T-Evos: A Large-Scale Longitudinal Study on CI Test Execution and Failure

An Ran Chen, *Student Member, IEEE,* Tse-Hsun (Peter) Chen, *Member, IEEE,*
Shaowei Wang, *Member, IEEE,*

Abstract—Continuous integration is widely adopted in software projects to reduce the time it takes to deliver the changes to the market. To ensure software quality, developers also run regression test cases in a continuous fashion. The CI practice generates commit-by-commit software evolution data that provides great opportunities for future testing research. However, such data is often unavailable due to space limitation (e.g., developers only keep the data for a certain period) and the significant effort involved in re-running the test cases on a per-commit basis. In this paper, we present T-Evos, a dataset on test result and coverage evolution, covering 8,093 commits across 12 open-source Java projects. Our dataset includes the evolution of statement-level code coverage for every test case (either passed and failed), test result, all the builds information, code changes, and the corresponding bug reports. We conduct an initial analysis to demonstrate the overall dataset. In addition, we conduct an empirical study using T-Evos to study the characteristics of test failures in CI settings. We find that test failures are frequent, and while most failures are resolved within a day, some failures require several weeks to resolve. We highlight the relationship between code changes and test failure, and provide insights for future automated testing research. Our dataset may be used for future testing research and benchmarking in CI. Our findings provide an important first step in understanding code coverage evolution and test failures in a continuous environment.

Index Terms—Evolution and Maintenance, Mining Software Repositories, Software Testing

1 Introduction

Software systems are continuously evolving. To ensure system quality, developers nowadays often execute regression tests in a continuous integration (CI) setting. In CI, for every commit or a small set of code changes, developers would execute all the test cases to check whether the new code changes have caused any test failure. Therefore, when a test failure happens, developers may investigate the most recent code changes to identify the cause. Different from traditional software testing practices, which only run all the test cases before releases, CI allows developers to reduce the needed time to deliver the changes to the clients by catching test failures as early as possible and reducing integration overhead.

Prior studies have proposed various automated testing techniques that leverage code coverage or test execution information to assist developers with test failure diagnosis or repair. For example, researchers have proposed using code coverage information in failed test cases for tasks such as bug localization [1, 2, 3, 4, 5] and automated program repair [6, 7, 8]. These studies rely on the general assumption that the buggy code may be related to the execution path of a failed test case. However, most of prior techniques only consider snapshots of the project, while in CI settings, the code changes are integrated and tested continuously. The evolution data (e.g., recent code and coverage changes) may provide additional values that can help improve automated testing techniques and CI testing practices.

To help facilitate software testing research, many researchers have created several datasets and benchmarks. Just et al. [9] created the Defects4J benchmark that contains failed test cases caused by real bugs, code coverage of the failing test cases, and the corresponding fixes. Lin et al. [10] presented the QuixBugs benchmark that contains both passed and failed test cases of known bugs. Le Goues et al. [11] presented the ManyBugs and IntroClass datasets that allow researchers to reproduce many real-world bugs and benchmark automated program repair techniques. Elbaum et al. [12] collected test execution result over time at Google. Although these datasets provide great benefits to the research community, they also have common limitations. The datasets only provide a clean snapshot of the projects: namely, selected bugs, their fixes, and the corresponding code coverage. Yet, there is no project evolution information, such as how the code coverage evolves and the relationship between recent code changes and test failures, which is crucial in CI settings.

In this paper, we present T-Evos, a dataset that contains the evolution of test execution information across 12 Java projects over the course of 8,093 commits. For every commit, T-Evos contains the code coverage at the statement level (i.e., the specific lines that are covered), test status (i.e., pass or fail), build log, test execution stack traces, and code changes. To execute the test cases and generate individual test case coverage, we need to ensure that all the studied projects can successfully compile. To that end, we manually resolve the compilation issues, and then integrate the resolution into automation scripts. For each studied project, we also automate the manual process of adding project-specific

- *A.R. Chen and T.-H. Chen are with the Department of Computer Science and Software Engineering, Concordia University, Montreal, Quebec, Canada. S. Wang is with Department of Computer Science, University of Manitoba, Manitoba, Canada*
  E-mail: anr_chen,peterc@encs.concordia.ca, shaowei@cs.umanitoba.ca

configurations to collect code coverage. In total, we spent hundreds of hours resolving the issues that we encountered when executing the test cases. The size of the resulting dataset is around 3.3TB and took over 10 CPU years to generate. Different from prior datasets, T-Evos provides a fine-grained dataset on the evolution of test execution in CI settings, which may be used by future research that aims to leverage such continuous information to improve automated testing techniques.

We also conduct an empirical study using T-Evos to study the characteristics of test failures in CI settings. In particular, we answer the four following research questions (RQs):

RQ1: How often do test failure instances occur? We find that, on average, 34% of the studied commits contain test failures. Among the test failure instances, 44% are non-flaky test failure instances. The same test failure might occur multiple times, and most failures are concentrated in a small set of test cases.

RQ2: How long does it take for developers to fix a test failure? We find that while most test failures are resolved within one day, some test failures require several weeks to resolve. Future studies may use T-Evos to study the characteristics of the test failures and understand the factors that affect failure resolution time.

RQ3: How does the code change when the test failure first happens? We find that 18% of the test failures are related to modified or added test files that contain the failed test cases. We also find that when the test failures are first introduced, the test execution may stop prematurely when it encounters a failure in test files, which results in a decrease in code coverage.

RQ4: How does the code change when the test failure is resolved? We find that developers often resolve test failures by modifying non-code files (21%) or only test files (14%). We also find that there is only an average of 66% overlap between the files modified in failure-resolving commits and files executed in the failing commits. Moreover, when developers resolve the test failure, the failure-resolving changes do not always alter code coverage, even though the changes modify source code or test files.

We present the key contributions of this work as follows:

- We present T-Evos, a new dataset that contains information on how tests were executed in CI environments. To the best of our knowledge, T-Evos is the first dataset to show code coverage data in a continuous fashion.
- We conducted an empirical study on the test failures collected in T-Evos. The results may help future research understand the characteristics of test failures, such as their prevalence and persistence over time.
- Our results also highlight the relationship between code changes and test failure, and provide insights for future automated testing research.
- We summarized and discussed some potential future research directions that may be done using T-Evos.

Paper organization. Section 2 discusses our test execution and data collection process. Section 3 and 4 present the motivation, approach, and findings of our research questions.

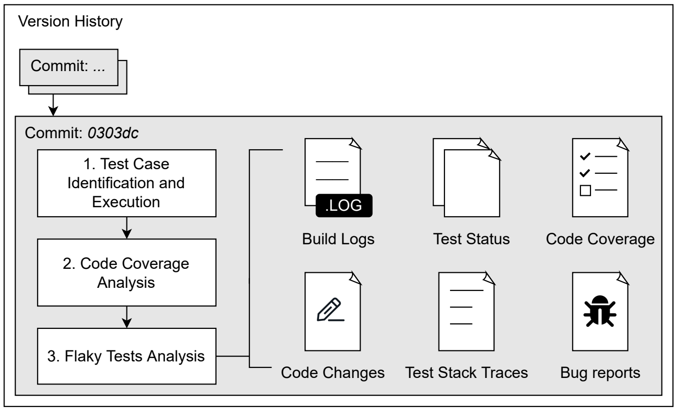

Fig. 1: An overview of our data collection process and the collected data.

Section 5 discusses future research directions. Section 6 discusses threats to validity. Section 7 presents the related work. Finally, Section 8 concludes the paper.

2 Test Execution and Data Collection

Continuous Integration is widely adopted in software evolution and most prior studies focus on test analysis at the release level or do not contain test execution information (e.g., code coverage) [9, 11, 12, 13]. However, there exists no dataset that collects fine-grained code coverage in a continuous environment (i.e., commit by commit over a period of time). Such dataset can benefit researchers in conducting various software studies (e.g., provide a realistic continuous integration setting, or develop a new technique that leverages the development evolution). Therefore, our goals are: 1) to provide a dataset that contains continuous test execution of statement level at the commit-level, which could be used for future research and benchmarking; 2) to conduct an empirical analysis on test failure and resolution in a CI environment, which may inspire future research on how to better leverage the code evolution information (e.g., code changes in prior commits) to assist developers in various aspects, such as improving test failure diagnose, quality assurance, and CI practices (particularly in testing).

Although many projects may execute test cases in a continuous fashion, the recorded code coverage data mostly contains only general coverage results (e.g., the branch coverage) without knowing which code statements were executed. Moreover, such code coverage data is only kept for a small period of time [14]. Thus, to achieve the goals, we need to execute the test cases for every commit, collect the *individual code coverage* for each test case, and analyze the changes between commits. To collect the code coverage at the statement level for each commit of the studied projects, our approach consists of three steps, as illustrated in Figure 1. First, for every commit, we build (i.e., compile) the project with the Maven Surefire plugin [15] to identify a list of test cases for each studied project. We need to perform this step for every commit because there may be newly added test cases, and some test cases may be deleted or disabled [16, 17]. We then execute the identified test cases, and analyze the code coverage information using JaCoCo [18] for the individual test case. Finally, some test cases may be flaky (i.e., sometimes pass and sometimes

TABLE 1: An overview of the studied projects, where *Total LOC* is the total lines of code, *Test LOC (%)* is the line of test code, and its percentage over total lines of code and *Average test cases* shows the average number of test cases per commit.

Project	**Total LOC**	**Test LOC (%)**	**Date range (Start, End)**	**Avg test cases per commit**	**Studied commits**	**Compiled commits**
commons-cli	12.2k	4.3k (35%)	(2002/06/10, 2020/09/19)	362	965	558
commons-codec	30.9k	14.6k (47%)	(2012/08/20, 2020/10/11)	799	1,000	709
commons-compress	65.8k	23.1k (35%)	(2017/01/07, 2020/10/13)	1,006	1,000	337
commons-csv	10.3k	7.2k (70%)	(2013/03/20, 2020/10/03)	230	1,000	936
commons-math	199.2k	76.0k (38%)	(2015/04/26, 2020/08/10)	4,200	1,000	551
gson	38.4k	15.2k (40%)	(2011/03/29, 2020/05/13)	1,103	1,000	942
jackson-core	50.8k	19.0k (37%)	2014/12/05, 2019/12/30)	320	1,000	453
jackson-dataformat-xml	564.7k	7.6k (1%)	(2014/11/12, 2020/10/16)	270	1,000	203
jfreechart	159.1k	40.0k (25%)	(2013/08/15, 2020/03/10)	2,477	1,000	439
jsoup	86.8k	9.4k (11%)	(2011/07/02, 2020/03/07)	532	1,000	990
junit4	37.9k	20.4k (54%)	(2013/02/05, 2021/02/13)	826	1,000	997
fastjson	354.6k	138.0k (39%)	(2018/09/03, 2021/04/05)	4,768	1,000	978

fail). Flaky tests can introduce biases in the dataset when analyzing test failure [19, 20, 21, 22]. To reduce the bias, we detect and remove such flaky tests from the dataset [23].

The overall process is challenging and requires a significant amount of both manual and computing effort, because 1) a project may contain thousands of test cases and we need to run each test case for every commit to collect all the statement-level coverage information; 2) To make each test case run, we need to manually resolve many compilation issues. Below, we discuss our data collection, test execution process, and the associated challenges in detail.

2.1 Test Case Identification and Execution

Studied Projects. We select 10 projects that are also used in the Defects4J benchmark [9] for our study. We choose these projects because Defects4J is widely used and the research community is familiar with the projects. Note that, the data in Defects4J only contains snapshots of a subset of the test execution (e.g., the failing test cases before a bug is fixed), while our goal is to collect continuous test execution and failure data on a per-commit basis. To increase the diversity of the studied projects, our study includes additional projects (i.e., fastjson and junit4) of different software foundations. We query the top Java Maven projects on GitHub based on the number of GitHub Stars. The two projects, fastjson and junit4, have 23.7k and 8.2k GitHub Stars, respectively. The selected projects are well-maintained and contain active tests. We also manually verify that the selected projects can be successfully built and the test cases can be executed. In total, we conduct our study on 12 projects and Table 1 presents their details. For each project, we execute the test cases in all the commits (i.e., code changes) specified in the time range shown in Table 1. On average, each commit executes from 230 to over 4,000 test cases. We study the 1,000 latest commits at the time when our analysis is conducted. We chose to study 1,000 commits in each project because it is a relatively large number, but still feasible for us to manually resolve compilation issues. Based on the first and last commit of the 1,000 commits, we then compute the start and end date of our study. In other words, the time range refers to the time between the latest commit (at the time of our data collection) to 1,000 commits prior to it. We determine those commits by executing the "`git log -n 1000`" command. Note that for the project commons-cli, the repository only contains 965 commits at the time of the study.

Handling compilation issues. Compiling the projects is not always straightforward. We perform the following steps to ensure that we can compile the projects and execute test cases in the studied commits. First, each project requires a considerable number of manual configurations to successfully compile. For instance, some projects may require a specific version of Java Runtime or Java Development Kit (JDK). Therefore, to automate the compilation and test execution process, we manually resolve all the compilation issues that we encounter and integrate the resolution in the automation scripts (e.g., changing JDK versions if needed). Second, some projects may not use JaCoCo [18] for code coverage analysis, so we need to manually configure the maven build script to add the JaCoCo dependency. Since the projects may have multiple modules, we need to identify all the maven build scripts when adding the JaCoCo dependence. Moreover, most of the studied projects use a handful of third-party libraries, so there may be other dependency issues that require manual fixes (e.g., some libraries are no longer available in the central Maven repository and need to be manually downloaded). We manually resolve the issues and update our automation scripts accordingly. Finally, even after the projects are successfully compiled, there may still be commit-specific configuration that prevents the test cases from executing. For instance, developers might refactor the structure of the configuration file (i.e., pom.xml) that specifies the dependencies. Thus, we need to modify our code accordingly to add the JaCoCo dependency. It took hundreds of hours of manual effort to resolve the compilation issues. Although we tried our best to resolve compilation issues, some commits may not be compilable even after we tried to manually resolve the issues due to reasons such as lack of dependencies. In total, we were able to successfully compile 8,093 commits.

Identifying test cases. After we successfully compile the projects, our next step is to identify a list of test cases that we need to execute. As found by Kim et al. [17], developers may disable some test cases during development. Therefore, we need to first identify the test cases that are active and will be executed in the continuous integration process. Since all the studied projects use Maven as the build system, we track the list of active test cases by compiling the projects using the Maven Surefire plugin [15]. More specifically, when the project is built and compiled, the Maven *test* and *verify* lifecycles execute the unit and integration tests, respectively. Once the test cases are executed, the Surefire plugin then generates a *surefire-reports* folder containing the information of all the executed test suites (i.e., test class) in XML format. We then parse the XML files to identify the list of test cases (e.g., test methods annotated with @Test in the test class) that belong to each executed test suite.

Executing the test cases. Since we have 12 studied projects and each with hundreds of commits to compile and run test cases, we used six servers to speed up the execution. All the servers have the same hardware specification: a four-core Intel Xeon (Skylake, IBRS) CPU (2.10 GHz) and 5 GB of RAM. Four of the servers ran Ubuntu 20.04, while the other two ran Ubuntu 18.04. To collect the coverage of each individual test case, we need to run each test case one-by-one. Therefore, we implement scripts to automate the process of checking out a commit (i.e., in a Git repository), updating the Maven build file to include needed dependencies such as JaCoCo [18], compiling the project, and running each individual test case separately. We need to run each test case separately because JaCoCo only generates statement level coverage information (i.e., which specific lines a test case covers) when the test cases are executed one-by-one. To avoid dirty states that may affect the test result, we also automate test cleanup before and after running each test case. Note that we re-executed test cases in different servers to make sure that the nature of the failures is not sensitive to its environment or configuration (e.g., executing a test at a different time zone might cause a test to fail).

We created five worker threads in each of the six virtual machine servers. Each worker has its own sets of commits and projects to run to further parallelize the test execution process. Despite that the execution of the test case requires no re-compilation, Maven performs additional checks before the test execution (e.g., predefined coding style checks), which adds additional overheads to every test case execution. The building and testing time of each commit vary from 2 to 32 minutes, while the time of running a single test case is usually less than a minute. In total, our data collection and test execution took over 10 CPU years.

2.2 Code Coverage Analysis

We want to collect the detailed coverage of each test case in a continuous fashion. Hence, we conduct code coverage analysis on every commit.

Integrating Code Coverage Tool. We use JaCoCo [18] to generate the code coverage report. JaCoCo is one of the most popular code coverage tools that instruments bytecode to trace the execution during the test run. We integrate JaCoCo as a Maven plugin. While the integration is straightforward for most Maven projects, for the multi-module Maven projects, we need to manually modify the Maven configuration to collect code coverage. More specifically, when integrating JaCoCo into a module, the collected code coverage is limited to the classes of that module. However, in the case of a test case covering several different modules in the system (e.g., integration tests), the out-of-module class coverage will not be shown. Therefore, when some test cases cover multiple modules, we add an extra *report-aggregate* goal to the parent Maven build script (i.e., the pom file). Additionally, we add the JaCoCo plugin to every module. Every time a test is run, JaCoCo updates the covered classes in the coverage report of the module that they belong to. A coverage collector, which we implemented, will then parse and merge the coverage from every module.

Analyzing Code Coverage Results. Once the test is executed, we collect the code coverage of the individual test from the JaCoCo report at three different levels of granularity: 1) covered classes, 2) covered methods, 3) covered line of statements. In addition, we also collect the test status (i.e., passed or failed) and some general coverage metrics (i.e., branch coverage).

2.3 Flaky Tests Analysis

Prior studies [20, 22, 23] found that some test cases may be flaky. Namely, the test result may sometimes pass and sometimes fail, even if the code remains the same. To reduce the noise caused by flaky tests when analyzing test results, we remove flaky tests from our list of active test cases. We use DEFLAKER [23] to compare the code change with the coverage of the failing test cases to identify the flaky tests. A failed test case is considered flaky if there is no overlap between the failure introducing code changes (i.e., the test case fails after the code changes) and the coverage of the failed test case. Namely, the failed test cases do not have overlap with the code changes. For every newly observed test case failure, we analyze whether any of the changed code is covered by test case, and if not, we mark the test failure as flaky.

2.4 Bug Reports Identification

To provide more information on the collected commits, we also identify the corresponding bug reports through the commit messages. The studied projects use the JIRA issue tracking system, where each bug report receives a unique bug report identifier (e.g., CLI-121). In addition, all 12 projects follow standards of including the bug report identifier in commit messages. Therefore, we use the git command "`git show -s --format=\%B`" to mine the commit messages, and use regular expressions to capture bug report identifiers. For instance, for the studied project commons-cli, we use the regular expression "`CLI-\\d+`". Then, we further leverage JIRA APIs [24] to verify that the detected identifiers are of type *Bug* rather than other types (e.g., *Feature request*). In total, we collect 221 bug reports.

2.5 The Collected Data

Figure 1 shows an overview of our data collection process and the collected data. At the end of our test execution and data collection process, we built 8,093 commits across 12 studied projects. The dataset includes: 1) all the build logs that were generated when compiling every studied commit; 2) the test status of every executed test case; 3) the code coverage of every passed and failed test case and how the coverage evolve overtime; 4) the code or configuration changes that developers made over time and their effect on the test result (e.g., a test case passes or fails after the change); 5) the stack traces generated by failed tests; 6) the associated bug reports. In total, the size of our collected data is 3.3 TB. Future studies may use our collected data to understand various testing practices in CI and help improve automated testing techniques.

TABLE 2: The total studied commits and the commits that contain test failure. *Test passing instances* show the number of times that a test executes without failure across all studied commits. *Test failure instances* show the number of failures that occur across all studied commits (a test failure may remain unresolved across multiple commits producing instances of test failure). *Test failures* show the number of new/resolved test failures (if a test case fails multiple times consecutively, it is counted as one test failure).

Project	**Total commits**	**Commits with failure (failure ratio)**	**Test passing instances** (in millions)	**Test failure instances**			**Test failures**	
				Total	Flaky	Non-flaky	New failure	Resolved failure
commons-cli	558	121 (22%)	0.17	183	53	130	60	60
commons-codec	709	340 (48%)	0.44	4,351	1,105	3,246	58	47
commons-compress	337	182 (55%)	0.29	3,812	3,697	115	192	191
commons-csv	936	23 (2%)	0.23	80	28	52	32	32
commons-math	551	95 (17%)	3.33	221	188	33	2	2
gson	942	253 (27%)	0.84	6,129	4,949	1,180	53	43
jackson-core	453	344 (76%)	0.48	2,672	162	2,510	99	97
jackson-dataformat-xml	203	153 (76%)	0.04	156	4	152	10	10
jfreechart	439	342 (78%)	1.02	1,253	635	618	27	13
junit4	997	321 (54%)	0.91	386	347	39	35	35
jsoup	990	42 (5%)	0.49	66	0	66	23	23
fastjson	978	333 (35%)	3.99	553	10	543	230	230
Total	8,093	2,724 (34%)	12.23	19,862	11,178	8,684	821	783

3 Studying the Characteristics of Test Failures in CI Settings

As discussed in Section 2, we execute the test cases in over 8,093 commits to collect information such as code coverage and test status. To the best of our knowledge, we are the first to study and provide a dataset that contains commit-by-commit code coverage and failure information throughout the version control history. We believe that such dataset has great potentials for future research. Therefore, we conduct an analysis to provide an initial overview. Specifically, in this section, we provide an overview of the collected dataset and study test failure introduction and resolution across the studied time period. In particular, we answer the two following research questions:

- RQ1: How often do test failure instances occur?
- RQ2: How long does it take for developers to fix a test failure?

3.1 RQ1: How often do test failure instances occur?

Continuous testing in CI helps expose software faults that might negatively impact the functionality of the system under test. A prior study [25] found that test failures constitute the main reason why builds fail in CI. Labuschagne et al. [26] showed in their study that 18% of test executions in CI fail. However, previous studies only consider instances of test failure that last for exactly one commit, rather than a prolonged period of time. In such cases, we cannot have a complete picture on the prevalence of test failures throughout the project evolution, since the same test can fail repeatedly (e.g., spanning multiple consecutive commits). Therefore, in this RQ, we study test failures both in their prevalence of happening on the individual commit, and the prevalence of the same test failure lasting through consecutive commits.

To better understand the prevalence of test failures in the studied projects, we identify and track each test failure. To ease the explanation, we refer to a test failure that happens across at least one consecutive commit (i.e., without being resolved) as a *test failure*, and an instance of test failure that occurred in a single commit as a *test failure instance*.

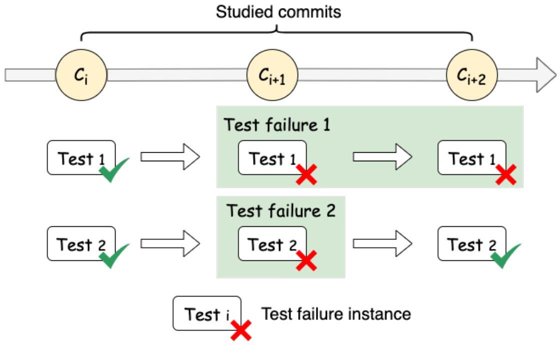

Fig. 2: Example of test failures and test failure instances.

We further illustrate them using our example shown in Figure 2. Given that *Test 1* fails at commits C_{i+1} and C_{i+2}, a test failure is the consecutive test failure happening across commits C_{i+1} and C_{i+2} (illustrated as *Test failure 1*), and the test failure instances are the individual failures happening on C_{i+1} and C_{i+2} (which may be the same test failure). We provide quantitative details of the test failures at two different levels of granularity: commits that contain failed test cases and individual failed test cases (i.e., a test case may fail multiple times before it is resolved). We further analyze the test failures to investigate how many are newly introduced and how many are resolved during the continuous process. Our findings provide a more comprehensive understanding of the test failures and their distribution over commits in the context of continuous integration, and provide insights for future research.

A considerable number (34%) of the analyzed commits contain at least one test failure instance. Table 2 presents an overview of the prevalence of test failures for each studied project. In total, we compile and execute the test cases in 8,093 commits across the studied projects. Among these 8,093 commits, we find that 2,724 (34%) of them contain at least one test failure instance. The percentage of the commits that contain at least one test failure instance ranges from 5% to 100%. Our finding shows that although the studied projects, in general, contain a large number of failed commits, the prevalence of test failures varies noticeably among projects. Regression testing in CI aims to guarantee the quality of software with each commit. However, we observe different degrees of test failure prevalence across the studied projects. Future research may use our dataset to investigate the reasons that may cause such differences across projects.

We find that flaky test failure instances account for over 58% of the test failure instances. For non-flaky test cases, the same test failure might occur multiple times in the sequential commits. We find that flaky test failure instances are common in the studied projects. Out of the 19,862 test failure instances found across 8,093 commits, we observe 11,178 (56%) of them are flaky (shown in the *Test Failure Instances* column in Table 2). In contrast, 44% of the test failure instances are identified as non-flaky. Our finding shows that failures caused by flaky tests may be common. Note that a non-flaky test case may remain unsolved and fail multiple times across the studied commits. Therefore, we further study the number of new and resolved *test failures* in each project from the non-flaky test failure instances. The *New failure* column in Table 2 shows the number of newly introduced failures (i.e., a test case fails after a commit). Similarly, the *Resolved failure* column shows the number of resolved failures (i.e., the test case no longer fails after a commit). Overall, we find that there were only 821 new failures and 783 resolved failures. Given the small number of resolved test failures and the large number of test failure instances, our finding shows the same test failure might occur multiple times for the test failing repetitively in the sequential commits. Future studies may be needed to study these repeatedly failed test cases, whether flaky or not, and how they may affect software quality in general.

Most failures are concentrated in a small set of test cases. To study how the failures are distributed across the test cases, we count the number of *unique test cases* that result in failure across the studied commits. Note that we count a test case as one unique failing test case even if the test case has failed and been resolved more than once. We find that there are only 332 unique failing test cases (i.e., out of 8,684 non-flaky test failures) across all projects. This result implies that test failures are concentrated in only a small number of test cases, and most test cases never fail. After some manual investigation, we find that these test cases are often related to the main functionality or complex business logic of a project. For example, jsoup is a HTML parser and one of its core features is to parse HTML documents into *Document* Java objects. The same set of test cases in the test suite *DocumentTest* (which tests object conversation and parsing) fail multiple times during the studied period, although the fixes were applied at different locations and for different reasons. One possible reason may be that some parts of the code undergo more changes, so the corresponding test cases are more likely to fail.

Our dataset provides a complete picture of how often test failures occur across commits in the CI context and the distribution of such test failures across commits and test cases. Future research may use the dataset to study how code evolution causes test failures and how to prioritize test execution. In addition, one interesting point as revealed by the results is that test failures are concentrated in only a small number of test cases, while most test cases never fail. Future research may further study the quality and effectiveness of the tests that never fail.

> Test failure is prevalent in the evolution of the 34% of commits that contain test failures. Among the test failure instances, 44% are non-flaky test failure instances. We also find that many test cases fail multiple times across commits, and most failures are concentrated in a small set of test cases.

3.2 RQ2: How long does it take for developers to fix a test failure?

In the CI context, the detection of test failures presents a compensatory benefit of continuous testing, especially when failures detect real faults. However, prior research found that test failures, despite being detected, might not be resolved for various reasons. Beller et al. [27] found that up to 30% of the failing tests are not repaired immediately although developers detect them directly in IDEs. Rogers [28] found that sometimes, developers might allow known test failures into CI, as long as those failures are resolved by the end of the development iteration. However, it is unknown how long test failures last in evolutionary settings. Therefore, in this RQ, we study the resolution time it takes for developers to fix a test failure, and how failures are distributed at different resolution times.

To calculate how long it takes for developers to resolve a test failure, we analyze every test failure in the version history and look for the *failure-introducing commit* (i.e., the first commit in which the test failure occurs) and the *failure-resolving commit* (i.e., the commit where the test failure is resolved). Then, we compute the time difference between the failure-introducing commit and the failure-resolving commit. Note that if the same test case fails again after a resolution, we consider it as a different test failure as developers have already made changes to resolve the failure.

While most test failures are resolved within one day, some may require more than a week to resolve. Table 3 shows the resolution time of the test failures. The mean resolution time across all the studied projects is *1* day, while the average of the median resolution time is less than *1* day. In addition, 10 out of 12 studied projects have more than 75% of the test failures resolved within a day. More than one third of the studied projects do not have any test failure extending for more than 7 days. On average, across the projects, more than 75% of the total test failures are resolved within a day, and only 5% of the failures persist more than 7 days. While our findings show that developers are actively trying to resolve the test failure once they occur, there are still some exceptional cases.

As shown in Table 2, while most of the studied test failures (783 out of 821) are resolved, there are still 38 test failures that were introduced but never resolved. Thus, we further studied those test failures, and calculated how long they are lasting. We find that most of the failures happen near the end of the studied periods of the projects. Around half of these 38 test failures were introduced no more than three days before the end of the studied periods. The maximum time difference between their introducing time

TABLE 3: The resolution time of test failures.

Project	**Resolution time**					
	Min	Max	Mean	Median	< 1 day	> 7 days
commons-cli	< 1 day	63 days	4 days	< 1 day	77%	12%
commons-codec	< 1 day	102 days	2 days	< 1 day	87%	2%
commons-compress	< 1 day	3 days	< 1 day	< 1 day	83%	0%
commons-csv	< 1 day	2 days	< 1 day	< 1 day	97%	0%
commons-math	< 1 day	< 1 day	< 1 day	< 1 day	100%	0%
gson	< 1 day	15 days	4 days	1 day	49%	23%
jackson-core	< 1 day	53 days	2 days	< 1 day	60%	4%
jackson-dataformat-xml	< 1 day	< 1 day	< 1 day	< 1 day	100%	0%
jfreechart	< 1 day	< 1 day	< 1 day	< 1 day	100%	0%
junit4	< 1 day	22 days	4 days	1 day	43%	26%
jsoup	< 1 day	17 days	< 1 day	< 1 day	96%	4%
fastjson	< 1 day	16 days	< 1 day	< 1 day	88%	1%
Average	< 1 day	21 day	1 day	< 1 day	75%	5%

and the ending period of the studied projects is no more than 23 days. In short, it is possible that the test failures were not resolved due to the time periods that we analyzed did not include their fixes.

By calculating the mean resolution time, we observe that most test failures, if ever resolved, are resolved within one day. Even when they are not resolved, they are new failures that were introduced for no more than 21 days. However, some projects contain a maximum test resolution time that is significantly longer than the majority of the dataset (e.g., several weeks compared to within one day). Our dataset identifies the test failures that may have different characteristics, which causes the fixing time to be much longer. Future testing research may use our dataset to better understand the characteristics of such long-lasting test failures, and further assist developers with improving code quality.

> While most of the test failures are resolved within one day, we still find some failures that take more than a week to resolve. Future research may use our dataset to study the characteristics of the test failures and understand the reasons for such differences.

4 Studying Code Changes and Their Relationship With Test Failures

Our collected dataset includes the complete code coverage evolution at the statement level and the code changes that developers made throughout the version control history. In this section, we study the changes that developers made when introducing/resolving test failures, and how code coverage change before and after fixing the failure. Our findings provide an understanding on the relationship between code coverage and test failure, and provide insights and a new dataset for future automated testing research such as bug localization. In particular, we answer the two following research questions:

- RQ3: How does the code change when the test failure first happens?
- RQ4: How does the code change when the test failure is resolved?

4.1 RQ3: How does the code change when the test failure first happens?

Prior research studied the characteristics of test failures in relation to the source code. Pinto et al. [16] showed that test execution might fail for three major reasons including removal of the required source class or method, catching runtime exceptions, and assertion violation. Marsavina et al. [29] further examined, in the case of introduction of test failure, how production and test code were changed. They found that many failed test cases may be added by developers while working on production code. In this RQ, we study the changes that developers made when introducing test failures, and how code coverage changes (which shows the dynamic execution information) before and after introducing the failure. Studying the change in code coverage may give insights on why test failure happens and provides an understanding of the relationship between code coverage and test failure.

To better understand the evolution of code coverage when the test failure first happens, we record the code coverage of each test failure. We provide quantitative analyses to study the test failure in CI settings where code coverage is one of the few pieces of information available to developers. First, we investigate how developers introduce test failures. In particular, we derive categories of code changes based on the types of files changed, and present the number of test failures belonging to each category. By studying what type of code changes might introduce the test failures, future research might inspire from our findings to better help developers with test failures. Then, we evaluate the impacts of test failures on code coverage. In other words, when test failures occur what can we observe from the code coverage? We present the coverage change based on the total line coverage increased and decreased.

To investigate which files were modified, for each test failure, we conduct quantitative analyses on the failure-introducing commits. We categorize the files with *.java* extension as either source code or test files, and otherwise as non-code files (e.g., data and configuration files). We use the list of active tests identified in Section 2.1 to further distinguish between source code and test files.

To investigate how the code coverage changes after failure introduction, we first compute the per-method line increased and decreased from the failure-introducing commit and the commit before it. Rather than checking whether the overall coverage increases or decreases, we calculate the individual line increased and decreased in each covered method. In this way, we can obtain finer-grained results and identify the case where the line coverage has changed but the overall coverage remains the same. Then, we sum

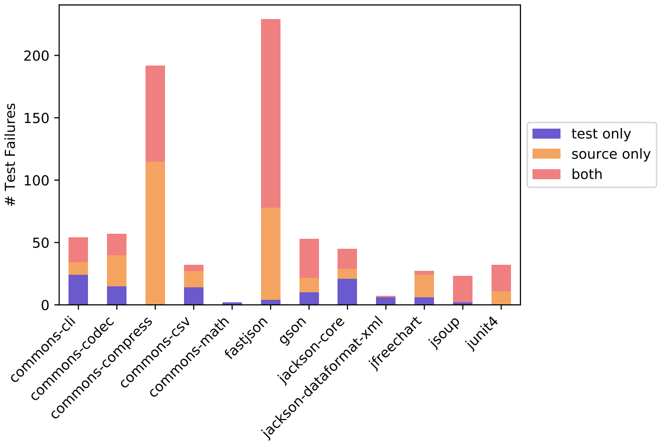

Fig. 3: The categories of the files that were modified in failure-introducing commits.

TABLE 4: Average code coverage before and after the failure-introducing commit.

Project	Method level		Line level	
	Increased	Decreased	Increased	Decreased
commons-cli	10	12	36	68
commons-codec	4	8	18	56
commons-compress	5	16	16	82
commons-csv	2	26	7	136
fastjson	15	27	329	511
gson	0	28	1	511
jackson-core	15	17	89	101
jfreechart	7	18	49	119
junit4	13	29	54	124
Average	8	20	67	190

TABLE 5: Test failures with and without coverage change for different introduction categories.

Test Only		Source Only		Both	
Changed	Unchanged	Changed	Unchanged	Changed	Unchanged
26	38	140	74	159	31

up the per-method line increased and decreased from all the covered methods to get the total lines increased and decreased. We quantify the coverage change based on the total line increased and decreased. As the line coverage counts the lines of code without including the conditional statements (e.g., `if` and `while`), we further compute the branch coverage in the case where we observe no difference in line coverage. Similar to the line coverage calculation, we calculate the individual branch increased and decreased from each covered method.

We find that many failed test cases may be added by developers while trying to resolve an issue. Figure 3 shows the types of files that were modified in failure-introducing commits. Overall, we find that many failure-introducing changes modify both source code and test files (52% on average), while just 35% modify only source code files. We also find that many of the commits that modify both source code and test files may be adding new test cases to address newly reported bugs. For instance, in fastjson, which is a JSON processor for data streaming, 63% of its failure-introducing commits (i.e., 147/233) modify both test and source code files. We find that nearly 99% (146/147) of those modifications added new test files to the project. Those new test files either contain or are named after some bug ID (e.g., *Issue2133*), which may indicate that the test files were added when developers were trying to resolve the bugs. Overall, we find that 18% (150/821) of the test failures in the studied projects either modified or added test files that contain the failed test cases. Moreover, 13% of the failure-introducing commits modify only test files. In other words, developers may be fixing a bug while modifying or adding test files.

Automated testing techniques (e.g., bug localization) use the information of running test cases to assist developers in identifying bugs. Our finding shows that, if a dataset is collected without considering such newly added test cases that are used to resolve the bugs, there may be potential biases in the result. Namely, developers were already trying to address the bug by adding new test cases based on their knowledge. For instance, if such newly added (and failing) test cases were used for evaluating techniques such as bug localization, one may be implicitly using developers' knowledge of the buggy location to assist the automated techniques. In contrast, our dataset specifically shows whether a failure-introducing commit modifies either source code, test file, or both. Future studies may use our dataset to filter out those newly added or modified test cases (e.g., the test is being modified by developers to capture the bug, so it is failing) to better evaluate automated testing techniques such as bug localization.

Test execution may stop prematurely when it encounters a failure in test files (e.g., assertion statement is invalid), which results in a decrease in code coverage. Table 4 shows the changes in the average code coverage (i.e., averaged across all failure-introducing commits in a project) before and after the failure-introducing commit. We report the changes in both method-level and line-level coverage. To note that we perform our coverage analysis on 468 (57%) new failures instead of all the new failures (821), as not all failures have code coverage change between the failure-introducing commit and the prior commit. In addition, there were some test cases that did not exist before the failure-introducing commit (i.e., new test cases).

Overall, we find that the failure-introducing commits decreased more coverage than increased. For example, the average decreased covered lines are 190, while the increased covered lines were only 67. We notice that since we generate the coverage based on the executed test code, when a test case fails, the part of the test code that is beyond the failing location will not be executed and will not be included in the coverage data. If a test case fails at the beginning of the execution, the code coverage may be empty in some cases (e.g., the pre-test check fails). For example, we observe a serialization test (i.e., *Issue3436#test_for_issue*) from fastjson, which fails with a *JSONException*. The failure happens at the beginning of the test case, so the remaining test code is not executed and no code coverage report is generated.

Our dataset provides both the code coverage before the failure-introducing commits and the code coverage of the failed commit. Future studies may use our dataset to study how the coverage evolution data may help locate where a bug is introduced when the coverage of the failed test cases is incomplete.

Around 6% of the failure-introducing changes do not

change the line coverage, nor the branch coverage. When analyzing the coverage change with different categories of failure-introducing changes, we find that around 44% (200/468) of the failure-introducing changes do not change line coverage, even though the changes modify source code or test files, as described in Table 5. While those failures do not have changes on the line coverage, 87% (174/200) of them do have a different branch coverage. The remaining 13% (26/200) of the test failures do not involve coverage change at all (e.g., the failed assertion statement is the last line in the test case and the test failure is due to incorrect test setup or variable value).

As we found, the coverage (i.e., line or method coverage) might not always change when the test failures are first introduced. Our dataset and findings may inspire researchers and practitioners to further investigate the prevalence of software regression and refactoring that caused test failures.

> Many failed test cases may be added when developers are trying to resolve an issue, and 13% of the failure-introducing commits only modify test files. We also find that, compared to the prior passing commit, the code coverage generally decreases in failure-introducing commits.

4.2 RQ4: How does the code change when the test failure is resolved?

Understanding the changes that developers apply when resolving a test failure may help improve future automated testing techniques. In this RQ, we study the types of files that are modified in failure-resolving commits, and how do code coverage changes when the failure is resolved. Specifically, we answer the RQ by answering two sub-RQs: *What types of changes do developers apply when fixing a test failure?* and *Are there overlaps between the code coverage of the failed test cases and the failure-resolving location?*

RQ4.1: What types of changes do developers apply when fixing a test failure?

We study the test failure resolution in two steps: 1) we investigate which files were modified during the resolution, and then 2) we study how the coverage changes after failure resolution. To investigate which files were modified, for each test failure, we examined the changed files in failure-resolving commits. Same as RQ3, we categorize the files with *.java* extension as either source code or test files, and otherwise as non-code files (e.g., data and configuration files). To investigate how the code coverage changes after failure resolution, similar to RQ3, we quantify the coverage change based on the total line increased and decreased. In other words, we sum up the per-method line added and deleted from all the covered methods to get the total increased and decreased lines. Knowing which files were modified during the resolution and how the coverage changes may help better understand how do developers fix a test failure and improve automated testing techniques.

Developers often resolve test failures by modifying non-code files (21%) or only test files (14%). Figure 4 shows the distribution of the files that were modified in failure-resolving commits. Overall, we find that 34% of the failure-resolving commits modify both test and source code files, and 31% of the commits modify only the source code files. We also find that developers commonly modify non-code files (21%), or only the test files (14%) in failure-resolving commits. The types of modified files vary across the studied projects. For instance, in fastjson, we observe 91% of the failure-resolving commits modify either only source code files, or both test and source code files. Only 6% of failure-resolving commits modify test files. However, in another project (i.e., commons-codec), we observe that more than 36% of the test failures are resolved by modifying only test files. In other projects, such as gson and jackson-core, developers might also only modify the non-code files, such as data or configuration files, to fix the test failures. We observe more than 49% and 76% of the failures are fixed through non-code changes in gson and jackson-core, respectively. Our findings show that the resolution of the test failure does not only limit to source code files, but also the test and non-code files. When studying test failures, future research might consider non-code files as a potential fix for failures, as well as the test files that might already contain some issues related to the failures.

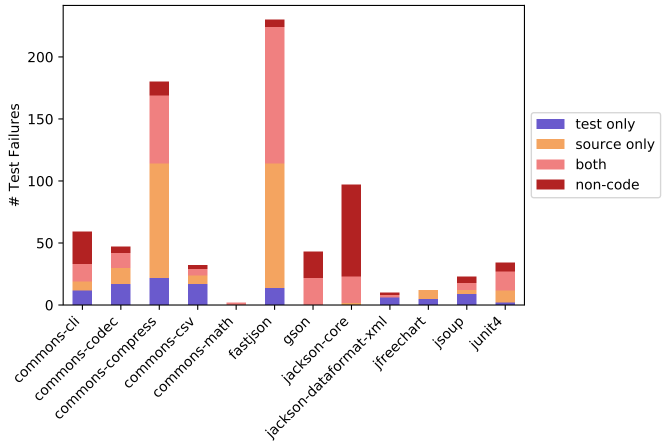

Fig. 4: The types of files that were modified in failure-resolving commits.

TABLE 6: Test failures with and without coverage change count for different resolution categories.

Test Only		Source Only		Both	
Changed	Unchanged	Changed	Unchanged	Changed	Unchanged
71	22	183	46	130	21

Around 19% of the failure-resolving changes do not alter code coverage, even though the changes modify source code or test files. In Table 6, we show the number of resolved test failures with and without coverage change. We find that there is a non-negligible number of failure-resolving changes that did not change code coverage. 23.7% (22/93), 20.1% (46/229), and 14.0% (21/151) of the test-resolving commits did not change code coverage in each resolution category (i.e., modify only test files, only source code files, or both), respectively. To note that we perform our coverage analysis on 473 (60%, out of 783) among all the resolved failures because the code coverage information may not be available for the failure-resolving commit or the

TABLE 7: Overlaps between the coverage of the failed test case and the failure-resolving location.

Project	commons-cli	commons-codec	commons-compress	commons-csv	fastjson	gson	jackson-core	jfreechart	junit
Overlaps(%)	64	76	65	90	71	14	58	100	58

commit before (i.e., the failing commit). Some test cases may be removed in the failure-resolving commit (as found by previous research [30, 31]), and there may be no coverage for some failing test cases (e.g., a test case fails early when no coverage is available yet). In addition, as discussed above, developers may modify non-code files (e.g., test configuration or data files) that are not visible through code coverage. The data files may be used in test cases to verify the expected test output, and some configuration issues may cause test cases to fail.

To better understand in what situation do failure-resolving commits have no code coverage change, we manually studied the test failures. We performed our manual study on all 310 resolved test failures in which the failure-resolving commits did not introduce any code coverage change. The first author manually examined the code changes applied to the failure-resolving commit, and the code coverage. This information provides hints on the relationship between the changed code and test execution. By leveraging this information, the first author uncovered a list of categories of code changes that may result in no code coverage change. Then, the second author systematically verified the assigned categories. In case of any discrepancy, the two authors further carried on discussions to reach a consensus. We observed four main types of changes: 1) *test assertion changes*, 2) *method parameter changes*, 3) *invisible dynamic changes*, 4) *conditional statement changes*. Each of these four types of changes can result in resolved test failures without coverage change. We briefly present the four types of changes as follows:

Test assertion changes: changes performed on assertion statements in the test cases. As only the assertion statement is modified, the dynamic execution is unchanged. For instance, one test failure captured in fastjson modified the expected string value in the assertion statement `assertEquals(expected_str, actual_str)` to adapt to recent changes in the source code that necessitate a new expected string value in the test.

Method parameter changes: changes that modified the parameter value inside method invocations. An example of such is correcting a wrong parameter to fix the test failure.

Invisible dynamic changes: changes that modified the flow of the third-party code which is not reported in the coverage report. For example, modifying the string format of a datetime object before passing it to a special third-party json datetime formatter (where the software will use the output).

Conditional statement changes: changes that modified the conditional statements (e.g., `if-else` or `while`) to fix logical errors. For example, developers may add a new condition (e.g., from `while(condition1)` to `while(condition1 && condition2)`) to fix a logical error. However, since the change does not add new code and the test may not be updated, there is no coverage change.

Our findings show that the resolution of test failure is not always visible through code coverage change. Moreover, the system dynamic behavior may change, even if developers fix the test failure and the code coverage remains the same. Future research may use our dataset to further evaluate whether existing automated testing techniques (e.g., bug localization or automated program repair) need to be tailored to work on this type of issues.

RQ4.2: Are there overlaps between the coverage of the failed test cases and the failure-resolving location?

Many automated testing techniques (e.g., fault localization) leverage code coverage to help developers identify buggy code [1, 2, 3, 4, 5]. The general assumption of such techniques is that the buggy code is on the execution path of a failed test. In this RQ, we wish to investigate whether the coverage of the failed test can provide useful insights on the failure-resolving location. By understanding this, we may provide insights for future research.

Here, we analyze the results of 473 test failures that have the code coverage information. We first extract a list of covered files (f_{covered}) from the coverage information of the commits that have failing test cases. Then, we compare f_{covered} with the changed files in failure-resolving commits (f_{changed}) and compute an overlap between them. We calculate the percentage of the overlap based on the total number of changed files as $\frac{\#f_{\text{covered}} \cap \#f_{\text{changed}}}{\#f_{\text{changed}}}$.

In all the studied projects except one, many modified files in the failure-resolving commits do not have overlap with the executed files in the failing commit. Table 7 shows the overlaps between f_{covered} and f_{changed}. We find that a notable number of the changed files are not on the execution path covered by the test cases in the failure-resolving commit. The average percentage overlap across the studied projects is 66% (except for jfreechart, where the overlap is 100%). We observe that the reason for a relatively low overlap may be that the coverage of failed test cases may be incomplete if the test case fails during the early execution stage. Such incomplete code coverage might present a limitation to existing automated testing techniques that analyzes code coverage (e.g., fault localization). We also find a non-trivial number of instances (i.e., 10% of the overlaps) where the failure-resolving commits only modify the failed test case and the coverage did not change. By examining these instances, we observe that some of the test failures were resolved in unconventional ways. For instance, as also reported in a prior study [17], developers may disable the assertion statements in the test code (e.g., the code is commented out), but the issue remains unsolved.

Developers often resolve test failures by modifying non-code files (21%) or only test files (14%). Even when modifying source code or test files, around 19% of the failure-resolving changes do not alter code coverage. In addition, in all studied projects except one, we observe that a notable number of the changed files are not on the execution path covered by the test cases in the failure-resolving commit.

5 Future Research Directions

As described in Section 2, we executed the test cases on consecutive sequences of commits in 12 studied projects. We collected the data and made it publicly available [32]. Our data is 3.3TB and may be used in future testing research or benchmarking different techniques. Below, we discuss some possible research directions using our data.

Studying and understanding quality issues in test code: In RQ4, we found that developers may only modify test files to resolve test failures. We also found that some test cases may be failing already when they were first added. Our findings indicate that there may be pre-existing issues causing test cases to fail. Future studies may use our dataset to study the quality of test code throughout software evolution.

Studying code coverage evolution: We observed that some test failures do not involve any coverage change either in the failure introduction or the failure resolution in all studied projects in RQ3 and RQ4. Future studies may use our dataset to study the evolution of such test failures and their coverage throughout their entire lifetime. We observe that the time of resolving test failures and how the failures are resolved is project-specific and our study provides the first-step insights towards studying the relationship between test failure and code coverage. Future studies may investigate whether there exists any correlation between the increased coverage and faster failure resolution time. It is also interesting to investigate whether better coverage helps to detect and resolve the failures.

Studying the roles of test cases that failed repetitively: In RQ1, our results show that most test failures are concentrated in a small number of test cases. Namely, the test failure may be resolved but the same test case may fail again multiple times during the evolution due to other reasons. Our dataset may be used to study the characteristics of such test cases, and whether it is possible to help developers enhance the quality of both the test code and the tested source code to prevent future bugs.

Studying how code change history may assist fault localization and program repair techniques: In RQ3, our findings show that test execution may stop prematurely when it encounters a test failure, which might result in a decrease in code coverage. As there exist many automated testing techniques that leverage code coverage (i.e., fault localization and program repair techniques), future studies may use our dataset to propose new or enhance current techniques. Future studies may mine the past code coverage data to complement the decrease in coverage. In addition, future studies may analyze the code changes that we collected in the dataset and examine how recent code changes contribute to test failures. Future studies might even use our dataset for benchmarking automated testing techniques.

Studying how build configurations may affect test result: Our dataset contains all the build logs that we collected when compiling and executing the test cases in the studied commits. As we found in RQ4, some failure-resolving commits only modify non-code files such as configuration files. Future studies may analyze the build logs to study how the quality of the build scripts contributes to test failures.

6 Threats to Validity

External validity. Our studied projects are all open source and implemented in Java, so our findings may not be generalizable to other projects. To minimize the threat, we try to choose the projects that are well studied in the research community or are commonly used by many systems around the world (e.g., junit4). Future research may consider collect similar datasets for projects that are implemented in other programming languages and verify with our findings. We base our findings on the data in the studied time range from each project. In some cases, studying a different time range may lead to slightly different results. To generalize our results as much as possible, we select a large range of commits (1,000 commits per studied project). This time range of commits was chosen because it is a relatively large number but still feasible to manually resolve compilation issues. Note that for some projects (e.g., commons-cli), the total number of commits is less than 1,000 commits as of September 2020. In total, we compile and execute the test cases in 8,093 commits across the studied projects.

Construct validity. In our study, the starting date varies from 2002 to 2018, which implies that the studied projects may be at different stages of development. While this can increase the diversity of the studied systems, it can also be a threat to the construct validity of our results. Nevertheless, our findings are consistent in general. We encourage future research to leverage our dataset and further explore the differences among the projects and their relationship between different time ranges. When analyzing code coverage, we notice that some test cases may have empty coverage. To ensure the validity of our results, we re-run all the failing tests in a new environment. Then, if there is still any test failure without coverage, we randomly select some tests to run manually. Based on our manual study, we observe that the code coverage might not be generated when the test case fails in the early stage of the execution where no source code is yet covered. Future studies should consider this situation when applying automated testing techniques that leverage coverage information. Even though we tried our best to compile and run the test cases in the studied projects, some of the excluded commits (e.g., we cannot compile) may still be compilable. Nevertheless, our dataset still includes over 8,000 compiled commits and test execution results, which we share with the research community. There are uncompilable commits between sequences of compilable commits, which might affect the continuation of our dataset. In Appendix A.1, we conducted analyses and showed that, despite the presence of uncompilable commits in some projects, in general, our dataset contains long and consecutive sequences of compilable commits. We encourage future studies to further investigate those compilation issues.

There are many factors that can influence the compilation of the commits (e.g., availability of past dependencies, inappropriate Java Development Kit version). For instance, most of the studied projects use a handful of third-party libraries, so there may be other dependency issues that require manual fixes (e.g., some libraries are no longer available in the central Maven repository and need to be manually downloaded). We manually resolve the issues and update our automation scripts accordingly. We spent our best effort to manually resolve the compilation issues. To provide better confidence on the accuracy of our results and allow continuous improvement on our dataset, we made all the data that we collected publicly available [32], as transparent as possible with this paper. Due to the size of the code coverage data, we published it separately in a Zenodo repository [33].

7 Related Work

Testing in Continuous Integration. Some prior research [28, 25, 26] have aimed to study CI testing practices. Rogers [28] found that developers might allow known test failures into CI, as long as those failures are resolved by the end of the development iteration. Our study also found that, despite the presence of CI environment, there are a non-negligible number of test failures that persist over multiple days. Beller et al. [25] observed that testing constitutes the main reason why builds fail in CI, with test failures responsible for 59% of broken builds. Labuschagne et al. [26] showed that 18% of test executions in CI fail and that 13% of these test failures are flaky. In their study, they categorized the resolution of failed tests into three categories: code fixes, test fixes, and combination of code and test fixes. In our study, we find that non-code changes (e.g., data files) might also constitute the resolution of test failure.

Previous studies [29, 30] also discussed some characteristics of test failure by exploring how the test code evolved over time. Marsavina et al. [29] discussed co-evolution patterns of production and test code. Our study further examines how code coverage, production and test code change upon the introduction of a test failure. We find that, compared to the prior passing commit, the code coverage generally decreases. They also found that, in some studied projects, up to 47% of all code changes are performed on test files. In our manual study, our goal is to study how the code changes when developers resolve the test failures. We find that 48% of the code changes modified test files when resolving test failures (i.e., 34% modifying both test and source code files, 14% on only the test files). Pinto et al. [30] studied the effect of newly added tests on code coverage, and found that, on average, 56% of the newly added tests do not change the previous code coverage (i.e., branch coverage). In this paper, we observe that test failures might impact code coverage, since the test execution may stop prematurely which results in a decrease in code coverage.

Testing Practices. Prior studies conducted empirical studies on test code, and proposed suggestions to help improve testing practices [34, 31, 35, 36]. Just et al. [34] evaluated the developer-provided tests (from version history) and the user-provided tests (from bug reports) on fault localization and automated program repair techniques. They found that developer-provided tests contain more information to detect bugs, as the tests are specifically tailored to cover the buggy code. Kim et al. [31] conducted an empirical study on the evolution and maintenance of test annotations. They found that developers may use test annotations to remove or disable failed tests. Liu et al. [35] discussed the potential bias of over-fitting issue in automated program repair, where test failure may be resolved without actually fixing the bug. Our results confirm this finding as we found that developers may disable the assertion statements to make failed tests pass again. Hilton et al. [36] evaluated the coverage change between project revisions and assessed the impact of code changes on test quality. Zaidman et al. [37] studied the co-evolution of source and test code. They investigated the test coverage evolution based on its relation with test-writing activity. Beller et al. [27] suggested in their study that the production and test code have some tendency to change together, but the production code change does not always involve test change and vice versa. Catolino et al. [38] surveyed developers to understand how assertion density relates to the quality of test code. In this study, we present the test result and test coverage in the context of CI where the code changes are integrated and tested continuously.

Bug and Test Datasets. Many studies [9, 11, 12, 13, 39] have proposed benchmark or dataset on failed (and passed) tests to facilitate research on automated testing techniques. Just et al. [9] proposed Defects4J which records the information on the failed tests before and after the failure resolution. Le Goues et al. [11] proposed ManyBugs and IntroClass for C projects that have the test failure, the version in which it occurs, and the repairs to the failure that describes expected behavior. Elbaum et al. [12] collected a dataset at Google that includes over 3.5M records of test suite executions. Madeiral et al. [13] shared BEARS-BENCHMARK that contains the test failures before and after the resolution. Saha et al. [39] presented Bugs.jar that contains test failures for existing bugs. We present T-Evos as a dataset that contains the evolution of test execution. As the code changes are integrated and tested continuously, the evolution data provide additional values in CI settings. Our dataset may also complement existing datasets and benchmarks.

8 Conclusion

In this paper, we present a dataset, T-Evos, which contains fine-grained test execution information collected on a commit-by-commit basis. We compile and execute the test cases in 8,093 commits across 12 projects, and collected the build logs, test status, code coverage, code changes, and stack traces. The data collection process took several hundred hours and over 10 year CPU time, resulting in over 3TB in data size. We also conduct an empirical study on the collected test execution data. Our results also show that 1) 42% of commits contain test failures, and most of the test failures are resolved within one day; 2) many of the test failures happen in newly added or recently modified test files (e.g., test is modified to address an issue); 3) when resolving the test failures, developer may modify non-code files or only test files; 4) there is only an average of 66% overlap between the files modified in failure-resolving commits and files executed in the failing commit, as the coverage of failed test cases may be incomplete if the test case fails during the early execution stag. Finally, we provide some possible research directions that may be done using T-Evos. In summary, our paper presents a new dataset on the evolution of test failures and code coverage. We highlight the characteristics of test failure and its relationship with code changes. In addition, our dataset may be used in future testing research or benchmarking different automated testing techniques.

References

[1] J. A. Jones, M. J. Harrold, and J. Stasko, “Visualization of test information to assist fault localization,” in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE, 2002, pp. 467–477.

[2] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, “Spectrum-based multiple fault localization,” in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’09, 2009, pp. 88–99.

[3] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[4] C. M. Rosenberg and L. Moonen, “Spectrum-based log diagnosis,” in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–12.

[5] T.-D. B. Le, R. J. Oentaryo, and D. Lo, “Information retrieval and spectrum based bug localization: Better together,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 579–590.

[6] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, “An empirical analysis of the influence of fault space on search-based automated program repair,” *arXiv preprint arXiv:1707.05172*, 2017.

[7] J. Yi, S. H. Tan, S. Mechtaev, M. Böhme, and A. Roychoudhury, “A correlation study between automated program repair and test-suite metrics,” *Empirical Software Engineering*, vol. 23, no. 5, pp. 2948–2979, 2018.

[8] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “Genprog: A generic method for automatic software repair,” *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[9] R. Just, D. Jalali, and M. D. Ernst, “Defects4j: A database of existing faults to enable controlled testing studies for java programs,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[10] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, “Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge,” in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 2017, pp. 55–56.

[11] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, “The ManyBugs and IntroClass benchmarks for automated repair of C programs,” *IEEE Transactions on Software Engineering (TSE)*, vol. 41, no. 12, pp. 1236–1256, December 2015.

[12] S. Elbaum, G. Rothermel, and J. Penix, “Techniques for improving regression testing in continuous integration development environments,” in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 235–245.

[13] F. Madeiral, S. Urli, M. de Almeida Maia, and M. Monperrus, “BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies,” in *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*, ser. SANER’19, 2019, pp. 468–478.

[14] Z. Peng, T. Chen, and J. Yang, “Revisiting test impact analysis in continuous testing from the perspective of code dependencies,” *IEEE Transactions on Software Engineering*, no. 01, pp. 1–1, dec 2021.

[15] “Maven surefire report plugin,” https://maven.apache.org/surefire/maven-surefire-report-plugin/, 2021, last accessed May 5 2021.

[16] L. S. Pinto, S. Sinha, and A. Orso, “Understanding myths and realities of test-suite evolution,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE ’12, 2012, pp. 33:1–33:11.

[17] D. J. Kim, B. Yang, J. Yang, and T.-H. P. Chen, “How disabled tests manifest in test maintainability challenges?” in *Proceedings of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE ’21, 2021, pp. 187–196.

[18] “Jacoco,” https://www.eclemma.org/jacoco/, 2021, last accessed May 5 2021.

[19] F. Palomba and A. Zaidman, “Does refactoring of test smells induce fixing flaky tests?” in *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME 2017, 2017, pp. 1–12.

[20] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, “An empirical analysis of flaky tests,” in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 643–653.

[21] P. Zhang, Y. Jiang, A. Wei, V. Stodden, D. Marinov, and A. Shi, “Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications.”

[22] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, “A study on the lifecycle of flaky tests,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE ’20, 2020, p. 1471–1482.

[23] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, “Deflaker: Automatically detecting flaky tests,” in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 433–444.

[24] Jira, “Jira rest apis,” 2020, last accessed: Feb. 1, 2020. [Online]. Available: https://developer.atlassian.com/server/jira/platform/rest-apis/

[25] M. Beller, G. Gousios, and A. Zaidman, “Oops, my tests broke the build: An explorative analysis of travis ci with github,” in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 356–367.

[26] A. Labuschagne, L. Inozemtseva, and R. Holmes, “Measuring the cost of regression testing in practice: A study of java projects using continuous integration,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 821–830.

[27] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, “When, how, and why developers (do not) test in their ides,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 179–190.

[28] R. O. Rogers, “Scaling continuous integration,” in *International conference on extreme programming and agile processes in software engineering*. Springer, 2004, pp. 68–76.

[29] C. Marsavina, D. Romano, and A. Zaidman, “Studying fine-grained co-evolution patterns of production and test code,” in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014, pp. 195–204.

[30] L. S. Pinto, S. Sinha, and A. Orso, “Understanding myths and realities of test-suite evolution,” in *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering*, 2012, pp. 1–11.

[31] D. J. Kim, N. Tsantalis, T.-H. P. Chen, and J. Yang, “Studying test annotation maintenance in the wild,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 62–73.

[32] “T-evos,” https://github.com/T-Evos/T-Evos, 2021, last accessed September 1 2021.

[33] “T-evos,” https://zenodo.org/record/5500821#.Yo9D6RNBw-Q, 2021, last accessed September 1 2021.

[34] R. Just, C. Parnin, I. Drosos, and M. D. Ernst, “Comparing developer-provided to user-provided tests for fault localization and automated program repair,” in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 287–297.

[35] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyandé, “A critical review on the evaluation of automated program repair systems,” *Journal of Systems and Software*, vol. 171, p. 110817, 2021.

[36] M. Hilton, J. Bell, and D. Marinov, “A large-scale study of test coverage evolution,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 53–63.

[37] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, “Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining,” *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, 2011.

[38] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, “How the experience of development teams relates to assertion density of test classes,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 223–234.

[39] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, “Bugs.jar: a large-scale, diverse dataset of real-world java bugs,” in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 10–13.

An Ran Chen An Ran Chen is a Ph.D. candidate in Software Engineering at Concordia University, Montreal, Canada. He received his Bachelor and Master degree in Software Engineering from Concordia University. He is a research assistant in the Software PErformance, Analysis, and Reliability (SPEAR) Lab. His research interests include bug localization, log analysis, mining software repositories, software testing and debugging. Prior to joining SPEAR, he also worked as a software engineer at Bank of Canada and McGill University IT Services. More information at: https://anrchen.github.io/home.

Tse-Hsun (Peter) Chen Tse-Hsun (Peter) Chen is an Associate Professor in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. He leads the Software PErformance, Analysis, and Reliability (SPEAR) Lab, which focuses on conducting research on performance engineering, program analysis, log analysis, production debugging, and mining software repositories. His work has been published in flagship conferences and journals such as ICSE, FSE, TSE, EMSE, and MSR. He serves regularly as a program committee member of international software engineering conferences, such as FSE, ASE, ICSME, SANER, and ICPC. He is also a regular reviewer for software engineering journals such as JSS, EMSE, and TSE, and serves as a guest editor for special issues in EMSE. Dr. Chen obtained his BSc from the University of British Columbia, and MSc and PhD from Queen's University. Besides his academic career, Dr. Chen also worked as a software performance engineer at BlackBerry for over four years. Early tools developed by Dr. Chen were integrated into industrial practice for ensuring the quality of large-scale enterprise systems. More information at: http://petertsehsun.github.io/.

Shaowei Wang Shaowei Wang is an assistant professor in the Department of Computer Science at the University of Manitoba. He leads the Software Management, Maintenance, and Reliability Lab (Mamba). His research interests include software engineering, machine learning, data analytics for software engineering, automated debugging, and secure software development. His work has been published in flagship conferences and journals such as FSE, ASE, TSE, TOSEM, and EMSE. He serves regularly as a program committee member of international conferences in the field of software engineering, such as ICSE, ASE, ICSME, SANER, ICPC, and MSR, and he is a regular reviewer for software engineering journals such as TOSEM, JSS, EMSE, and TSE. He is one of four recipients of the 2018 distinguished reviewer award for the Springer EMSE (SE's highest impact journal). He obtained his Ph.D. from Singapore Management University and his BSc from Zhejiang University. More information at: https://sites.google.com/view/mambalab.