BUILD-SHERIFF: Change-Aware Test Failure Triage for Continuous Integration Builds

Chen Zhang†
School of Computer Science
Fudan University
Shanghai, China

Xin Peng∗
School of Computer Science
Fudan University
Shanghai, China

Bihuan Chen††
School of Computer Science
Fudan University
Shanghai, China

Wenyun Zhao∗
School of Computer Science
Fudan University
Shanghai, China

ABSTRACT

Test failures are one of the most common reasons for broken builds in continuous integration. It is expensive to diagnose all test failures in a build. As test failures are usually caused by a few underlying faults, triaging test failures with respect to their underlying root causes can save test failure diagnosis cost. Existing failure triage methods are mostly developed for triaging crash or bug reports, and hence not applicable in the context of test failure triage in continuous integration. In this paper, we first present a large-scale empirical study on 163,371 broken builds caused by test failures to characterize test failures in real-world Java projects. Then, motivated by our study, we propose a new change-aware approach, BUILD-SHERIFF, to triage test failures in each continuous integration build such that test failures with the same root cause are put in the same cluster. Our evaluation on 200 broken builds has demonstrated that BUILD-SHERIFF can significantly improve the state-of-the-art methods on the triaging effectiveness.

CCS CONCEPTS
• Software and its engineering → Software testing and debugging

KEYWORDS
Test Failures, Failure Triage, Continuous Integration

1 INTRODUCTION

Continuous integration (CI) [20] has gained widespread use and continued growth [21, 50] as it helps detect integration errors earlier, enhance developer productivity, and reduce development risk [31, 66]. Each integration is verified by an automated build that includes dependency installation, code compilation, static analysis, and test case execution. However, CI builds often break (i.e., fail), and developers need to spend much effort in troubleshooting broken builds [30]. As evidenced by several recent studies with open-source and industrial projects [8, 34, 37, 55, 67], test failures are one of the most frequent types of CI build failures; e.g., test failures are responsible for 59.0% of broken builds in open-source Java projects [8]. Developers need to manually localize and repair the underlying faults of test failures in each build. However, it is non-trivial for developers to analyze test failures in a build because a build is not always triggered for every commit, and a build may change multiple source code files. In that sense, it is usually time-consuming and expensive to manually diagnose test failures in each build.

To reduce test failure diagnosis cost, a number of techniques have been proposed from different perspectives. One line of work tries to design fault localization techniques to automatically localize faults that cause test failures [27, 53, 70, 74]. Another line of work tries to develop program repair techniques to automatically fix faults [48]. Orthogonal to these two types of techniques, test failure triage techniques are designed to cluster test failures that are caused by the same fault into the same cluster [25, 26]. In this way, test failure diagnosis can be realized by only analyzing one test failure in each cluster but not all the test failures, which can reduce manual diagnosis cost or boost automated fault localization and program repair techniques.

Existing failure triage methods are mostly designed to triage crash or bug reports. Sharing the same problem as test failures in CI builds, multiple reports can be filed for the same fault. These methods can be grouped into stack trace-based (e.g., [10, 15, 17, 43]), profiling-based (e.g., [14, 22, 44, 52]), and text-based methods (e.g., [3, 57, 61, 62]), depending on what information (i.e., stack traces, execution profiles, and textual descriptions) is used to measure failure similarity.

Unfortunately, stack trace-based methods are not applicable in the context of test failure triage in CI because they are not designed for test failures in CI, and are not aware of the test failure characteristics and potentially valuable knowledge in CI for better triage. First, we show in our study (Sec. 2.1) that 73.2% of broken builds with test failures are caused
by assertion failures, which do not report any informative exception stack trace. As a result, stack trace-based methods could have a poor performance on triaging assertion failures. Second, we also report in our study (Sec. 2.1) that test failures in 78.5% of the 200 randomly selected broken builds share one root cause. Hence, stack trace-based methods could tend to triage test failures into one single cluster such that they could still yield an overall good performance although they have a poor performance on triaging test failures into multiple clusters. Third, code changes in CI builds, a valuable knowledge in CI but not available in crash or bug reports, usually have a great impact on test results [46]. Thus, stack trace-based methods, simply relying on stack traces without taking into account code changes, could be less effective. Profiling-based and text-based methods are also not applicable to test failure triage in CI because execution profiles would impose too large overhead to be practically used in CI [46] while textual descriptions (which are often written by users in crash or bug reports) of test failures are not produced during automated CI builds.

In this paper, we first present a large-scale empirical study, using 163,371 broken builds caused by test failures from 1,337 GitHub Java projects, to understand test failures in CI builds. We characterize the prevalence of two types of test failures (i.e., exception failures and assertion failures), and motivate the potential value and the design insights of test failure triage. Then, we propose a new change-aware approach, BuildSheriff, to triage test failures for CI builds so that test failures with the same root cause are put in the same cluster. The key idea of BuildSheriff is to consider code changes as important triage knowledge as root causes of test failures are often introduced by code changes [75]. We develop a pipeline of three strategies based on complexity of code changes, change-aware stack trace similarity, and exception message similarity for exception failure triage, and a pipeline of two strategies based on complexity of code changes and change-aware test code similarity for assertion failure triage. To the best of our knowledge, this is the first work to triage test failures in general CI.

To evaluate the effectiveness and efficiency of BuildSheriff, we compared it with one naive method and three state-of-the-art methods [12, 15, 17] on 200 broken builds with respect to 20 metrics. Our evaluation results have demonstrated that i) BuildSheriff can significantly improve the naive method on 13 of the 20 metrics (e.g., by 14%+ on the number of correctly triaged builds, and 77%+ on the number of missed root causes); ii) BuildSheriff can significantly improve the best of the state-of-the-art methods on 15 of the 20 metrics (e.g., by 45%+ on the number of correctly triaged builds, and 28%+ on test failure inspection effort saving) and slightly improve the best of the state-of-the-art methods on 4 of the 20 metrics; and iii) the average time overhead to triage a build is 1.61 seconds.

In summary, this paper makes the following contributions.

- We conducted a large-scale empirical study to characterize test failures in real-world Java projects and motivate test failure triage.
- We proposed a new change-aware approach, BuildSheriff, to triage test failures in CI builds effectively and practically.
- We conducted experiments on 200 broken builds to demonstrate the effectiveness and efficiency of BuildSheriff.

2 MOTIVATION

In this section, we first present an empirical study of test failures, and then introduce motivating examples of test failures.

2.1 An Empirical Study of Test Failures

Data Set. To construct the data set for our empirical study of test failures, we start with the data set proposed by Zhang et al. [73], which includes the CI build history of 3,799 Java projects on GitHub. To the best of our knowledge, it is the largest available data set of CI builds. To ease the extraction of test failure information from build logs, we focus on projects that use Maven as the automated build tool, which results in 1,763 projects. To further ensure that CI is commonly used, we exclude the projects that only have less than 100 builds, which restricts our selection to 1,739 projects with a total of 3,981,842 builds. Of these builds, 833,209 (20.9%) builds have a build state of errored or failed, which are also known as broken builds. In particular, 163,371 (19.6%) of these broken builds, covering 1,337 projects, are caused by test failures, which is lower than the 59.0% as reported by Beller et al. [8] (using 423 Java projects). The difference may be due to the different scale of studies. Still, test failure is a non-negligible failure type of broken builds, and affects many of the Java projects. Notice that a test is considered as failed in CI if it throws an exception (i.e., exception failure) or fails an assertion (i.e., assertion failure), and the build log from Maven projects also provides a summary of the number of tests that signals exception failures and assertion failures.

Research Questions. Using 163,371 broken builds with test failures (hereafter referred to as test-failed builds) from 1,337 projects, our study is designed to answer the following research questions.

RQ1: How many test-failed builds in a project exhibit the symptom of exception and assertion failures during test execution?

RQ2: How many tests in a test-failed build signal exception and assertion failures during test execution?

RQ3: How many root causes affect failed tests in a test-failed build?

In RQ1, we measure for each project the ratio of test-failed builds that exhibit the symptom of exception and assertion failures during test execution, and compute the ratio distribution across all projects. Our results from RQ1 aim to characterize the prevalence of both exception and assertion failures, and motivate the need for a triage approach to support both of them. In RQ2, we measure for each test-failed build the number of all tests and the number of tests that signal exception and assertion failures, and compute their distribution across all test-failed builds. In RQ3, we randomly select 100 test-failed builds with exception failures and 100 test-failed builds with assertion failures, manually triage the failed tests via locating their root causes, and report the distribution of the number of root causes. Our results from RQ2 and RQ3 aim to motivate the potential value of triaging test failures in CI, and demonstrate the specific characteristics of test failures in CI to better design a triage approach.

Test-Failed Build Analysis (RQ1). Overall, 80,778 (49.4%) of the 163,371 test-failed builds, covering 1,154 projects, exhibit the symptom of exception failures, and 119,514 (73.2%), covering 1,246 projects, exhibit the symptom of assertion failures. In detail, Fig. 1a shows the distribution of the ratio of test-failed builds that have exception failures and assertion failures across all projects in violin plot. The three lines in each plot respectively denote the upper quartile, the median and the lower quartile. We can see that at least 24.1%, 58.8% and 85.7% of the test-failed builds contain exception failures in 75%, 50% and 25% of the projects, whereas more than 55.8%, 81.1% and 100% of the test-failed builds have assertion failures in 75%, 50% and 25% of the projects. These results indicate that assertion failures are even more common
than exception failures to cause test-failed builds. Therefore, test failure triage in CI should support both exception and assertion failures, and existing stack trace-based failure triage techniques, which mainly support exception failures, are not directly applicable in CI.

Failed Test Analysis (RQ2). Fig 1b reports the distribution of the number of all tests (in the left violin plot) and the number of failed tests (in the right violin plot) across the 80,778 test-failed builds that have exception failures and the 119,514 test-failed builds that have assertion failures in logarithmic scale. We can observe that in 75%, 50% and 25% of the test-failed builds that have exception failures, there are at least 50, 403 and 1,636 tests, while at least 1, 3 and 9 tests signal exception failures; and in 75%, 50% and 25% of the test-failed builds that have assertion failures, there are at least 132, 536 and 2,000 tests, while at least 1, 2 and 4 tests signal assertion failures. Compared to the number of all tests, the number of failed tests is relatively very small. In particular, 52,586 of the 80,778 test-failed builds have at least two tests signal exception failures, and their median number of tests signaling exception failures is 6, while 62,494 of the 119,514 test-failed builds have at least two tests signal assertion failures, and their median number of tests signaling assertion failures is 4. These results demonstrate that multiple test failures are moderately common in CI builds, which represents the potential reduction of test failure diagnosis cost that can be achieved by test failure triage.

Root Cause Analysis (RQ3). We randomly pick 100 test-failed builds that have at least two tests signal exception failures and 100 test-failed builds that have at least two tests signal assertion failures, achieving a confidence level of 95% and a margin of error of 9.8%. It is worth mentioning that over 30% of these 200 broken builds were triggered after more than two commits, and on average, around nine source code files were changed in each of the 200 broken build. Thus, it is non-trivial for developers to diagnose test failures in a build. Two of the authors separately diagnose failed tests in these 200 test-failed builds by investigating the previous commits that may introduce the failure, the succeeding commits that may fix the failure and the build log in order to identify the root cause for each failed test. We define a root cause as the code changes that introduce the failure. Then, they investigate inconsistent cases together to reach consensus. We spent around 1.5 person-month to complete the manual analysis. Fig. 1c reports the manual triage results, i.e., the distribution of the number of root causes across the 200 test-failed builds. We can see that the failed tests with exception failures have the same root cause in 82 builds, and have multiple root causes in 18 builds. The failed tests with assertion failures have the same root cause in 75 builds, and have multiple root causes in 25 builds. These results indicate that test failures in a large part of the test-failed builds have only one root cause. Thus, test failure triage in CI should be aware of this specific characteristic. Existing stack trace-based failure triage techniques fail to do so, and their triage model tends to triage test failures into one root cause such that it can still yield an overall good performance although it has a poor performance on triaging test failures with multiple root causes.

2.2 Motivating Examples of Test Failures

Fig. 2 shows the exception log of three of the 62 exception failures in a build of the project bioformats. The first line of the log includes the signature of the test that signals the exception failure. The second line shows the exception message that is composed of two parts, i.e., the exception type and a developer-written message. The rest of the lines list the exception stack trace. Fig. 3 shows the code changes that cause the exception failures. The call to mciis.close() was inserted after reader.dispose() in method readImage in Fig. 3a, causing the failure in Fig. 2a. Similarly, the call to mciis.close() was inserted after reader.Dispose() in method readRaster in Fig. 3b, which caused the failures in Fig. 2b and 2c. In fact, the code change in Fig. 3a caused six of the 62 exception failures, and the code change in
3 APPROACH

In this section, we first present an overview of BUILDERSHERIFF, and then introduce each step of BUILDERSHERIFF in detail.

3.1 Overview

Fig. 7 presents the overview of our change-aware test failure triage approach. BUILDERSHERIFF is designed to triage failed tests in each CI build of a target project so that the failed tests with the same root cause are put in the same cluster. BUILDERSHERIFF is triggered when a CI build from a project repository occurs. It works in three steps: triage knowledge preparation (Sec. 3.2), exception failure triage (Sec. 3.3), and assertion failure triage (Sec. 3.4). It first prepares triage knowledge by analyzing the build log, the project source code, and the code changes from the previous passed build. Based on the triage knowledge, it then uses different strategies to triage exception failures and assertion failures. The key characteristic of our triage strategies is the awareness of code changes. BUILDERSHERIFF is currently implemented for Java projects that use Travis as CI service and Maven as build tool.

3.2 Triage Knowledge Preparation

BUILDERSHERIFF has three steps to prepare the triage knowledge for exception failures and assertion failures. The triage knowledge is three-fold: test failures (i.e., test signature, exception message, stack trace, and test code), code changes at the file, method and field level that potentially cause the test failures, and file dependencies that capture the code change impacts that also potentially cause the test failures.

Test Failure Analysis. As the build log contains a well-formatted failure analysis, the key characteristic of our triage strategies is the awareness of code changes. BUILDERSHERIFF is currently implemented for Java projects that use Travis as CI service and Maven as build tool.
extract two sets of failed tests that respectively signal exception failures and assertion failures, and parse the test signature \( \text{sig} \) (i.e., class name and test method name) and the line number of the test code that signals the failure for each of the failed tests. If no test failure occurs (i.e., the build is not a test-failed build), BuildSheriff stops.

Then, for a failed test signaling an exception failure, we use its test signature to locate and parse the exception message \( \text{msg} \) and the stack trace \( \text{st} \) from the build log. The exception message contains the exception type and a developer-written message, while the stack trace consists of an ordered list of frames (i.e., methods) that were active on the call stack before the exception occurred. Generally, if two exception failures share the same root cause, they are more likely to be similar in the exception message and the stack trace. Differently, for a failed test that signals an assertion failure, we do not extract the assertion message and the stack trace because of their low discrimination. Specifically, the assertion messages in the build log mostly contain the same error type (e.g., `java.lang.AssertionError`), while the stack traces mostly consist of similar methods that do not belong to the target project but are assertion-related methods from the testing infrastructure (e.g., JUnit) as the methods before the assertion statement in the tests successfully returned. In that sense, assertion failures that have different root causes could have similar assertion messages and stack traces. For example, all the frames except for the first frame in the two stack traces in Fig. 4 are methods from Java reflection, JUnit and Maven, and are exactly the same, although the two assertion failures have different root causes.

Finally, for a failed test that signals an assertion failure, we obtain the test method from the project source code according to its test signature, and tailor the test code \( tc \) from the first line of the test method to the line number of the test code that signals the assertion failure. In other words, only the executed test code is included in \( tc \) because the unexecuted test code is actually not related to the test failure. However, for a failed test signaling an exception failure, we do not tailor the test code as the triage knowledge. The reason is that tests signaling exception failures, compared to tests signaling assertion failures, are mostly partially executed, and hence the executed test code contains less informative knowledge for exception failure triage.

Formally, we define an exception failure \( x_e \) in the exception failures \( X_e \) of a build as a 3-tuple \((\text{sig}, \text{msg}, \text{st})\), and define an assertion failure \( x_a \) in the assertion failures \( X_a \) of a build as a 2-tuple \((\text{sig}, \text{tc})\).

**Code Change Analysis.** Motivated by the fact that root causes of test failures are often introduced by code changes [75], we consider code changes as important triage knowledge. Instead of considering code changes from the previous build, we consider code changes from the previous passed build because all code changes in previous consecutive failed builds can potentially cause the test failures in the current build. To this end, we use the code differencing tool ClDiff [33] to obtain the changed source code files \( F_c \), methods \( M_c \) and fields \( D_c \) from the previous passed build to the current build.

**File Dependency Analysis.** Code affected by the code changes can also potentially cause test failures. Thus, we use code dependencies to capture such code change impacts and sever as the triage knowledge. We model code dependencies at the file-level rather than method-level granularity for two reasons. On one hand, we are inspired by recent studies on practical regression test selection [24, 42], which show that better results can be achieved by selecting tests at a coarser (file-level) compared to a finer (method-level) granularity of code dependencies. On the other hand, it is more efficient to analyze file dependencies than method dependencies, which is important for practical failure triage in CI. Specifically, we adopt an incremental way to analyze file dependencies. We first construct a file dependency graph \( \mathcal{G} = (V, E) \) for the target project when the first test-failed build occurs. Each node in \( V \) denotes a source code file, and each edge in \( E \) denotes a usage dependency between two files. A file can use all files from its own package, or files from other packages via import statement or fully qualified name. Thus, we use JDT to parse each source code file, i.e., to parse its own package and import statements to identify potentially used files and establish usage dependencies. Here we do not analyze file usages with fully qualified name due to its heavy-weight analysis cost and its uncommon adoption. Note that this step is a one-time job for the target project. Then, we update \( \mathcal{G} \) for each subsequent test-failed build by parsing changed (i.e., deleted, added, and modified) files from the previous test-failed build.

### 3.3 Exception Failure Triage

BuildSheriff achieves the triage of exception failures \( X_e \) in a build by a pipeline of the following three strategies.

**\( S_1^e \): Complexity of Code Changes.** Generally, the higher complexity of code changes in a build, the higher chance of causing test failures, and the larger number of root causes for test failures. However, CI requires developers to frequently merge their code to find errors as early as possible. Therefore, code changes in a build are mostly not complex, and as revealed by our study (Sec. 2.1), a large part of test-failed builds have only one root cause for test failures. Motivated by these observations, we propose a strategy \( S_1^e \) to use the complexity of code changes to determine whether \( X_e \) has one root cause or not. Here, we use the number of changed methods (i.e., \( |M_c| \)) as the indicator of code change complexity, and consider \( X_e \) as having one root cause if \( |M_c| \) is lower than a threshold \( \Delta_e \) (i.e., \( |M_c| \leq \Delta_e \)). Thus, if \( |M_c| \leq \Delta_e \) is satisfied, BuildSheriff triages all exception failures into one cluster; otherwise, it uses the next strategy in the pipeline.

**\( S_2^e \): Change-Aware Stack Trace Similarity.** A stack trace is an ordered list of frames on the call stack when an exception occurred. Each frame represents a method, and records its enclosing file name. A stack trace carries important information for debugging [59]. We propose a new change-aware stack trace similarity metric based on the insight that a higher weight should be given to the frames that are closer to code changes because such frames are more likely to be affected by code changes and be blamed for the exception. Based on this metric, we propose a clustering strategy \( S_2^e \) to triage \( X_e \).
Based on the above insight, we first define the distance of a frame $m$ to code changes, as formulated in Eq. 1, where $f_m$ denotes the enclosing file of $m$, and $dt(f_m, f_c, G)$ denotes the distance of $f_m$ to a changed file $f_c \in F_c$ on the file dependency graph $G$. Basically, $dt(m)$ is zero if $m$ itself is changed, and one if the enclosing file of $m$ is changed; otherwise, $dt(m)$ is measured by the minimum distance of $f_m$ to all changed files on $G$ if $f_m$ reaches some changed file, and a large value $mDt_e$ if $f_m$ does not reach any changed file.

$$dt(m) = \begin{cases} 
0, & m \in M_1 \\
1, & f_m \in F_c \\
1 + \min_{f_c \in F_c} dt(f_m, f_c, G) & f_m \text{ reach some } f_c \\
mDt_e & \text{otherwise} 
\end{cases} \quad (1)$$

Then, we define the weight (or importance) of a frame $m$ based on the distance to code changes $dt(m)$, as formulated in Eq. 2, where $e_c$ is a coefficient for the distance to code changes. The smaller the distance to code changes, the higher the weight.

$$w(m) = e^{-e_c dt(m)} \quad (2)$$

Then, we define the similarity of two stack traces $s_1 = \{m_1, m_2, \ldots\}$ and $s_2 = \{m_1', m_2', \ldots\}$ using the Sørensen-Dice index [18] (i.e., Eq. 3).

$$\text{sim}(s_1, s_2) = \frac{2 \cdot \sum_{m \in s_1 \cap s_2} w(m)}{\sum_{m \in s_1} w(m) + \sum_{m \in s_2} w(m)} \quad (3)$$

Finally, we apply the single-linkage agglomerative clustering technique [16] to cluster exception failures $X_e$ based on the similarity of the stack traces of exception failures. In other words, we use the maximum stack trace similarity of all exception failure pairs between two clusters as the cluster distance, as formulated in Eq. 4, where $X_e^1$ and $X_e^2$ are two clusters of exception failures; and we use a cluster distance threshold $dt_{cl}$ as a stopping criterion for clustering.

$$dt(X_e^1, X_e^2) = \max_{x_e^1 \in X_e^1, x_e^2 \in X_e^2} \text{sim}(x_e^1, s, x_e^2, s) \quad (4)$$

When the stack trace size (i.e., the number of frames) of an exception failure is small, the stack trace contains less informative knowledge for failure triage. Hence, we use $S^2_e$ if the maximum stack trace size of the exception failures is larger than a size threshold $\text{size}_{\text{cl}}$ (i.e., $\max_{e \in X_e} |x_e| \geq \text{size}_{\text{cl}}$); otherwise, we use the next strategy.

$S^2_e$: Exception Message Similarity. An exception message contains the exception type and a developer-written message, which potentially describes the root cause, symptom, or handling hint of the exception. Intuitively, two exception failures having the same root cause are likely to have similar exception messages. Hence, we design a clustering strategy $S^2_e$ based on exception message similarity to triage $X_e$.

First, we define the similarity of two exception messages $msg_1$ and $msg_2$ based on their Levenshtein distance at the token level, as formulated in Eq. 5, where $t(msg_1)$ denotes a sequence of tokens in $msg_1$ split by white space, and $dt(t(msg_1), t(msg_2))$ denotes the Levenshtein distance, i.e., the minimum number of token-level edits (i.e., insertion, deletion and substitution) to change $t(msg_1)$ into $t(msg_2)$.

$$\text{sim}(msg_1, msg_2) = \frac{\text{maxLength} - dt(t(msg_1), t(msg_2))}{\text{maxLength}} \quad (5)$$

We then apply the single-linkage agglomerative clustering technique to cluster exception failures $X_e$ based on the similarity of the exception messages of exception failures (i.e., Eq. 6). We use a cluster distance threshold $dt_{msg}$ as a stopping criterion for clustering.

$$dt(X_e^1, X_e^2) = \max_{x_e^1 \in X_e^1, x_e^2 \in X_e^2} \text{sim}(x_e^1, msg_1^2, x_e^2, msg_2) \quad (6)$$

Example. Given exception failures in Fig. 2, BuildSheriff adopts $S^2_e$ to achieve the triage. The distance of each frame to code changes is highlighted in the left of each frame in Fig. 2. As explained in Sec. 2.2, the exception failures in Fig. 2a and 2b had different root causes, even though their stack traces look similar. On the other way around, the exception failures in Fig. 2b and 2c had the same root cause, even though their stack traces look dissimilar. BuildSheriff correctly triages them due to our consideration of the distance of each frame to code changes in computing stack trace similarity. In detail, BuildSheriff gives the highest weight to the frame of $\text{readImage}$ and $\text{readRaster}$ because they are changed (as shown in Fig. 3). With such weighted similarity measure, BuildSheriff successfully triages the failures in Fig. 2a and 2b into two clusters, as the stack trace in Fig. 2a contains $\text{readImage}$ but not $\text{readRaster}$ and the stack trace in Fig. 2b contains $\text{readRaster}$ but not $\text{readImage}$; and it also correctly triages the failures in Fig. 2b and 2c into the same cluster, as both stack traces in Fig. 2b and 2c contain $\text{readRaster}$. The state-of-the-art method [15] fails on these cases.

3.4 Assertion Failure Triage

BuildSheriff achieves the triage of assertion failures $X_a$ in a build by a pipeline of the following two strategies.

$S^2_a$: Complexity of Code Changes. Similar to exception failure triage, we propose a strategy $S^2_a$ to use the complexity of code changes to determine whether $X_a$ has one root cause or not. We regard $X_a$ as having one root cause if the number of changed methods (i.e., $|M_c|$) is lower than a threshold $\Delta_a$ (i.e., $|M_c| \leq \Delta_a$). Thus, if $|M_c| \leq \Delta_a$ is satisfied, BuildSheriff triages all the failed tests into one cluster; otherwise, it uses the next strategy in the pipeline.

$S^2_a$: Change-Aware Test Code Similarity. The test code $tc$ of an assertion failure is the executed code when the assertion failure occurred. Intuitively, a higher test code similarity between two assertion failures, a higher likelihood that they share the same root cause. Similar to our change-aware stack trace similarity metric, we design a change-aware test code similarity metric based on the insight that a higher weight should be assigned to code tokens that are closer to code changes as such code tokens are more likely to be affected by code changes and be blamed for the assertion failure. Using this metric, we develop a clustering strategy $S^2_a$ to triage $X_a$.

Based on the above insight, we first tokenize test code $tc$ into a list of code tokens $\{t_1, t_2, \ldots\}$. Then, we define the distance of a code token $t$ to code changes, as formulated in Eq. 7, where $f_t$ denotes the enclosing file of $t$ if $t$ corresponds to the name of a method or a field, and $dt'(f_t, f_c, G)$ denotes the distance of $f_t$ to a changed file $f_c \in F_c$ on the file dependency graph $G$. Basically, $dt(t)$ is zero if $t$ is reached and corresponds to the name of a method or a field, and one if $f_t$ is changed; otherwise, $dt(t)$ is measured by the minimum distance of $f_t$ to all changed files on $G$ if $f_t$ reaches some changed file, and a large value $mDt_a$ if $f_t$ does not reach any changed file.

$$dt(t) = \begin{cases} 
0, & t \in M_c \cup D_c \\
1, & f_t \in F_c \\
1 + \min_{f_c \in F_c} dt'(f_t, f_c, G) & f_t \text{ reach some } f_c \\
mDt_a & \text{otherwise} 
\end{cases} \quad (7)$$
Then, we define the weight (or importance) of a code token \( t \) based on the distance to code changes \( dt(t) \), as formulated in Eq. 8, where \( c_a \) is a coefficient for the distance to code changes.

\[
w(t) = e^{-c_a dt(t)}
\]  

(8)

Then, we define the similarity of two test code \( tc_1 = \{l_1, l_2, \ldots \} \) and \( tc_2 = \{l_2, l_3, \ldots \} \) using the Sørensen-Dice index [18] (i.e., Eq. 9).

\[
sim(tc_1, tc_2) = \frac{2 \times \sum_{l \in tc_1 \cap tc_2} w(l)}{\sum_{l \in tc_1} w(l) + \sum_{l \in tc_2} w(l)}
\]  

(9)

We finally use the single-linkage agglomerative clustering technique to cluster assertion failures \( X_a \) based on the similarity of the test code of exception failures (i.e., Eq. 10). We use a cluster distance threshold \( dt_{tc} \) as a stopping criterion for clustering.

\[
dt(X_a^1, X_a^2) = \max_{x_a^1 \in X_a^1, x_a^2 \in X_a^2} \text{sim}(x_a^1, tc, x_a^2, tc)
\]  

(10)

**Example.** Given assertion failures in Fig. 4, BuildSheriff adopts \( X_a^1 \) to realize the triage. The code tokens of the test in Fig. 5a are "try", "new", "Retrofit", "Builder", "callbackExecutor", "null", "fail", "catch", "NullPointerExceptionException", "e", "assertThat", "e", "hasMessage" and "callbackExecutor == null", "Retrofit", "Builder" and "callbackExecutor" respectively have a distance of 1, 1 and 0 to code changes, and other code tokens have a distance of \( mDt_a \) to code changes. Similarly, "Retrofit", "Builder" and "addConverterFactory" in Fig. 5b respectively have a distance of 1, 1 and 0 to code changes. As introduced in Sec. 2.2, the assertion failures in Fig. 4 had different root causes, even though their failed tests in Fig. 5 look similar. BuildSheriff successfully triages them because of our consideration of the distance of each test code token to code changes in computing test code similarity. Specifically, BuildSheriff gives the highest weight to "callbackExecutor" and "addConverterFactory" because the corresponding methods are changed (as shown in Fig. 6). As the test code token with the highest weight is different in the two tests, BuildSheriff correctly triages the two failures into two clusters with weighted similarity measure.

### 3.5 Usage Scenario of Triage Results

After triaging test failures, BuildSheriff presents developers with a set of clusters, and notifies developers that each cluster has a set of test failures whose root cause is considered as the same. For each test failure in a cluster, BuildSheriff provides developers with the distance information to code changes (e.g., Fig. 2), and tells developers that methods on stack traces or methods in test code that have small distances to code changes (e.g., Fig. 2), and tells developers that

The differences between the two usage scenarios are that more test failures are inspected by developers in Usage Scenario 1 than in Usage Scenario 2, but more executions of all the tests are needed in Usage Scenario 2 than in Usage Scenario 1 (where at most two executions of all the tests are needed). In fact, the lower bound on the number of inspected test failures is achieved in Usage Scenario 2 at the cost of more executions of all the tests. In other words, the two usage scenarios have a different tradeoff between the number of inspected test failures and the number of executions of all the tests. Practical usage scenarios can be in-between Usage Scenario 1 and Usage Scenario 2. In that sense, the significance of BuildSheriff is that it reduces test failure diagnosis cost for CI developers by reducing the number of inspected test failures or the number of executions of all the tests through effectively triaging test failures.

## 4 EVALUATION

We have implemented BuildSheriff in 17.0K lines of Java code, and have released the code and data of BuildSheriff at our website [1].

### 4.1 Evaluation Setup

To evaluate the effectiveness and efficiency of BuildSheriff, we designed our evaluation to answer the following research questions.

- **RQ4**: How is the effectiveness of BuildSheriff in triaging test failures, compared with the state-of-the-art approaches? (Sec. 4.2)
- **RQ5**: How is the efficiency of BuildSheriff in triaging test failures, compared with the state-of-the-art approaches? (Sec. 4.3)
- **RQ6**: How is the contribution of each triage strategy in BuildSheriff to the achieved effectiveness of BuildSheriff? (Sec. 4.4)
- **RQ7**: How is the sensitivity of each parameter in BuildSheriff to the effectiveness of BuildSheriff? (Sec. 4.5)

**Data Set.** We used the 100 broken builds with exception failures and the 100 broken builds with assertion failures from RQ3 of our empirical study in Sec. 2.1 as the data set for our evaluation. The ground truth has already been constructed during the analysis of RQ3.

**Comparison Approaches.** For RQ4 and RQ5, we selected three state-of-the-art approaches: i) IFrame, which triages crash reports using the top frame in stack traces. We selected it because it is practically used in Mozilla [17]. ii) IFile, which triages crash reports using the name of the source file in which the top frame is defined. We selected it as it achieved the highest triage precision in a recent empirical study [12]. iii) ReBucket, which triages crash reports based on the number of functions in two stack traces, the distance of those functions from the top frame, and the offset distance between the matched functions [15]. We selected it as it outperformed the triage method in Microsoft’s Windows Error Reporting system. Moreover, given the results in RQ3, we developed one naive method 1Cluster, which simply puts all test failures into a single cluster.

**Evaluation Metrics.** To comprehensively evaluate the triage effectiveness, we used the number of builds whose failures were correctly triaged (referred to as C.B.) as an overall indicator of triage effectiveness, used the number of executions of all tests a developer needs to run (referred to as T.E.) and the number of test failures a developer needs to inspect (referred to as T.F.) in the two usage scenarios discussed in Sec. 3.5 to reflect the effort saving for a developer when using BuildSheriff, and used the number of missed root
causes when a developer inspects one test failure per cluster (referred to as M.R.) to measure the impact of ineffective triage. Further, as the goal of BuildSheriff is to produce a set of clusters of test failures whose root cause is considered as the same, we need to compare the distance between the clusters generated by BuildSheriff and the ground-truth clusters manually constructed. To this end, we used all the five families of clustering evaluation metrics in Amigó et al. [4] as different families capture different perspectives of four clustering qualities, i.e., cluster homogeneity (a cluster should not mix test failures with a different root cause), cluster completeness (test failures with the same root cause should be grouped into the same cluster), rag bag (introducing noise into a noisy cluster is less harmful than introducing noise into a clean cluster), and cluster size versus quantity (a small error in a big cluster should be preferable to a large number of small errors in small clusters). We briefly list the five families as follows, and the detailed definition and the satisfaction level of the four qualities can be found in Amigó et al. [4].

- Metrics based on set matching: Purity, Inverse Purity (I.P.), and their combination F-Measure (F.M.).
- Metrics based on counting pairs: Rand Statistic (Rand), Jaccard Coefficient (Jac.), and Folkes and Mallows (F.&M.).
- Metrics based on entropy: Entropy (Ent.), Class Entropy (C.E.), Mutual Information (M.I.), and Variation of Information (V.I.).
- Metric based on edit distance: Edit Distance (E.D.).

For Ent., T.E., T.F., M.R., C.E., V.I. and E.D., the lower the better, and for the others, the higher the better.

### 4.2 Effectiveness Evaluation (RQ4)

Table 1 shows the results of 1Cluster, 1File, 1Frame, ReBucket and BuildSheriff with respect to the 20 effectiveness metrics on the 100 broken builds with exception failures. T.E. and T.F. for Usage Scenario 1 are denoted as T.E.1 and T.F.1, while T.E. and T.F. for Usage Scenario 2 are denoted as T.E.2 and T.F.2. The naive method 1Cluster achieved worse performance on 16 of the 20 metrics than BuildSheriff. Specifically, with 1Cluster in Usage Scenario 1, developers would inspect 263 of the 1,903 test failures and run 118 executions of all tests, which are 81.5% and 13.5% more than with BuildSheriff. In other words, BuildSheriff would save 81.5% of the test failure inspection effort and 13.5% of the test execution effort. With 1Cluster in Usage Scenario 2, developers would inspect 124 test failures and run 124 executions of all tests, which are 3.1% less and 18.1% more than with BuildSheriff. In other words, BuildSheriff would save 18.1% of the test execution effort at the cost of 3.1% more test failure inspection effort. Besides, developers would miss 18 of the 124 root causes when only inspecting one test failure per cluster with 1Cluster, which is 350.0% more than with BuildSheriff.

On the other hand, the state-of-the-art methods 1File, 1Frame and ReBucket had similar results on all the metrics except for T.F.1, T.E.2, C.E., VI. and E.D. on which 1File achieved better results, while BuildSheriff outperformed them on all the metrics. In particular, BuildSheriff significantly improved the best of the three state-of-the-art methods on all the metrics except for T.E.1, T.E.2, Purity, F.M. and B.Rec., i.e., by 46.9% on C.B., 62.6% on T.F.1, 56.0% on T.F.2, 42.9% on M.R., 16.0% on C.E., 13.4% on F.M., 27.9% on Rand, 28.7% on Jac., 40.0% on Ent., 96.3% on C.E., 22.9% on M.I., 89.5% on V.I., 57.0% on E.D., 18.7% on B.Rec., and 15.5% on B.F.M.

Table 2 shows the results on the 100 broken builds with assertion failures. Compared to 1Cluster, BuildSheriff achieved better performance on 13 of the 20 metrics. Although developers would inspect 148 and 137 of the 490 test failures with 1Cluster in Usage Scenario 1 and 2, which are 5.1% and 12.2% less than with BuildSheriff, developers would run 125 and 137 executions of all tests, which are 12.8% and 35.6% more than with BuildSheriff. In other words, BuildSheriff would save 12.8% and 35.6% of the test execution effort at the cost of 5.1% and 12.2% more test failure inspection effort. Besides, developers would miss 25 of the 137 root causes when only inspecting one test failure per cluster with 1Cluster, which is 2400.0% more than with BuildSheriff.

On the other hand, 1File and ReBucket achieved similar results but had better results than 1Frame on most of the metrics, while BuildSheriff outperformed 1File, 1Frame and ReBucket on all the metrics except for F.M. In particular, BuildSheriff significantly improved the best of the three state-of-the-art methods on all the metrics except for F.M. in particular. BuildSheriff significantly improved the best of the three state-of-the-art methods on all the metrics except for F.M. In particular.
We analyzed the average end-to-end time overhead of the five triage methods over the 200 broken builds. 1Cluster, 1FILE, 1FRAME, ReBucket and BuildSheriff respectively took 0.01, 0.10, 0.10, 0.13 and 1.61 seconds to triage a build. We excluded the time overhead of constructing the file dependency graph (which on average took 15.48 seconds for each project) from the time overhead of BuildSheriff as it was a one-time job for each project. Instead, BuildSheriff took 0.54 seconds to update the file dependency graph. While being the slowest due to code change analysis, BuildSheriff is still practical for CI.

Summary. BuildSheriff significantly outperformed the naive method and the best of the state-of-the-art methods on 13% of the 20 metrics in triaging exception failures and assertion failures in CI builds. BuildSheriff achieved a test execution effort saving of up to 35.6% at the cost of up to 12.2% more test failure inspection effort when compared to 1Cluster, and achieved a test execution effort saving of up to 15.8% and a test failure inspection effort saving of up to 35.6% at the cost of up to 12.2% more test failure inspection effort.

4.3 Efficiency Evaluation (RQ5)

We analyzed the average end-to-end time overhead of the five triage methods over the 200 broken builds. 1Cluster, 1FILE, 1FRAME, ReBucket and BuildSheriff respectively took 0.01, 0.10, 0.10, 0.13 and 1.61 seconds to triage a build. We excluded the time overhead of constructing the file dependency graph (which on average took 15.48 seconds for each project) from the time overhead of BuildSheriff as it was a one-time job for each project. Instead, BuildSheriff took 0.54 seconds to update the file dependency graph. While being the slowest due to code change analysis, BuildSheriff is still practical for CI.

Summary. BuildSheriff took 1.61 seconds to triage test failures in a CI build, which was acceptable for practical usage in CI.

4.4 Ablation Study (RQ6)

As BuildSheriff adopts three strategies for triaging exception failures and two strategies for triaging assertion failures, we broke down the broken builds with respect to the strategies that triaged them. 58 of the 100 broken builds with exception failures were triaged by S1a, and 56 of them were correctly triaged; 28 were triaged by S2a, and 25 of them were correctly triaged; and 14 were triaged by S2, and 13 of them were correctly triaged. 56 of the 100 broken builds with assertion failures were triaged by S1b, and all of them were correctly triaged; 44 were triaged by S2b, and 33 of them were correctly triaged. These results demonstrate that each strategy contributes to the effectiveness of BuildSheriff. As S1a and S2a are not related to stack traces and thus can also be used in combination with 1FILE, 1FRAME and ReBucket, we used the 42 broken builds with exception failures that were not triaged by S1a or S2a, and all of them were correctly triaged; and 44 were triaged by S2b, and 33 of them were correctly triaged. To further investigate the advantage of S2b, we used the 44 broken builds with assertion failures that were not triaged by S2b to further investigate the advantage of S2b.

Table 3 reports the results of 1FILE, 1FRAME, ReBucket, BuildSheriff and its variants with respect to the 20 metrics on the 42 broken builds with exception failures. We can observe that if we only used S2b (i.e., the fifth row in Table 3), S2b still outperformed 1FILE, 1FRAME and ReBucket on all metrics thanks to its consideration of code changes, but achieved worse results than a pipeline combination of S1a and S2a (i.e., the last row). If we gave the same weight to each frame in a stack trace (i.e., Naive S2b in the sixth row), Naive S2b had a degradation on all metrics, compared to S2b. If we only distinguished whether a frame was changed or not without considering change impacts (i.e., Simple S2b in the seventh row), Simple S2b achieved better results than Naive S2b but had worse results than S2b. These results show the advantage of considering code changes and their impacts during stack trace similarity measure. Besides, if we only used S2b (i.e., the eighth row), S2b achieved worse results than S2b and a pipeline combination of S1a and S2b on most of the metrics. If we adopted a weighted combination of S2a and S2b by equally weighting stack trace similarity and exception
message similarity (i.e., $S^2_{\alpha} + S^2_e$ in the ninth row), $S^2_{\alpha} + S^2_a$ suffered a degradation on most of the metrics. These results indicate the advantage of combining $S^2_e$ and $S^2_a$ in a pipeline. Further, if we adopted $S^2_a$ from assertion failure triage (i.e., the tenth row), $S^2_a$ had the worst results on most of the metrics. It demonstrates the rationality of not considering test code similarity for exception failure triage.

Table 4 reports the results of 1File, 1Frame, ReBucket, BuildSheriff and its variants with respect to the 20 metrics on the 44 broken builds with assertion failures. We can see that $S^2_\alpha$ (i.e., the last row in Table 4) outperformed 1File, 1Frame and ReBucket on all metrics except for $TF_1$, $TF_2$, $IP$, $F&M$, C.E., E.D. and B.Rec. It shows that stack traces provide a limited triage capability for assertion failure. If we gave the same weight to each test code token (i.e., Naive $S^2_{\alpha}$ in the fifth row), Naive $S^2_{\alpha}$ had a degradation on 16 of the 20 metrics, compared to $S^2_\alpha$. If we only distinguished whether a test code token was changed or not without considering change impacts (i.e., Simple $S^2_e$ in the sixth row), Simple $S^2_e$ achieved better results on 15 of the 20 metrics than Naive $S^2_{\alpha}$ but had worse results on all metrics than $S^2_{\alpha}$. These results demonstrate the advantage of considering code changes and their impacts during test case similarity measure. Moreover, if we adopted $S^2_{\alpha}$, $S^2_e$ and a pipeline combination of $S^2_{\alpha}$ and $S^2_e$ from exception failure triage (i.e., the seventh to ninth rows), they had worse results on 14 of the 20 metrics. It indicates a stronger capability of test code not than stack traces and assertion messages for assertion failure triage.

**Summary.** Each triage strategy contributes to the effectiveness of BuildSheriff in triaging exception and assertion failures.

### 4.5 Parameter Sensitivity Analysis (RQ7)

Each triage strategy in BuildSheriff has some configurable parameters. We tuned these parameters in three ways. For $\Delta_e$ in $S^1_{\alpha}$ and $\Delta_a$ in $S^1_{\alpha}$, we configured them from 1 to 10 by a step of 1 to evaluate their impact on the triage results of $S^1_{\alpha}$ and $S^1_a$. The results are reported in Fig. 8a and 8b, where the x-axis denotes the value of $\Delta_e$ and $\Delta_a$, and the y-axis denotes the number of correctly/incorrectly triaged builds by $S^1_{\alpha}$ and $S^1_a$. As $\Delta_a$ and $\Delta_e$ increased, both the number of correctly and incorrectly triaged builds increased. As $S^1_{\alpha}$ and $S^1_a$ are desired to achieve a low number of incorrectly triaged builds so that other strategies can be tried, a small value of 7 and 3 was set to $\Delta_a$ and $\Delta_e$.

For $c_e$, $dt_{st}$ and $size_{st}$ in $S^2_{\alpha}$, $dt_{msg}$ in $S^3_\alpha$, and $c_a$ and $dt_{tc}$ in $S^2_a$, we configured $c_e$ and $c_a$ from 0.1 to 2 by a step of 0.1, $dt_{st}$, $dt_{msg}$ and $dt_{tc}$ from 0.51 to 1.0 by a step of 0.01, and $size_{st}$ from 1 to 50 by a step of 1 to evaluate their impact on the triage results of BuildSheriff. Here $\Delta_e$ and $\Delta_a$ was fixed to 7 and 3. Parameters in $S^2_{\alpha}$ and $S^2_a$ were tuned together, and 2.5 million (i.e., $20 \times 50 \times 50 \times 50$) configurations were ran to obtain the optimal configuration. Similarly, 1000 (i.e., $20 \times 50$) configurations were ran to obtain the optimal configuration of the two parameters in $S^2_a$. The results are shown in Fig. 8c-8h, where the x-axis denotes the parameter value, and the y-axis denotes the number of correctly triaged builds by BuildSheriff. As $c_e$ and $c_a$ increased, the number of correctly triaged builds increased and then stabilized. As $dt_{st}$ and $dt_{msg}$ increased, the number of correctly triaged builds decreased due to the increasingly strict cluster distance threshold. As $size_{st}$ increased, the number of correctly triaged builds first increased, then decreased and finally stabilized. As $dt_{tc}$ increased, the number of correctly triaged builds was stable at first, then increased and finally decreased. The optimal configuration of $c_e$, $dt_{st}$, $size_{st}$, $dt_{msg}$, $c_a$ and $dt_{tc}$ is 1.7, 0.57, 4, 0.73, 1.8 and 0.82. As these parameters changed, the number of correctly triaged builds was always above 79, which was significantly higher than the achieved results of 1File, 1Frame and ReBucket. In that sense, the effectiveness of BuildSheriff is not very sensitive to these parameters.

For $mDt_e$ in $S^3_e$, we heuristically set it as one plus the maximum of the distances of the frames that reach changed files on $G$ for all the 100 broken builds with exception failures. We heuristically set $mDt_a$ in $S^2_a$ in the same way. In this way, both $mDt_a$ and $mDt_e$ were configured to 6. We used such a heuristic way that the parameter configuration space is reduced, and frames and test code tokens that do not reach any changed file can still contribute to triage.

Note that in Sec. 4.2 and 4.4, we reported the results of the optimal configuration of BuildSheriff. For 1File, 2Frame and ReBucket, we also reported the results of their optimal configuration.

**Summary.** Overall, the sensitivity of the configurable parameters to the effectiveness of BuildSheriff is acceptable.

### 4.6 Discussion

**Assumptions.** As BuildSheriff is designed for test failure triage in CI builds, it can be used for software systems whose development...
process follows the continuous integration practice, where developers frequently merge their code changes into a central repository and builds and tests then run. Therefore, if the continuous integration practice is not followed, or the continuous integration practice is followed but the software system has a small number of tests, BuildSheriff will become not applicable, or be less helpful.

**Threats.** One threat to our evaluation is that the size of our data set is not large. This is primarily due to the expensive effort in building the ground truth as we are not familiar with the business logic of open-source projects. We plan to collaborate with our industrial partners to deploy our tool into their CI to get developers’ feedback. Besides, BuildSheriff is evaluated based on broken builds across heterogeneous projects. We plan to evaluate BuildSheriff within the build history of one project. In addition, two usage scenarios in Sec. 3.5 are considered in our evaluation, and there could be other usage scenarios. However, we believe the two usage scenarios are representative in practice. It is also hard to determine whether test execution error or test failure inspection effort is more expensive, and thus we evaluate the triage effectiveness with both dimensions.

**Limitations.** First, BuildSheriff only triages test failures, but cannot directly pinpoint the root causes in source code which would be useful for developers to fix test failures. Second, some engineering effort is needed to extend BuildSheriff to support other programming languages, other CI services and other build tools by providing specific implementations for triage knowledge preparation. Third, the design of BuildSheriff does not consider flaky tests. If a flaky test is triaged into a cluster that has a different root cause from flakiness, and is selected as the representative test for manual inspection, the root cause of the cluster will not be fixed. Therefore, flaky tests could reduce the effort saving capability of BuildSheriff. To mitigate this problem, we can apply flaky test detection techniques (e.g., [7]) before BuildSheriff. Notice that there is no failure that is due to flakiness in our evaluation data set. Last, exception messages may not be available for some failed tests. In all the failed tests in 80,778 broken builds with exception failures, 89.0% of the failed tests contain exception messages. Without such messages, $S^2_3$ becomes useless. In the extreme case, BuildSheriff without $S^2_3$ still outperforms the existing methods (i.e., the fifth row in Table 3).

5 RELATED WORK

**Stack Trace-Based Failure Triage.** Brodie et al. [11] used the longest subsequence of common functions in stack traces as the indicator of stack trace similarity to identify re-occurring failures. To improve it, Brodie et al. [10] and Modani et al. [47] removed recursion and uninformative functions from stack traces before stack trace matching.

Bartz et al. [6] proposed a machine learning technique to find similar stack traces. However, this technique requires features (e.g., process name) that are often not available in test failures in CI.

Glerum et al. [23] proposed the crash bucketing algorithm in Windows Error Reporting (WER) by analyzing memory dumps collected from users. This technique requires knowledge (e.g., heap corruption) that are not available in test failures in CI. Kim et al. [39] proposed crash graphs to provide an aggregated view of the stack traces in the same bucket in WER. Differently, Koopaei and Hamou-Lhadji [40] modeled and abstracted the stack traces in the same bucket as an automaton. Dang et al. [15] improved the algorithm in [23] by measuring similarity between two stack traces. However, it needs to learn parameters’ value from historical buckets.

Mozilla grouped crash reports by the top function in the stack trace, which was then improved by Dhaliwal et al. [17]. Lerch and Mezini [43] compared stack traces with TF-IDF to group bug reports.

These techniques use stack trace similarity to triage crash or bug reports. Differently, our work is focused on triaging test failures for each CI build. In this context, code changes in each build provide a good source of knowledge for triage, and thus we involve code changes in measuring stack trace and test code similarity. Moreover, these techniques are not applicable to assertion failures as such failures do not report any informative stack trace, but our work does.

To the best of our knowledge, one closely related test failure triage work was recently proposed by Golagha et al. [25]. It targeted test failures in CI. However, this technique was specifically designed for automotive industry, and thus the general, bug history and test case similarity features are not available in test failures in general CI.

**Proiling-Based Failure Triage.** Podgurski et al. [52] used profiles of failure executions and successful executions to group failures. Francis et al. [22] improved it by refining the initial classification of failures. Liu et al. [45] measured the failure similarity on dynamic slices. Given failing and passing executions, Liu et al. [44, 45] located bug location for each failure, and regarded two failures as similar if roughly the same bug location was suggested. DiGiuseppe and Jones [19] clustered failures with the semantic concepts that were expressed in the executed source code. Golagha et al. [26] clustered failed tests with execution profiles. However, they would impose large overhead due to profiling, which are not practical for CI [46].

Cui et al. [14] triaged crashes based on program semantics reconstructed from memory dumps. Pham et al. [51] used the symbolic execution tree to cluster failed tests. van Tonder et al. [65] used the approximated fixes to group crashes. These techniques aim to achieve semantic-aware crash triage, but would suffer scalability issue in CI.

**Text-Based Failure Triage.** The textual information (e.g., title and description) in bug reports are used to identify duplicate bug reports. Various techniques have been used to achieve this purpose, e.g., natural language processing [57, 68], machine learning [9, 35, 41, 58, 62], information retrieval [13, 32, 54, 61, 63, 64], and topic modeling [2, 3, 49]. However, for test failure triage in CI, there is no such descriptive text that can be utilized.

**Crash Analysis.** Other crash analysis methods include fault localization [27, 53, 70, 72, 74] and failure-inducing change identification [69, 71, 75], crash prioritization and assignment [38, 60], crash reproducing [5], and crash root cause classification [28, 29, 36, 56]. It is interesting to explore how they can practically work in CI.

6 CONCLUSIONS

In this paper, motivated by our large-scale study on test failures in CI, we propose a new change-aware approach, BuildSheriff, to triage test failures in each CI build. Our evaluation has indicated that BuildSheriff can significantly improve the state-of-the-art and be practically used in CI with a time overhead of 1.61 seconds for a build.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (Grant No. 61802067).
REFERENCES


[2] Kavita Aggarwal, Fanbarr Timbers, Tanner Rogers, Abram Hindle, Eleni Strouli- 

a, and Russell Greiner. 2017. Detecting duplicate bug reports with software 

engineering domain knowledge. Journal of Software: Evolution and Practice 29, 3 


for more accurate duplicate bug report detection. In Proceedings of the 10th 


parison of Extrinsic Clustering Evaluation Metrics Based on Formal Constraints. 

Information Retrieval 12, 4 (2009), 461–486.

[5] Shai Artzi, Sunghun Kim, and Michael D Ernst. 2009. ReCrash: a tool for cap-

turing and reproducing program crashes in deployed applications. In Proceed-

ings of the 7th joint meeting of the European software engineering conference and 

the ACM SIGSOFT symposium on the foundations of software engineering. 295–296.


Proceedings of the Third Conference on Tackling Computer Systems Problems with 


and Darko Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In 

Proceedings of the IEEE/ACM 18th International Conference on Software Engineering. 

433–444.


the build: An explorative analysis of Travis CI with GitHub. In Proceedings of the 


Duplicate bug reports considered harmful. really? In Proceedings of the 


Natwar Modani, Mark Wilding, Jon Chaplin, and Peter Sohn. 2005. Quickly 

Finding Known Software Problems via Automated Symptom Matching. In Pro-


problem determination using call-stack matching. Journal of Network and Systems 


unreasonable effectiveness of traditional information retrieval in crash report 


Software Repositories. 269–280.


formulating queries for duplicate bug report detection. In Proceedings of the IEEE 

26th International Conference on Software Analysis, Evolution and Reengineering. 

218–229.

[14] Weidong Cui, Marcus Pinados, Sang Kil Cha, Yanick Pratatonio, and Vasileios P. 

Kemerlis. 2016. RETRacer: Tracing Crashes by Reverse Execution from Partial 

Memory Dumps. In Proceedings of the 38th International Conference on Software 

Engineering. 820–831.


Relucket: A Method for Clustering Duplicate Crash Reports Based on Call 

Stack Similarity. In Proceedings of the 34th International Conference on Software 

Engineering. 1084–1093.


glomerative hierarchical clustering methods. Journal of Classification 1, 1 (1984), 

7–24.


reports for fixing bugs: A case study of Mozilla Firefox. In Proceedings of the 17th 


[19] Nicholas DiGiuseppe and James A Jones. 2012. Concept-based failure cluster-

ing. In Proceedings of the ACM SIGSOFT 26th International Symposium on the 


improving software quality and reducing risk. Pearson Education.

[21] Nicole Forsgren, Gene Kinne, Jim Humble, Alanna Brown, and Nigel Kersten. 


download/10069/143970/file/2017-state-of-devops-report.pdf

[22] Patrick Francis, David Leong, Melinda Mitchell, and Andy Podgurski. 2004. Tree-

based methods for classifying software failures. In Proceedings of the 15th Inter-


[23] Kirk Glerum, Kinshuman Kishunthman, Steve Greenberg, Gabriel Aul, Vince 


Debugging in the (Very) Large: Ten Years of Implementation and Experience. In 


Symposium on Software Testing and Analysis. 211–222.


ACM SIGSOFT International Symposium on Software Testing and Analysis. 

134–145.


Reducing failure analysis time: An industrial evaluation. In Proceedings of the 

IEEE/ACM 39th International Conference on Software Engineering: Software Engi-

neering in Practice Track. 293–302.

[27] Chao Liu, Xiangyu Zhang, and Jiawei Han. 2008. A systematic study of failure 
