex/commit/acd58e99a9ebfeed2c289f50f37a9516c50e72be.
[27] 2021. About stars (GitHub). https://help.github.com/articles/about-stars/.
[28] 2021. Checkstyle. http://checkstyle.sourceforge.net/.
[29] 2021. Jacoco. https://github.com/jacoco/jacoco.
[30] 2021. JUnit 5 User Guide. https://junit.org/junit5/docs/current/user-guide/.
[31] 2021. Maven-introduction to the standard directory layout. https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html.
[32] 2021. Pytest. https://pytest.org/en/latest/explanation/goodpractices.html#test-discovery.
[33] 2021. SonarQube. https://www.sonarqube.org/.
[34] 2021. The dataset of SITAR. https://github.com/sqlab-sustech/Sitar-project.
[35] Mamdouh Alenezi, Mohammed Akour, and Hiba Al Sghaier. 2019. The impact of co-evolution of code production and test suites through software releases in open source software systems. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)* 9, 1 (2019), 2737–2739.
[36] Andrea Arcuri and Lionel Briand. 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250.
[37] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and Don'ts of Machine Learning in Computer Security. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. USENIX Association, 3971–3988. https://www.usenix.org/conference/usenixsecurity22/presentation/arp
[38] Philipp Bouillon, Jens Krinke, Nils Meyer, and Friedrich Steimann. 2007. EzUnit: A Framework for Associating Failed Unit Tests with Potential Programming Errors. In *Agile Processes in Software Engineering and Extreme Programming, 8th International Conference, XP 2007, Como, Italy, June 18-22, 2007, Proceedings (Lecture Notes in Computer Science,*

Vol. 4536). Springer, 101–104. https://doi.org/10.1007/978-3-540-73101-6_14

[39] Jiachi Chen, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. 2021. Maintenance-related concerns for post-deployed Ethereum smart contract development: issues, techniques, and future challenges. *Empirical Software Engineering* 26, 6 (2021), 1–44.

[40] Norman Cliff. 2014. *Ordinal methods for behavioral data analysis*. Psychology Press.

[41] Viktor Csuvik, András Kicsi, and László Vidács. 2019. Source code level word embeddings in aiding semantic test-to-code traceability. In *Proceedings of the 10th International Workshop on Software and Systems Traceability, SST@ICSE 2019, Montreal, QC, Canada, May 27, 2019*. IEEE / ACM, 29–36. https://doi.org/10.1109/SST.2019.00016

[42] Barrett Ens, Daniel Rea, Roiy Shpaner, Hadi Hemmati, James E. Young, and Pourang Irani. 2014. ChronoTwigger: A Visual Analytics Tool for Understanding Source and Test Co-evolution. In *2014 Second IEEE Working Conference on Software Visualization*. IEEE Computer Society, 117–126.

[43] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. ACM, 313–324.

[44] Yuanrui Fan, Xin Xia, David Lo, Ahmed E. Hassan, Yuan Wang, and Shanping Li. 2021. A Differential Testing Approach for Evaluating Abstract Syntax Tree Mapping Algorithms. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1174–1185. https://doi.org/10.1109/ICSE43902.2021.00108

[45] Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological bulletin* 76, 5 (1971), 378.

[46] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. 2007. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Trans. Software Eng.* 33, 11 (2007), 725–743.

[47] Mohammad Ghafari, Carlo Ghezzi, and Konstantin Rubinov. 2015. Automatically identifying focal methods under test in unit test cases. In *15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, September 27-28, 2015*. IEEE Computer Society, 61–70. https://doi.org/10.1109/SCAM.2015.7335402

[48] Jin Huang and Charles X. Ling. 2005. Using AUC and Accuracy in Evaluating Learning Algorithms. *IEEE Trans. Knowl. Data Eng.* 17, 3 (2005), 299–310. https://doi.org/10.1109/TKDE.2005.50

[49] Qiao Huang, Xin Xia, and David Lo. 2017. Supervised vs Unsupervised Models: A Holistic Look at Effort-Aware Just-in-Time Defect Prediction. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 159–170. https://doi.org/10.1109/ICSME.2017.51

[50] Victor Hurdugaci and Andy Zaidman. 2012. Aiding software developers to maintain developer tests. In *2012 16th European Conference on Software Maintenance and Reengineering* (2012). IEEE, IEEE Computer Society, 11–20.

[51] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2016. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering* 21 (2016), 2035–2071.

[52] András Kicsi, László Tóth, and László Vidács. 2018. Exploring the Benefits of Utilizing Conceptual Information in Test-to-Code Traceability. In *6th IEEE/ACM International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, RAISE@ICSE 2018, Gothenburg, Sweden, May 27, 2018*. ACM, 8–14. https://doi.org/10.1145/3194104.3194106

[53] Mijung Kim, Jaechang Nam, Jaehyuk Yeon, Soonhwang Choi, and Sunghun Kim. 2015. REMI: defect prediction for efficient API testing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. ACM, 990–993.

[54] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. 2016. The emerging role of data scientists on software development teams. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, ACM, 96–107.

[55] Ahmed Lamkanfi and Serge Demeyer. 2010. Studying the co-evolution of application code and test cases. In *Proceedings of the 9th Belgian-Netherlands Software Evolution Seminar (BENEVOL 2010), Lille, France*, Vol. 16.

[56] Valentina Lenarduzzi, Antonio Martini, Davide Taibi, and Damian Andrew Tamburri. 2019. Towards surgically-precise technical debt estimation: early results and research roadmap. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation, MaLTeSQuE@ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 27, 2019*. ACM, 37–42. https://doi.org/10.1145/3340482.3342747

[57] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. 2019. The Technical Debt Dataset. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2019, Recife, Brazil, September 18, 2019*. ACM, 2–11. https://doi.org/10.1145/3345629.3345630

[58] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. 2020. Some sonarqube issues have a significant but small effect on faults and changes. a large-scale empirical study. *Journal of Systems and Software* 170 (2020), 110750.

[59] Stanislav Levin and Amiram Yehudai. 2017. The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, IEEE Computer Society, 35–46.

[60] Zeeger Lubsen, Andy Zaidman, and Martin Pinzger. 2009. Using association rules to study the co-evolution of production & test code. In *Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009 (Co-located with ICSE), Vancouver, BC, Canada, May 16-17, 2009, Proceedings*. IEEE Computer Society, 151–154. https://doi.org/10.1109/MSR.2009.5069493

[61] Matej Madeja and Jaroslav Porubän. 2019. Tracing Naming Semantics in Unit Tests of Popular Github Android Projects. In *8th Symposium on Languages, Applications and Technologies, SLATE 2019, June 27-28, 2019, Coimbra, Portugal (OASIcs, Vol. 74)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:13. https://doi.org/10.4230/OASIcs.SLATE.2019.3

[62] Cosmin Marsavina, Daniele Romano, and Andy Zaidman. 2014. Studying Fine-Grained Co-evolution Patterns of Production and Test Code. In *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014*. IEEE Computer Society, 195–204. https://doi.org/10.1109/SCAM.2014.28

[63] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. 2005. Challenges in software evolution. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*. IEEE, 13–22.

[64] Audris Mockus and Lawrence G. Votta. 2000. Identifying Reasons for Software Changes using Historic Databases. In *2000 International Conference on Software Maintenance, ICSM 2000, San Jose, California, USA, October 11-14, 2000*. IEEE Computer Society, 120–130. https://doi.org/10.1109/ICSM.2000.883028

[65] Leon Moonen, Arie van Deursen, Andy Zaidman, and Magiel Bruntink. 2008. On the interplay between software testing and evolution and its effect on program comprehension. In *Software evolution*. Springer, 173–202.

[66] Reza Meimandi Parizi, Sai Peck Lee, and Mohammad Dabbagh. 2014. Achievements and Challenges in State-of-the-Art Software Traceability Between Test and Code Artifacts. *IEEE Transactions on Reliability* 63, 4 (2014), 913–926. https://doi.org/10.1109/TR.2014.2338254

[67] Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. 2014. Recovering test-to-code traceability using slicing and textual analysis. *Journal of Systems and Software* 88 (2014), 147–168.

[68] Bart Van Rompaey and Serge Demeyer. 2009. Establishing Traceability Links between Unit Test Cases and Units under Test. In *13th European Conference on Software Maintenance and Reengineering, CSMR 2009, Architecture-Centric Maintenance of Large-SCale Software Systems, Kaiserslautern, Germany, 24-27 March 2009*. IEEE Computer Society, 209–218.

[69] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. 2008. *Introduction to information retrieval*. Vol. 39. Cambridge University Press Cambridge.

[70] Lin Shi, Fangwen Mu, Xiao Chen, Song Wang, Junjie Wang, Ye Yang, Ge Li, Xin Xia, and Qing Wang. 2022. Are we building on the rock? on the importance of data preprocessing for code summarization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 107–119. https://doi.org/10.1145/3540250.3549145

[71] Danilo Silva, João Paulo da Silva, Gustavo Jansen de Souza Santos, Ricardo Terra, and Marco Tulio Valente. 2021. RefDiff 2.0: A Multi-Language Refactoring Detection Tool. *IEEE Trans. Software Eng.* 47, 12 (2021), 2786–2802.

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

[73] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.

[74] Xiaobing Sun, Xin Peng, Hareton Leung, and Bin Li. 2016. ComboRT: A New Approach for Generating Regression Test Cases for Evolving Programs. *Int. J. Softw. Eng. Knowl. Eng.* 26, 6 (2016), 1001.

[75] Jie Tan, Daniel Feitosa, and Paris Avgeriou. 2022. Does it matter who pays back Technical Debt? An empirical study of self-fixed TD. *Information and Software Technology* 143 (2022), 106738.

[76] Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. 2022. What Makes a Good Commit Message?. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2389–2401. https://doi.org/10.1145/3510003.3510205

[77] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 3 (2022), 930–950. https://doi.org/10.1109/TSE.2020.3007722

[78] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 483–494. https://doi.org/10.1145/3180155.3180206

[79] Zhiyuan Wan, Xin Xia, Ahmed E. Hassan, David Lo, Jianwei Yin, and Xiaohu Yang. 2020. Perceptions, Expectations, and Challenges in Defect Prediction. *IEEE Trans. Software Eng.* 46, 11 (2020), 1241–1266. https://doi.org/10.1109/TSE.2018.2877678

[80] Zhiyuan Wan, Xin Xia, David Lo, and Gail C. Murphy. 2021. How does Machine Learning Change Software Development Practices? *IEEE Trans. Software Eng.* 47, 9 (2021), 1857–1871. https://doi.org/10.1109/TSE.2019.2937083

[81] 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.

[82] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 53–64. https://doi.org/10.1109/ICPC.2019.00019

[83] Robert White, Jens Krinke, and Raymond Tan. 2020. Establishing multilevel test-to-code traceability links. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 861–872. https://doi.org/10.1145/3377811.3380921

[84] Xin Xia, Emad Shihab, Yasutaka Kamei, David Lo, and Xinyu Wang. 2016. Predicting crashing releases of mobile applications. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.

[85] Meng Yan, Xin Xia, Yuanrui Fan, Ahmed E. Hassan, David Lo, and Shanping Li. 2022. Just-In-Time Defect Identification and Localization: A Two-Phase Framework. *IEEE Trans. Software Eng.* 48, 2 (2022), 82–101. https://doi.org/10.1109/TSE.2020.2978819

[86] Meng Yan, Xin Xia, Yuanrui Fan, David Lo, Ahmed E. Hassan, and Xindong Zhang. 2020. Effort-aware just-in-time defect identification in practice: a case study at Alibaba. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. ACM, 1308–1319.

[87] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. ACM, 157–168. https://doi.org/10.1145/2950290.2950353

[88] 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. *Empir. Softw. Eng.* 16, 3 (2011), 325–364.

[89] 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, IEEE Computer Society, 220–229.