# Which Builds Are Really Safe to Skip? Maximizing Failure Observation for Build Selection in Continuous Integration

Xianhao Jin[a,*], Francisco Servant[b,a,1,*]

[a] *Department of Computer Science, Virginia Tech, Virginia, United States of America*
[b] *Departamento de Teoría de la Señal y las Comunicaciones y Sistemas Telemáticos y Computación, Universidad Rey Juan Carlos, Madrid, Spain*

**Abstract**

Continuous integration (CI) is a widely used practice in modern software engineering. Unfortunately, it is also an expensive practice. Google and Mozilla estimate their expenses for their CI systems in millions of dollars. To reduce the cost of CI, researchers developed multiple approaches to reduce its computational workload requirements. However, these approaches sometimes make mispredictions and skip failing builds which are not desirable to be skipped. Thus, in this paper, we aim to save computational cost in CI, while also maximizing the observation of failing builds, *i.e.,* to skip builds more safely. First, we perform empirical studies to understand which builds are safe to skip, starting from CI-Skip rules [1] that characterize builds that developers decide to skip. We observe that CI-Skip rules are not so safe as expected. We then develop a collection of CI-Run rules that can complement these rules. Based on our findings, we propose PreciseBuildSkip, a novel approach that maximizes build failure observation and reduces the cost of CI through the strategy of build selection. We evaluate our approach and results show that our approach saved more cost (5.5%) than the safest existing technique but reduced the falsely skipped failing builds from 4.1% to 0% (median value).

*Keywords:* continuous integration, build prediction, safety, maintenance cost

## 1. Introduction

Continuous integration (CI) is a popular practice in modern software engineering that encourages developers to build and test their software in frequent intervals [2]. For simplicity and consistency with previous studies [3], we refer as *build* to the full process of building the software and running all the tests when CI is triggered.

While CI is widely recognized as a valuable practice, it is also high-cost. The cost of CI is characterized in past work as the computational workload that it requires — to regularly execute software builds [4, 5, 6, 7, 8]. This is the definition of the cost of CI that we use in this paper. Adopting CI can be very expensive — *i.e.,* executing its high computational workload can often require a high Budget: Google estimates the cost of running its CI system in millions of dollars [4], and Mozilla estimates theirs as $201,000 per month [9]. For smaller-budget companies that have not yet adopted CI, this high cost can pose a strong barrier.

Some existing research approaches aim to save cost in CI — *i.e.,* to reduce its computational workload requirements. Most past works follow the premise that observing failing executions (builds or tests) is more valuable to developers than observing passing ones — since failures present actionable feedback. So, they automatically predict and skip executions that would likely pass — to save the cost of executing them. Most of these techniques use heuristics or machine-learning algorithms for their predictions.

A popular approach to this goal in previous works is to automatically predict and skip passing test cases. Past approaches were proposed to skip, *e.g.,* tests that historically failed less [8, 10], that have a long distance with the code changes [11], that test unchanged modules [12], or that are predicted to pass by a machine learning classifier [13]. Techniques to skip the execution of passing tests — to reduce the cost of testing — were proposed even before CI was a popular practice. These are known as regression test selection (RTS) techniques *e.g.,* [14, 15, 16, 17, 18, 19, 20].

Another, more recent, approach is to predict and skip passing builds, *e.g.,* [3]. This approach has the potential for higher cost savings — when a build is skipped, it saves the cost of running all its tests as well as its build-preparation steps. Finally, other past approaches predict and skip builds that developers would have manually skipped [21, 1]. When asked about the characteristics of the builds that they skip, developers for the most part describe builds that will likely pass, *i.e.,* they skip builds with: non-source code changes, with no test coverage, with trivial source code changes, or with other likely-to-pass characteristics, *e.g.,* refactoring changes [1].

*Corresponding authors.
*Email addresses:* xianhao8@vt.edu (Xianhao Jin), francisco.servant@urjc.es (Francisco Servant)
[1] Some work performed while at Virginia Tech.

Unfortunately, since these techniques make predictions, they may also make mistakes, resulting in either: missed opportunities to save cost (not-skipped passing executions), or missed observations of failures (skipped failing executions). We aim to minimize the latter kind of mistakes — *i.e.,* to **maximize failure observation**, and we specifically target **build selection** techniques. Build selection techniques carry a trade-off: as they skip more builds, they save more cost, but they are also more likely to skip builds that would have failed. We believe that many practitioners may prefer a build selection technique that maximizes its **safety** (*i.e.,* failure observation ratio) — even if it may save less cost than other approaches. We aim to help those practitioners in this paper.

We perform two empirical studies to better understand which builds are safe to skip. Recent work studied the characteristics of builds that developers **manually decided to skip**, and encoded them into rules [1]. We will refer to these as **CI-Skip rules**. While it would be intuitive to assume that developers decide to skip builds that are guaranteed to pass, the actual safety of these *seemingly-safe* CI-Skip rules is yet unknown. First, we study the **benefit** (*i.e.,* how much cost (number of builds) can be saved) and **safety** (*i.e.,* how many failures can be observed) of CI-Skip rules. Next, we study **why** CI-Skip rules sometimes capture failing builds, and develop a set of **CI-Run rules** to complement them, increasing their safety.

Additionally, we encode the findings of our empirical studies in an **automated build-selection technique**, PreciseBuildSkip (PBS), to predict the outcome of builds as safely — *i.e.,* to correctly predict as many build failures — as possible. PreciseBuildSkip uses a random-forest classifier for prediction, with CI-Skip rules and CI-Run rules as features. We also evaluated PreciseBuildSkip's performance in different scenarios and compared it with existing build selection approaches.

We performed multiple observations in our studies. First, we observed that **no CI-Skip rule is completely safe** — all CI-Skip rules captured some builds that ended up failing. Generally, as CI-Skip rules provided higher potential cost savings, they also skipped more failing builds. Therefore, CI-Skip rules cannot be used *as-is* to safely skip builds. Developers that manually used CI-Skip rules to skip builds would miss the observation of some build failures (generally, more so for CI-Skip rules that save more cost).

Second, we identified **four main CI-Run rules** why builds under CI-Skip rules may fail: (1) changes in build scripts, (2) in configuration files, (3) subsequent failures, and (4) increasing platform numbers. We observed that at least one of these CI-Run rules tends to be present when builds fail under CI-Skip rules. In particular, the *subsequent-failure* CI-Run rule was correlated with build failures for all CI-Skip rules. That is, the most common reason why builds under CI-Skip rules failed is that they were subsequent to another build failure, *e.g.,* a build that does not change source-code files may still fail if the previous one failed (*i.e.,* it was already broken and these changes did not fix it).

Third, our proposed safe approach to build selection, PreciseBuildSkip, **provided both higher cost saving and failure observation rates than the state of the art** build-selection techniques: Abd19 [1], Abd20 [21], and Jin20 [3].

We designed PreciseBuildSkip with customizable tendency to predict builds to pass. A higher tendency to predict builds to pass will achieve a higher ratio of skipped builds — and thus higher cost savings, but it may also result in higher rates of mistakenly skipped failing builds. In our experiments, to compare with the results of existing build-selection techniques, we highlighted four values of PreciseBuildSkip's prediction tendency: PBS_Safe, PBS_Moderate, PBS_Relaxed, and PBS_More_Relaxed (from lower to higher tendency to predict passing builds).

When customized (PBS_Relaxed) to save as much effort as the highest-effort-saving previous technique (Abd19 saved 22.3% build executions), PBS_Relaxed provided higher safety (PBS_Relaxed observed 87.61% failures compared to 80.7% by Abd19). When customized (PBS_Moderate) to provide as much safety as the safest existing technique (Abd20 observed 96% build failures), PBS_Moderate provided higher cost savings (PBS_Moderate saved 12.9% failures compared to 5.2% by Abd20). When customized (PBS_Safe) for highest safety, PBS_Safe observed 100% build failures, while still saving 5.5% of build executions.

Finally, our new approach outperformed existing build selection approaches when comparing all variants' abilities of predicting build failures with the corresponding build selection approach. We also found that the performance of PreciseBuildSkip is not impacted by the previously self-impacted train data set. Besides, the executing time of PreciseBuildSkip is negligible compared to its saved duration. We then performed an additional analysis to understand the extent to which our CI-Run rules benefitted PreciseBuildSkip's effectiveness. We observed that the variants of PreciseBuildSkip that applied CI-Run rules as features provided higher effectiveness than the corresponding variants without them.

This paper provides the following contributions:

- The first empirical study to understand the cost-saving ability (the ratio of builds that they can skip) and safety (the ratio of failing builds that they can observe) of CI-Skip rules.
- A collection of CI-Run rules, that explain why CI-Skip rules sometimes characterize builds that will fail, and that complement them to make them safer.
- A customizable, automated approach (PreciseBuildSkip) that saves cost in CI by automatically predicting and skipping builds that are

Published version available at: https://doi.org/10.1016/j.jss.2022.111292

likely to pass, and that is safer and saves more cost than the state-of-the-art build-selection techniques.

- A novel evaluation metric for build-selection techniques (SFRD), that provides a balanced measurement of the Cost Saving and Observed Failures metrics.
- An evaluation of the overhead of PreciseBuildSkip by comparing its build time saved with its required execution time.
- A study of the impact of CI-Run rules on the effectiveness of PreciseBuildSkip
- An evaluation of the practicality of PreciseBuildSkip in terms of how its effectiveness is impacted when it is trained on projects that already apply build selection.

We also include a replication package for our paper [22].

## 2. Related Work

### *2.1. Characterizing Builds*

To the extent of our knowledge, only Abdalkareem *et al.* [1] aimed to characterize CI-Skip builds. They proposed a human study to understand reasons why developers decide to skip builds. Then they designed a rule-based technique based on CI-Skip rules from those reasons and evaluated the cost-saving ability of their approach. In contrast, we included more CI-Skip rules in this paper and evaluated both the cost-saving ability and safety for each single rule. We then explored CI-Run rules to complement CI-Skip rules and encoded our findings into a new approach that can better discriminate passing and failing builds.

Other studies investigated the reasons for build failures. Some studies [23, 24] sort common build failures into compilation [25], unit test, static analysis [26], and server errors. Paixão *et al.* [27] studied the interplay between non-functional requirements and failing builds. Other studies found factors that contribute to build failures: architectural dependencies [28, 29] and other more specific factors, such as the stakeholder role, the type of work item and build [30], or the programming language [31]. Other less obvious factors that could cause build failures are build environment changes or flaky tests [32]. Some work [32][3] also found that build failures tend to occur consecutively, which Gallaba *et al.* [33] describe as "persistent build breaks".

Other studies found change characteristics that correlate with failing builds, such as: code churn [32, 34], build tool [34], and statistics on the last build and the history of the committer [35]. Hassan *et al.* [36] found that build scripts (BuildScripts) can result in build failures and proposed an approach to fix this kind of failing build automatically. Our approach explored CI-Run rules that can invalid CI-Skip rules including BuildScripts and use them for build predictions. Jin and Servant [3] differentiated first and subsequent failures and developed a predictor for first failures. Others [37, 35] studied to predict the build outcomes based on these findings. In contrast, PreciseBuildSkip focuses on detecting builds that can be safely skipped in CI and maximizing the observed failing builds while providing some cost-savings.

### *2.2. Empirical Studies of CI and its Cost*

There are multiple works focusing on understanding the practice of CI, in dimensions of both practitioners *e.g.,* [4] and software repositories [38]. Stahl *et al.* [39] and Hilton *et al.* [4] studied the benefits and costs of CI usage, and the trade-offs between them [5]. Lepannen *et al.* similarly studied the costs and benefits of continuous delivery [40]. Zhao *et al.* tried to understand the impact of CI in other development practices, like bug-fixing and testing [41]. Vasilescu *et al.* studied CI as a tool in social coding [42], and later studied its impact on software quality and productivity [38]. Felidré *et al.* [43] studied the adherence of projects to the original CI rules [2]. Other recent studies focused on the barriers of CI adoption [6] and pain points [7] of CI. Many empirical studies highlighted The high cost of running builds and delaying failing builds observation as an important problem in CI [4, 5, 6, 7, 8] — which reaches millions of dollars in large companies, *e.g.,* at Google [4] and Microsoft [8]. They also highlighted the long waiting duration as the main pain point in those companies [44].

### *2.3. Approaches to Reduce the Cost of CI*

A popular effort to reduce the cost of CI focuses on understanding what causes long build durations *e.g.,* [45, 46]. Thus, most of the approaches skip tests within builds, *e.g.,* tests that historically failed less [8, 10], that have a long distance with the code changes [11], that test unchanged modules [12], or that are predicted to pass by a machine learning classifier [13]. These techniques are based on regression test selection (RTS) *e.g.,* [14, 15, 16, 17, 18, 19, 20]. While these techniques focus on making every build cheaper, our work addresses the cost of CI differently: by reducing the total number of builds that get executed. A related recent technique saves cost in CI by running fewer builds [1, 21, 3]. Another recent study compared the benefits of test selection and build selection techniques [47, 48]. Our technique uses a more comprehensive set of CI-Skip rules and adds CI-Run rules as supplement which provides better safety. Our work also aims at solving the main concern of adopting build selection approaches — skipping failing builds.

A related effort for improving CI aims at prioritizing its tasks to provide early fault observation. The most common approach in this direction is to apply test case prioritization (TCP) techniques *e.g.,* [49, 50, 10, 51, 52, 53] so that builds fail faster. Another similar approach achieves faster feedback by prioritizing builds instead of tests [54] when there is a queue of builds waiting for executed under

Published version available at: https://doi.org/10.1016/j.jss.2022.111292

limited computation source. In contrast, our work aims at cost-saving in CI by skipping unfruitful tasks, *i.e.,* only executing a subset of tasks. Prioritization-based techniques advance feedback but are not able to save cost, *i.e.,* all builds and tests still get executed. Finally, other existing efforts to reduce cost in CI make individual builds cheaper, by running less computation in them *e.g.,* [55][56].

## 3. Research Questions

Our goal is to help practitioners skip builds to save cost (*i.e.,* skip passing builds) more safely (*i.e.,* skipping fewer failing builds) than with existing approaches. For that, we perform two empirical studies and three experiments.

First, we empirically study the cost-saving potential and safety of CI-Skip rules, *i.e.,* rules that past work observed developers using to skip builds in practice [1]. Second, we propose a collection of CI-Run rules to capture why CI-Skip rules sometimes include builds that fail, and to make them safer — *i.e.,* capture fewer failing builds.

While the findings of these two studies are useful by themselves to educate practitioners about how to better identify builds that are safe to skip — *i.e.,* that will likely pass, we also create a novel technique to automatically make that decision for them: PreciseBuildSkip. We perform three experiments to evaluate PreciseBuildSkip. First, we evaluate PreciseBuildSkip compared to the state of the art build-selection techniques. This experiment evaluates techniques both in terms of the correctness of their predictions and in terms of the cost-saving ability and safety that they provide. It also measures the overhead introduced by PreciseBuildSkip to build duration — to understand how the cost of running PreciseBuildSkip impacts its provided cost savings. Second, we perform an additional study to understand the impact of considering CI-Run rules in PreciseBuildSkip's predictions. Finally, we study how the predictions of PreciseBuildSkip would be impacted in the scenario where it has been used for some time, and thus its training data has been affected by build selection.

In our studies and experiments, we answer the following research questions:

**Empirical Study 1: Evaluating CI-Skip rules**
**RQ1:** How much cost can each CI-Skip rule save?
**RQ2:** How safe is each CI-Skip rule?

**Empirical Study 2: Supplementing CI-Skip rules with CI-Run rules**
**RQ3:** What proportion of failing builds under CI-Skip rules are covered by our CI-Run rules?
**RQ4:** How helpful are CI-Run rules at discriminating between failing and passing builds under CI-Skip rules?

**Experiment 1: Evaluating PreciseBuildSkip**
**RQ5:** How correct are PreciseBuildSkip's predictions?
**RQ6:** How much cost-saving and safety do PreciseBuildSkip's predictions provide?
**RQ7:** How much overhead does PreciseBuildSkip add to build duration?

**Experiment 2: Understanding the Impact of CI-Run rules**
**RQ8:** What is the impact of including CI-Run rules as features in PreciseBuildSkip?

**Experiment 3: Evaluating PreciseBuildSkip when trained on Builds affected by Build-selection**
**RQ9:** How much cost-saving and safety does PreciseBuildSkip provide when trained on projects that used build selection?

### *3.1. Data Set*

We performed our study over the Travis Torrent dataset [31], which includes 1,359 projects (402 Java projects and 898 Ruby projects) with data for 2,640,825 build instances including changes on all different files such as source files or configuration files. We remove "toy projects" from the data set by studying those that are more than one year old, and that have at least 200 builds and at least 1000 lines of source code, which is a criteria applied in multiple other works [35, 34]. To be able to explore CI-Skip rules on test information, we also filter out those projects whose build logs do not contain any test information. We focused our study on builds with passing or failing outcome, rather than error or canceled. Besides, in Travis a single push or pull-request can trigger a build with multiple jobs, and each job corresponds to a configuration of the building step. As many existing papers have done [33, 57, 58], we considered these jobs as a single build since they share the same build result and duration. After this filtering process, we obtained 82,427 builds from 100 projects (13,464 failing builds).

To be able to implement our approach and replicate the state of the art build-selection techniques (Abd19 [1], Abd20 [21], and Jin20 [3]), we extended the information in TravisTorrent of these 100 projects in multiple ways. First of all, we implemented scripts to download the raw build logs from Travis and parse them to extract all of the information about test executions, such as test name, duration and outcome. Replicating Abd19 [1] and Abd20 [21] required additional information that TravisTorrent does not provide for builds, such as the content of commit messages, changed source lines and changed file names. For that, we also mined additional information about commits in the projects' code repositories through Github such as changed file names and changed line content by running scripts to read the content of commits using Github's API. Finally, we built a dependency graph for the source code of each project using a static code analysis tool (Scitool Understand [59]) to compute the paths between files for implementing CI-Skip rules. For Java projects, we ran Scitool Understand on the command line to scan them. Understand generates a .CSV file with the static dependency graph of the project. For Ruby projects, we obtained their

Published version available at: https://doi.org/10.1016/j.jss.2022.111292

static dependency graph using rubrowser [60]. We used a project's static dependency graph to check if there is a path between changed files and test files.

## 4. Empirical Study 1: Evaluating CI-Skip rules

The goal of this study is to understand the impact that developers would observe when applying CI-Skip rules to decide which builds to skip manually. Existing work [1] recommends to skip builds if any of the CI-Skip rules is met. When applying such CI-Skip rules, developers can obtain cost savings, but they may also mistakenly skip failing builds. Ideally, CI-Skip rules would also be highly safe — they would cause developers to mistakenly skip few failing builds.

We evaluated CI-Skip rules in two dimensions: cost-saving ability and safety, over a large dataset of continuous integration builds (see §3.1). The former reflects how much cost-saving can be achieved by applying each rule, while the latter shows how safe it is to skip builds based on these rules. The results of this study will be useful for developers who are already using CI-Skip rules to manually skip builds, to understand the risk of skipping failing builds that they are incurring, depending on what CI-Skip rules they are applying. They will also inform developers to plan to use CI-Skip rules to skip builds, and want to know which rules save the most cost and incur the lowest risk of skipping passing builds. Next, we describe CI-Skip rules and how we studied our research questions in this study.

### *4.1. Studied Factors: CI-Skip rules*

To the extent of our knowledge, no previous work studied which builds are fully safe to skip, *i.e.,* are guaranteed to pass. The work with the closest goal was Abdalkareem *et al.*'s [1], who captured the characteristics of builds that developers decided to skip. We refer to these rules as **CI-Skip rules**. Our goal in this empirical study is to understand to what extent these CI-Skip rules are actually safe to skip or not, *i.e.,* whether they capture only builds that pass.

We study all the rules from Abdalkareem *et al.*'s work, and we created two additional novel CI-Skip rules as additional rules that would intuitively signal that a build is likely to pass: AllPassingTest and NoReachableTest. We list our studied CI-Skip rules in Table 1, along with a brief description.

**SourceCommentChange (SCC)**: Developers sometimes skip builds whose commits only modify comments in source code. We implement this rule using regular expressions to determine whether each modified source line is a comment change. One could think of this rule as a simple way to capture builds that cannot fail. However, one example of changes in comments that could cause build failures is that of changes in JavaDoc comments, which this rule skips [1]. For example, errors in the Javadoc syntax, the usage of deprecated features in it, or an incorrect Java version may still cause a build failure.

**SourceFormatModification (SFM)**: Developers sometimes choose to skip builds whose commits only modify the format of source code. Abdalkareem *et al.* report this CI-Skip rule as *"Formatting source code without changing the semantic of the code"* [1]. We created the SourceFormatModification rule to capture changes that only change the format of the code.

**SourceFormatCommentChange (SCC_SFM)**: Abdalkareem *et al.*'s implementation of the SourceFormatModification CI-Skip rule is slightly different from how it was described [1]. So, we give their implemented version of SFM a new rule name (SCC_SFM), and we study it separately. SCC_SFM first applies SourceCommentChange (SCC) removing comment lines, it removes all white spaces and new line symbols that are ignored by programming language grammars, and then it checks if the remaining lines modified by the change are the same, *i.e.,* if the change only modifies the code format. Changes fall under this rule whether they change only comments (SourceCommentChange) or they change only comments and format (SourceFormatCommentChange).

**NonSrcFileChange (NSF)**: Sometimes developers decide to skip builds with changes that only touch non-source code files, *e.g.,* ".git" files. Abdalkareem *et al.* originally defined non-source code files as those with a file extension in a pre-defined list[2]. A build falls under this rule if it only changed files with extensions in that list.

**MetaFileChangeOnly (MFC)**: Developers also sometimes skip builds with changes only on meta files. We identified meta files[2] (*e.g.,* ".ignore" or ".git" file) by looking at the extensions of the files modified in the build. We used the same process and extensions as Abdalkareem *et al.*'s study [1].

**VersionRelease (VR)**: Developers sometimes skip a release preparation commit. Following Abdalkareem *et al.*'s study [1], we analyzed the changed files in a build's commits and check if it only modified the version in build scripts, *e.g.,* Maven or Gradle.

**AllPassingTest (APT)**: We created this additional CI-Skip rule. It reflects a criterion by which a build that is safe to skip (*i.e.,* that will not fail) is one in which all its tests pass. We implement it by flagging builds in which none of their tests failed, as stated in its raw build logs. We realize that this rule is not useful for prediction — since the outcome of tests is unknown before a build is executed. However, we decided to add it to this study to empirically understand the safety of this seemingly-strong criterion for anticipating safe builds.

**NoReachableTest (NRT)**: We also created this additional CI-Skip rule, since we believe it could be another

[2]A complete list of the file extensions can be found here: http://das.encs.concordia.ca/publications/which-commits-can-be-ci-skipped/

Published version available at: https://doi.org/10.1016/j.jss.2022.111292

Table 1: Studied CI-Skip rules that can be used to skip CI builds.

| CI-Skip rule | Short Description |
|---|---|
| SourceCommentChange | The commits of this build only change comments in source code. |
| SourceFormatModification | The commits of this build only change the format of source code. |
| SourceFormatCommentChange | The build's commits only change both the source code comments (optional) and format. |
| NonSrcFileChange | The build's commits change no source file. |
| MetaFileChangeOnly | The build's commits only change meta-file. |
| VersionRelease | The build only includes release preparation commits. |
| AllPassingTests | The build has no failing test. |
| NoReachableTest | The build has no test for changed files. |

strong criterion for anticipating safe builds. Additionally, developers report skipping builds *"When tests are not written to work for that particular source branch/repo"* [1]. NoReachableTest flags builds whose tests have no path to the changed files — *i.e.,* the changes in this build are not covered by the tests. We use the static dependency graph to check the existence of the path between the test and the changed files, *i.e.,* if any of the tests are reachable to the changed files in this build. We propose NoReachableTest as a proxy for AllPassingTest that can be used for predicting safe builds — *i.e.,* it can be calculated before builds are executed.

### *4.2. RQ1: How much cost can each CI-Skip rule save?*

To answer this research question, we measured the proportion of builds under each CI-Skip rule, among all the builds in each studied project. We show the distribution of such proportions in Figure 1. For example, if 30% of all builds have only non_source file changes (NonSrcFileChange), it means that developers can save 30% of build effort by skipping builds under this rule.

#### *4.2.1. Result*

Figure 1 shows the cost-saving ability of each CI-Skip rule. We can find that the performance of CI-Skip rules on cost-saving differs from each other. Some CI-skip rules can provide high cost-saving, but others are much less effective. Five of eight rules (SourceCommentChange, SourceFormatModification, SourceFormatCommentChange, MetaFileChangeOnly and VersionRelease) cover a very small proportion of builds (median less than 5%) which shows that they have a low prevalence in all builds. Developers may achieve very low cost-saving by applying these rules. In contrast, AllPassingTest provides really high cost savings (median 95.7%). This means that AllPassingTest represents a majority of passing builds, *i.e.,* those that had no failing tests. While AllPassingTest is not usable in practice to predict build outcomes, this shows us that AllPassingTest is a very promising feature to try to approximate through other features that can be used for prediction, *i.e.,* that can be computed pre-build-execution, such as NoReachableTest. Finally, we also observed that

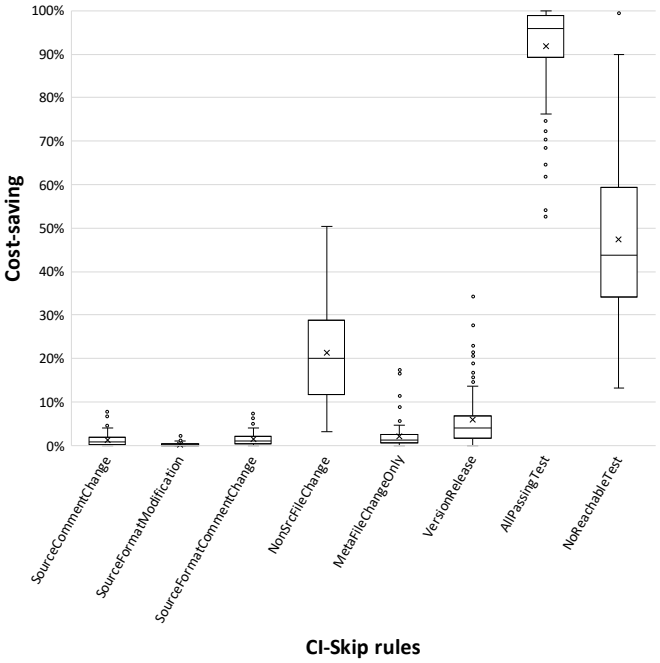


Figure 1: Proportion of builds that CI-Skip rules could save.

NonSrcFileChange and NoReachableTest provide medium cost-saving (20% and 43.8% respectively).

These observations also show us that the majority of builds skipped by CI-Skip rules were skipped by only two rules: NonSrcFileChange and NoReachableTest (with the exception of AllPassingTest). Thus, for developers looking for a simple way to skip builds based on a rule-of-thumb, we could advise them to focus only on NonSrcFileChange and NoReachableTest, and they would save almost the same amount of builds as if they applied every single CI-Skip rule — since all other rules save little cost in comparison.

### *4.3. RQ2: How safe is each CI-Skip rule?*

Ideally, CI-Skip rules not only can save a reasonable amount of builds, but are also safe. To study this second aspect in this research question, we measured the ratio of failing builds among the builds under each CI-Skip rule in each studied project. We show the distribution of such

Published version available at: https://doi.org/10.1016/j.jss.2022.111292

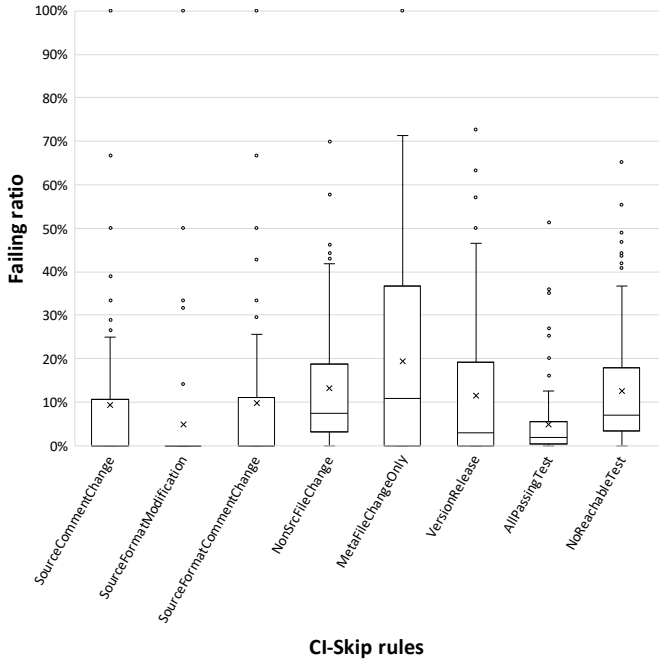


Figure 2: Proportion of failing builds among builds under each CI-Skip rule.

ratios in Figure 2. For example, if 30% of builds with only non_source file changes (NonSrcFileChange) fail, it means that the likelihood to miss a failing build by NonSrcFileChange is 30%.

*4.3.1. Result*

Figure 2 shows that all CI-Skip rules had a relatively low fail ratio (*i.e.,* all their median values are below 11%), but **none of them were completely safe to apply**. Thus, it is not 100% safe to simply use CI-Skip rules to achieve cost-saving in practice.

Among CI-Skip rules, MetaFileChangeOnly had the highest fail ratio (median 11%) which means that relatively often changes in meta files can result in build failures. It also provides low potential cost savings (§4.2.1). Thus, applying MetaFileChangeOnly manually would not be an effective way to safely skip builds.

We also found that SourceCommentChange, SourceFormatModification and SourceFormatCommentChange were highly safe (with median 0% failing builds). Unfortunately, they also provide very few opportunities to save cost, as seen in §4.2.1. NonSrcFileChange and NoReachableTest have a relatively low fail ratio based on our observations and they can also provide considerable cost-saving. Also, NonSrcFileChange is easy to implement in the real world, making it one of the best CI-Skip rules to apply manually in practice.

In summary, we found that CI-Skip rules have limitations: some of them provide few opportunities to save cost, and none are fully safe. Some of them do provide a reasonable trade-off of cost-saving and safety, but since none are fully safe, we propose to not apply them manually. We instead propose an automated technique that predicts which builds to skip using CI-Skip rules as features (§6).

Table 2: Studied CI-Run rules that may override CI-Skip rules.

| CI-Run rules | Short Description |
|---|---|
| BuildScripts | The commits in this build modify build scripts. |
| ConfigurationFiles | The commits in this build modify configuration files. |
| SubsequentFailures | The build has already broken. |
| IncreasingPlatforms | The build is tested in more platforms than its previous build. |

## 5. Empirical Study 2: Supplementing CI-Skip rules

From Empirical Study 1, we found that CI-Skip rules are not 100% safe, especially those that produce higher cost-savings (NonSrcFileChange, AllPassingTest and NoReachableTest). In this next study, we aim to improve the trade-off of cost-saving and safety provided by CI-Skip rules. For that goal, we provide a collection of CI-Run rules that could complement CI-Skip rules to make them safer. CI-Run rules capture characteristics of builds that would intuitively signal that the build may fail, even when a CI-Skip rule applies. We then studied what ratio of the failing builds under each CI-Skip rule are covered by these CI-Run rules, and how strongly they discriminate between failing and passing builds under each CI-Skip rule.

### *5.1. Studied Factors: CI-Run rules*

We designed four CI-Run rules that we believed could flag builds that fail under a CI-Skip rule, based on our experiences. We thought about possible causes for builds to fail that developers may not expect, *i.e.,* that may cause failures even under the conditions described by CI-Skip rules. We also consulted the research literature that characterizes failing builds, looking for those that could still apply under CI-Skip rules. We list our proposed CI-Run rules in Table 2.

**BuildScripts (BS)**: We realized that the NonSrcFileChange (NSF) CI-Skip rule included build scripts. However, we believed that changes in build scripts may still cause failures, such as when dependencies change. For example, when a build depends on new modules (*e.g.,* because they migrated from Python 3.7 to 3.8), some functions may not work any more or may raise warnings because they are not supported (*e.g.,* the importlib load module() is abandoned in python 3.10). Furthermore, we also found that previous work also reported that changes in build scripts could cause build failures [36]. This rule is triggered when a build changes a build-script file (*e.g.,* "pom.xml" or "build.gradle").

Published version available at: https://doi.org/10.1016/j.jss.2022.111292

**ConfigurationFiles (CF)**: In our experience, another source of unexpected failures could be when changes happen in the configuration file for the CI engine. These changes would also be captured by the NonSrcFileChange (NSF) CI-Skip rule. We thought that such changes could cause failures, for example, when the script command is mistakenly input with a wrong flag and fails. This rule is triggered when a build changes the configuration file for the CI engine (*i.e.,* "travis.yml").

**SubsequentFailures (SF)**: We also thought that, even if a build falls under a CI-Skip rule, it could still fail if the source code is already broken — if a previous defect was not correctly fixed. Builds under some CI-Skip rules, *e.g.,* NonSrcFileChange (NSF), are less likely to break the build, but for the same reason they are also less likely to fix it if it was broken in the previous build. Previous work also reported that the subsequent build to a failing build is also likely to fail [3]. This rule is triggered when a failing build preceded the current build.

**IncreasingPlatforms (IPN)**: Another situation which we could envision builds failing even under CI-Skip rules is when the software will be tested in a new platform. A build can have multiple jobs, and each job is deployed and tested in different platforms. Even when no other changes happen in source code, bringing a new platform may cause new defects to emerge. This rule is triggered when the number of platforms for a build increases.

### *5.2. RQ3: What proportion of failing builds under CI-Skip rules are covered by our CI-Run rules?*

Our proposed CI-Run rules will be most effective in making CI-Skip rules safer if they cover a large proportion of the builds that failed under the rules. Thus, we measure the distribution of failing builds that fall under each possible combination of CI-Run rules for each CI-Skip rule. For example, a failing build that only contains SubsequentFailures falls into a different category from the failing build that satisfies both SubsequentFailures and ConfigurationFiles. Figure 3 shows the distribution of any combination of CI-Run rules present in failing builds under CI-Skip rules for any studied project.

#### *5.2.1. Result*

In Figure 3, we can observe that **most of failing builds under CI-Skip rules are captured by these four CI-Run rules**. In particular, 97% of VersionRelease failing builds can be captured by CI-Run rules.

Among these four CI-Run rules, we can observe that SubsequentFailures is the dominant factor for making builds fail under CI-Skip rules. At least 64% of failing builds under each CI-Skip rule can be explained by one combination including SubsequentFailures. This is because builds with seemingly-safe changes normally do fix an already-present defect, so the build continues to fail. For CI-Skip rules SourceCommentChange, SourceFormatModification and SourceFormatCommentChange, as they only exist for changes on source files, they cannot be captured by BuildScripts and ConfigurationFiles by definition. The most present CI-Run rule for these 3 rules was SubsequentFailures.

For NonSrcFileChange, the CI-Run rule of BuildScripts occupies the second largest population (37%), while ConfigurationFiles and IncreasingPlatforms take 11% and 2% respectively, which means build scripts and configuration files as non_source files can also make the build fail. SubsequentFailures and ConfigurationFiles take the same highest proportion (68%) of MetaFileChangeOnly failing builds, while the combination of SubsequentFailures+ConfigurationFiles is the most popular (44%). This is because ConfigurationFiles is a major component of meta files. IncreasingPlatforms also covers 11% of MetaFileChangeOnly failing builds. This shows that changes on meta files sometimes also include increasing platform numbers. BuildScripts captures 92% of VersionRelease failing builds, showing that most of VersionRelease builds modify build scripts. AllPassingTest and NoReachableTest have similar composition. NoReachableTest has a higher proportion of BuildScripts (19%) and ConfigurationFiles (6%) than AllPassingTest does (BuildScripts 15%, ConfigurationFiles 4%) because changes on these non_source files have no reachable test intuitively.

Finally, we also investigated some of those cases where build failures were not covered by the CI-Run rules — we labeled them as "Other" in Figure 3. In our investigation, we learned that these builds failed for multiple varied reasons, in addition to those described in CI-Run rules. However, we did not find any of these reasons appearing more than a handful of times — *i.e.,* they likely would not be generalizable. For example, a few failing builds under CI-Skip rules (8 out of 5,684) failed because of broken `links` in JavaDoc comments, which can cause a build to fail — *e.g.,* a build under the SourceCommentChange rule only contained changes in JavaDoc, but it changed a `link` to an incorrectly-named code entity, which broke the build (abdeldahak/jackson-core: 8a6a899). Also, a few other failing builds under CI-Skip rules (7 out of 5,684) failed because they used custom names for build scripts (*e.g.,* with no file extension). So, these builds were captured by the NonSrcFileChange CI-Skip rule, but in truth the build process had been modified, and failed (rspec/rspec-mocks:7f0828a).

### *5.3. RQ4: How helpful are CI-Run rules at discriminating between failing and passing builds under CI-Skip rules?*

Some CI-Run rules are dominant in failing builds under specific CI-Skip rules, but their popularity may be simply because they are widespread in builds under this rule. That is, they still may not discriminate between passing and failing builds among those captured by a CI-Skip rule. To learn that, we did an experiment to calculate the correlation between the presence of each CI-Run rule and the ratio of builds that failed under each CI-Skip rule.

Published version available at: https://doi.org/10.1016/j.jss.2022.111292

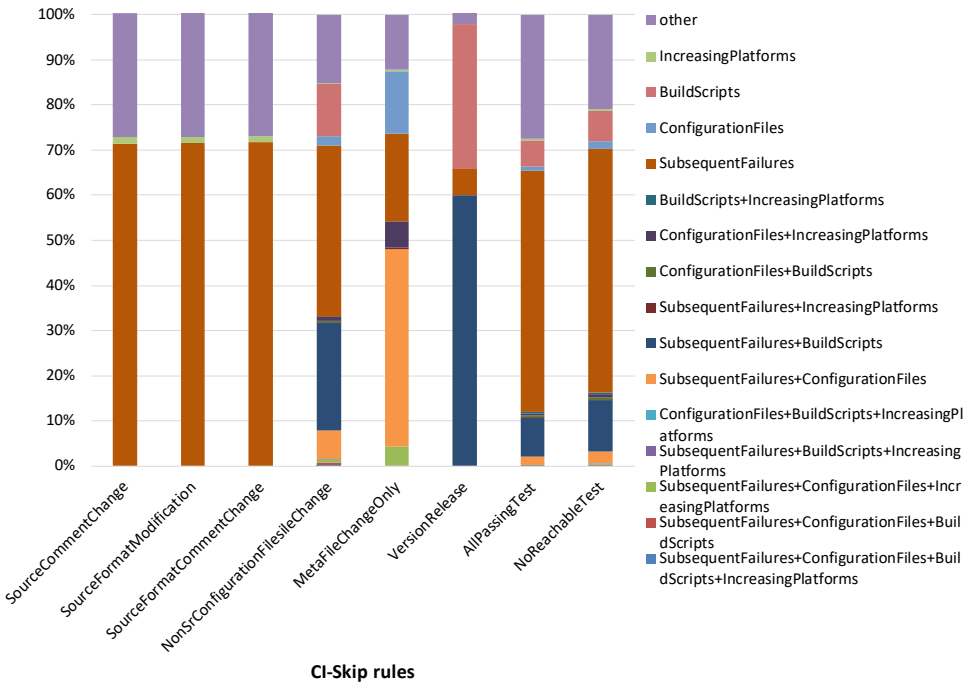


Figure 3: Distribution of failing builds captured by CI-Run rules under each CI-Skip rule.

We divided builds under each CI-Skip rule into four groups: with each CI-Run rule pass, with each CI-Run rule fail, without each CI-Run rule pass and without each CI-Run rule fail. For example, if we want to study the correlation between failing builds under SourceCommentChange and BuildScripts, we firstly divide all SourceCommentChange builds from all projects into four groups: with BuildScripts passing builds, without BuildScripts passing builds, with BuildScripts failing builds and without BuildScripts failing builds. We calculate correlation as the effect size using Cramer's V, which is designed for measuring the association between nominal variables. We then test for statistical significance using the Chi Square test. The sample size in this experiment for each CI-Skip rule was SourceCommentChange: 1035, SourceFormatModification: 1264, SourceFormatCommentChange: 229, NonSrcFileChange: 13103, MetaFileChangeOnly: 1329, VersionRelease: 2889, AllPassingTest: 73465, NoReachableTest: 34578.

*5.3.1. Result*

We report in Table 3 the results of this experiment. We leave cells blank when the correlation between a CI-Run rule under a CI-Skip rule and failing builds was not statistically significant (p_value >= 0.05). We found that **SubsequentFailures had a strong correlation with failing builds under every CI-Skip rule**. SubsequentFailures was also the only correlated CI-Run rule with failing builds under SourceCommentChange, SourceFormatModification, SourceFormatCommentChange and VersionRelease. With the results in RQ3 and RQ4, we can conclude that SubsequentFailures is the major CI-Run rule that makes CI-Skip rules unsafe. This reflects that those changes are mostly safe, but cannot fix the broken build.

Under other CI-Skip rules, failing builds also have a correlation with BuildScripts, ConfigurationFiles and IncreasingPlatforms. Among them, failing builds under NoReachableTest and NonSrcFileChange have correlations with all CI-Run rules, and failing builds under MetaFileChangeOnly and AllPassingTest have correlations with three CI-Run rules: ConfigurationFiles, SubsequentFailures, and IncreasingPlatforms. This shows that changes on build scripts and configuration files can also make some rules unsafe. Among them, changes on configuration files have a stronger correlation than changes on build scripts. This indicates that changes on project configuration files can be more risky. Though the effect size is small, we think they are still effective because most of the builds under each rule are passing builds and these correlated CI-Run rules can help predict failing builds. We also note that IncreasingPlatforms was a correlated CI-Run rule even if it was less popular in RQ3. This shows that projects rarely increased the platform set where they build, but when they did, it correlated with builds failing under NonSrcFileChange, MetaFileChangeOnly, AllPassingTest, and NoReachableTest. This findings can be used to warn developers when the program is going to be tested on more platforms.

## 6. Our Approach: PreciseBuildSkip

In our empirical studies, we observed that CI-Skip rules have a reasonable potential for cost savings (RQ1) and are relatively safe (RQ2), although not 100% so. We also identified CI-Run rules that capture the majority of failing builds under CI-Skip rules (RQ3), and identified how

Published version available at: https://doi.org/10.1016/j.jss.2022.111292

Table 3: Correlation between CI-Run rules and failing builds under CI-Skip rules.

| | | CI-Run Rules | | | |
|---|---|---|---|---|---|
| | | BuildScripts | ConfigurationFiles | SubsequentFailures | IncreasingPlatforms |
| CI-Skip Rules | SourceCommentChange | | | 0.83 | |
| | SourceFormatModification | | | 0.83 | |
| | SourceFormatCommentChange | | | 0.82 | |
| | NonSrcFileChange | 0.06 | 0.07 | 0.83 | 0.05 |
| | MetaFileChangeOnly | | 0.11 | 0.82 | 0.07 |
| | VersionRelease | | | 0.8 | |
| | AllPassingTest | | 0.03 | 0.79 | 0.04 |
| | NoReachableTest | 0.02 | 0.04 | 0.82 | 0.05 |

strongly they can discriminate between builds that will pass and builds that will fail (RQ4). These observations show that practitioners could manually use CI-Skip rules to save cost, but not 100% safely, even when they also apply our CI-Run rules.

Thus, we also created PreciseBuildSkip, a novel technique to allow practitioners to automatically predict which builds to skip, while maximizing the number of build failures that are observed (*i.e.,* not skipped). PreciseBuildSkip takes advantage of both CI-Skip rules (except AllPassingTest) and CI-Run rules as features. Our intuition is that by training PreciseBuildSkip with CI-Skip rules and CI-Run rules, its predictions will be highly safe, *i.e.,* it will prefer to err executing passing builds than to erring skipping failing ones. We train it as a cross-project predictor (*i.e.,* we train PreciseBuildSkip in the past builds of different software projects than the one in which we apply it). This helps with the cold-start problem [61] in software projects for which only a few builds have been executed and thus they need additional data for training [62]. PreciseBuildSkip then predicts the outcome of each build and only executes those that it predicts to fail. Finally, we make PreciseBuildSkip customizable, *i.e.,* we can customize its prediction sensitivity to varying levels of tolerance to skipping failing builds.

## 7. Experiment 1: Evaluating PreciseBuildSkip

We evaluate PreciseBuildSkip in three ways. First (RQ5), we evaluate the correctness of its predictions, using the traditional metrics for prediction tasks: precision, recall, F1, and AUC. Then (RQ6), we evaluate the impact that PreciseBuildSkip's predictions provide to developers in more practical terms — how much cost they allow them to save, and how many build failures they allow them to observe. Finally (RQ7), we evaluate how much build time PreciseBuildSkip allows developers to save, when we account for the overhead of executing it.

RQ5 teaches us the quality of PreciseBuildSkip's predictions — irrespective of its context of usage, and RQ6 teaches us the benefit and drawback that developers can obtain from them in more practical terms — cost saving and failure observation. Then, RQ7 teaches us the extent to which the cost (execution time) of running PreciseBuildSkip threatens the cost (execution time) it saves.

### *7.1. Research Method*

We describe the details of our evaluation below.

#### *7.1.1. Studied Techniques*

We evaluated PreciseBuildSkip in two versions. First, our proposal, PreciseBuildSkip, using a random-forest classifier (§6). Second, PBS_RB, as a rule-based variant of PreciseBuildSkip, to represent the cost-saving and safety that a developer would observe when manually using our set of CI-Skip rules (except AllPassingTest) and CI-Run rules. We also replicated all existing build-selection techniques for our evaluation.

**PreciseBuildSkip (PBS):** Our proposed approach (see §6). Since it is customizable, we evaluate it for multiple prediction-sensitivity thresholds: 0–0.1 (101 data points in this range). This is the range of thresholds for which we observed PBS provide a range of different levels of cost savings. Higher prediction sensitivities make PBS more likely to predict builds to pass. This will let us observe the multiple trade-offs that it could achieve in terms of cost saving and safety.

**PBS_RB:** A rule-based approach including all CI-Skip rules (except AllPassingTest) and their corresponding CI-Run rules. This variant goes through our list of CI-Skip rules (except AllPassingTest), and skips builds under them when none of their correlated CI-Run rules are present.

**Jin20 [3]:** A 2-phase build selection approach, using a random-forest classifier with size features. Since Jin20 is a customizable approach that can be set to varying prediction sensitivities, we replicated its most conservative (safest) configuration, as described in its original paper. This means that we configured its predictor to have a prediction sensitivity of 0, which causes it to have a strong preference to predict build failures — the predictor will predict builds to fail, unless it is 100% confident that it will pass.

**Abd19 [1]:** The first rule-based build selection approach based on CI-Skip rules, which uses a subset of

Published version available at: https://doi.org/10.1016/j.jss.2022.111292

our studied CI-Skip rules (SourceCommentChange, SourceFormatCommentChange, NonSrcFileChange, MetaFileChangeOnly and VersionRelease). We replicated Abd19 by using the data in TravisTorrent for the number of source files changed. For other rules, we downloaded each software project locally, used Python (lib git.Repo) to request commit messages, changed files, and changed lines for each commit. Then, after each rule, was ready, we ran the simulation to skip one build whose all commits follow at least one rule.

**Abd20 [21]:** A machine-learning approach (also random-forest classifier) using Abd19's CI-Skip rules as features. We replicated Abd20 following the same process of replicating Abd19: we git cloned the project and requested the commit information using Python. Since Abd20 requires more rules, we implemented additional steps to replicate it. For example: I mined the author names and commit time, for the rule that considers recent expertise.

#### *7.1.2. Training and Testing*

We used the same data set as Empirical Study 1 and 2, which includes 82,427 builds from 100 projects (see §3.1) We use 10-fold cross validation (each fold has 10 projects) to evaluate machine-learning-based techniques: PreciseBuildSkip and Jin20. Each build in the testing fold is tested by a classifier trained on the other 90 projects. Abd20, however, can not be trained in our dataset. Abd20 trains its classifier with developer-skipped commits, and our dataset has too few of these commits. So, we trained Abd20 in the 10-project dataset in which it was originally evaluated [21], and tested it in ours (see §3.1). Rule-based techniques (PBS_RB and Abd19) do not require training. So, we applied them directly to our dataset.

As in past work [3], we simulated a realistic scenario in which the outcomes of builds that are skipped are not available for coming predictions. That is, we only update the information connected to the last build, *e.g.,* SubsequentFailures, when it was actually executed (not when it was skipped). When a predictor predicts the upcoming build as a pass, we skip the build and accumulate the value of its size factors (such as number of changed source files) for the next build, also as past work did [3].

#### *7.1.3. Metrics*

We measured three sets of metrics, one for each research question in this experiment.

**RQ5:** To measure the correctness of PreciseBuildSkip's predictions, we used four metrics.

*Precision:* the number of correctly predicted build failures, divided by the number of builds that the technique predicted as build failures. We expect PreciseBuildSkip to provide low precision (by design), since it aims to maximize the observation of build failures.

*Recall:* the number of correctly predicted build failures, divided by the number of actual build failures. For the same reason, we expect PreciseBuildSkip to provide high recall.

*F1 score:* the harmonic mean between precision and recall. We expect PreciseBuildSkip to provide low F1 score, since we expect it to provide low precision.

*AUC:* the Area Under the ROC (Receiver Operating Characteristic) Curve. We expect PreciseBuildSkip to provide low AUC score, since we expect it to provide low precision.

**RQ6:** To understand how much PreciseBuildSkip could benefit developers, we measured three metrics: *Cost Saving*, *Observed Failures* and *Skipped Failure Relative Density (SFRD)*. The first metric was included in all prior works [3, 21, 1], and the second metric was covered in an existing work [3]. The last metric is designed in this paper to measure how strongly a technique targeted skipping passing builds.

*Cost Saving.* To measure how much of the computational cost of CI a technique reduced, we measure the proportion of builds that it skipped, among all builds. By this metric, a technique that skipped (*i.e.,* avoided the computational cost of executing) a high proportion of builds highly reduced the computational cost of CI.

$$CostSaving = \frac{\#\ skipped\ builds}{\#\ all\ builds}$$

*Observed Failures* is the proportion of failing builds that are correctly predicted and not skipped, among all failing builds. It measures a technique's ability of detecting failing builds. A technique performs better in this metric if it catches more failing builds.

$$ObservedFailures = 1 - \frac{\#\ skipped\ failing\ builds}{\#\ all\ failing\ builds}$$

We also designed the *Skipped Failure Relative Density (SFRD)* metric. It measures the fail ratio in skipped builds divided by the fail ratio in all builds. This metric allows us to understand how strongly one technique can discriminate passing and failing builds. A lower value in this metric indicates a better performance. A technique performs better in this metric if it skips builds with a lower fail ratio than the original fail ratio of all builds. This metric has two values with special meanings. The metric value of 1 means that a technique achieved roughly the same trade-off as skipping builds randomly. The metric value of 0 means that a technique observes all failing builds.

$$SFRD = \frac{fail\ ratio\ of\ skipped\ builds}{fail\ ratio\ of\ all\ builds}$$

**RQ7:** To understand how much the overhead of running PreciseBuildSkip impacts its provided cost savings, we measured one metric.

*Saved Build Duration:* This metric gives us different information than our earlier "Cost Saving" metric in RQ6, since it accounts for the time that it takes to run builds in CI. This metric measures the proportion of build-execution

Published version available at: https://doi.org/10.1016/j.jss.2022.111292

time that a technique skipped, among all build-execution time — *i.e.,* the cumulative execution time of all the builds that a technique skipped, divided by the cumulative execution time of all builds (skipped or not).

We compare the *Saved Build Duration* including PRECISEBUILDSKIP's execution time and excluding it — to understand its overhead. We measured PRECISEBUILDSKIP's execution time by including the time for running its feature techniques and its own prediction time.

### *7.2. Results for RQ5: How correct are PRECISEBUILDSKIP's predictions?*

Figure 4 shows the results for this research question. This figure shows the median value for each metric across studied projects. The Y axis represents the metric for evaluation and the X axis is the prediction sensitivity for PRECISEBUILDSKIP.

To be able to compare PBS with existing techniques, we highlight four prediction thresholds of interest for it (see Figure 4-Recall). *Safe*: the highest threshold that provides 100% recall. *Moderate*: the threshold that provides the closest recall to Abd20's. *Relaxed*: the threshold that provides the closest recall to Jin20's. *More relaxed*: the threshold that provides the closest recall to Abd19's. We also highlight PBS's scores for the same prediction thresholds for the remaining metrics in Figure 4.

In Figure 4-Precision, we can observe that almost all techniques have low (and very similar) precision scores (lower than 0.1). This is by design, since all these techniques are designed to be highly safe, *i.e.,* they are conservative and predict many builds to fail. The exception is PRECISEBUILDSKIP, which obtains higher precision scores as we configure it with higher prediction thresholds.

The counterpart to precision is recall. Figure 4-Recall shows that most techniques obtain high recall — by design, *i.e.,* for the same reason that they obtain low precision. However, the range of their recall scores is more varied than their precision scores, allowing us to differentiate among them more clearly. In terms of recall, the best-performing technique was PBS, achieving 100% recall for its *Safe* threshold — and keeping a precision score that is similar to all other techniques'. As we increased its prediction threshold, PBS's precision increases and its recall decreases.

In terms of F1 score (see Figure 4-F1 score), most techniques achieve low values, as a result of their low precision. We observe a similar effect for AUC scores in Figure 4-AUC.

In summary, all studied techniques achieved very close precision scores, but they differentiated themselves in terms of recall — for which PBS obtained the highest score.

### *7.3. Results for RQ6: How much cost-saving and safety do PRECISEBUILDSKIP's predictions provide?*

We plot the results for this research question in Figure 5. This figure shows the median value for each metric across

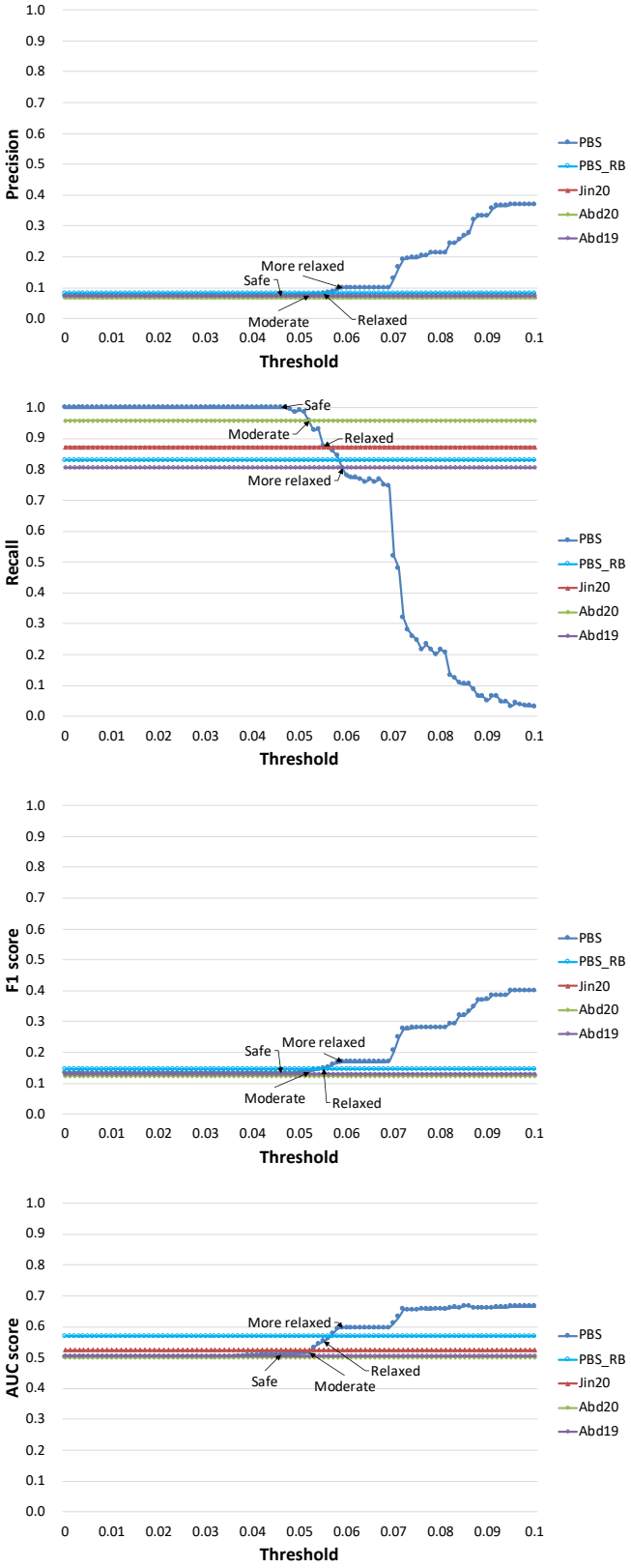


Figure 4: Performance comparison on predicting build failures.

Published version available at: https://doi.org/10.1016/j.jss.2022.111292

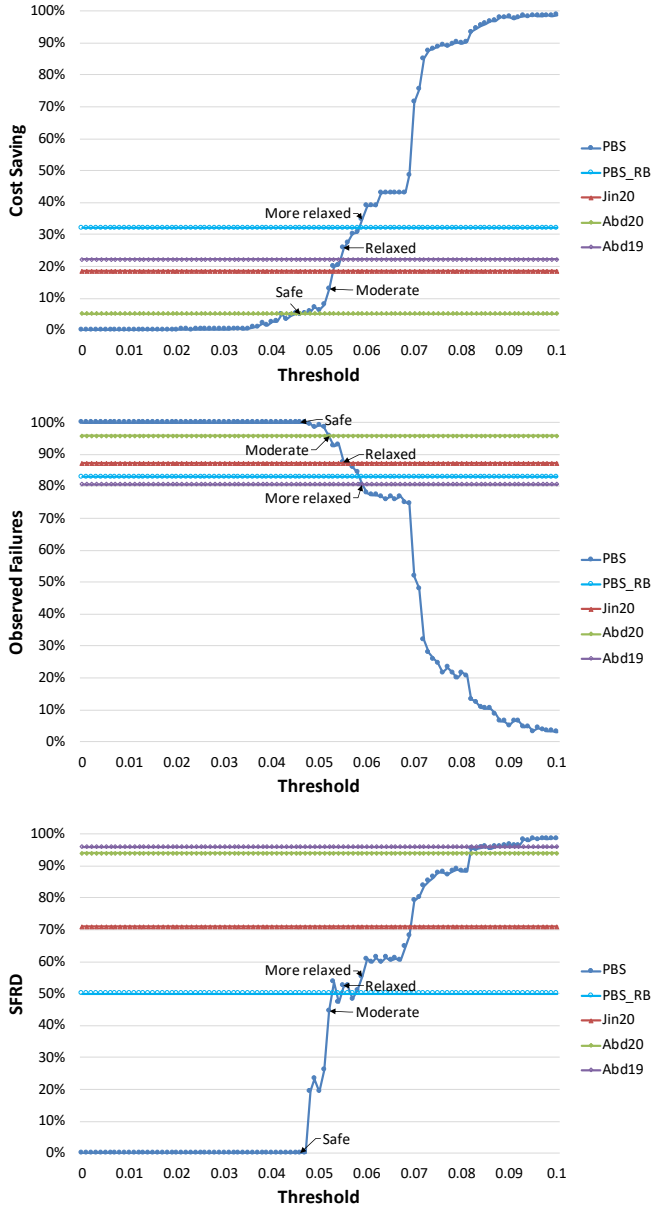


Figure 5: Cost saved and value kept by evaluated techniques.

studied projects. The Y axis represents the metric for evaluation and the X axis is the prediction sensitivity for PreciseBuildSkip.

We first make observations from comparisons among existing techniques. We can observe that Abd19 is the existing technique that achieves highest *Cost Saving*. Abd19 is able to save 22.3% builds while Jin20 and Abd20 save 18.6% and 5.2% respectively. Abd20 is the safest technique that observes most failing builds among existing approaches. It can observe 96% of build failures while Jin20 observes 87% and Abd19 observes 81% failing builds. We can also observe that Jin20 performs best *SFRD*. *SFRD* of Jin20 is 0.71 in while Abd20 and Abd19 achieves 0.94 and 0.96 respectively. From these observations, we can find that each exiting approach has its own strengths and none of them can be really safe.

We also make a few observations about how PBS performs across prediction thresholds. First, PreciseBuildSkip shows little impact when the threshold is smaller than 0.014, which means it observes all failing builds by seldom skipping builds. Then along with the increasing of the threshold, *i.e.,* making the predictor less sensitive to the build failures, *Cost Saving* increases and *Observed Failures* drops. However, *Observed Failures* starts dropping later than *Cost Saving*'s increasing, *i.e., SFRD* remains at 0, which means in this range PreciseBuildSkip is able to observe all build failures and save some cost. In the most optimized scenario, our approach can save 5.5% of builds and observe all build failures. After that, *Cost Saving* gets continuously increasing and *Observed Failures* gets decreasing correspondingly until they come to the ending scenario where all builds are skipped and no failing builds is observed, making *SFRD* reach 1.

Next, we compare PreciseBuildSkip with existing techniques by highlighting the same prediction thresholds discussed for RQ5 (see §7.2). **First**, we observe that PBS is able to achieve 5.5% *Cost Saving* while keeping 100% *Observed Failures* in a **safe** mode (threshold 0.047). This shows that PBS can observe more build failures than safest existing work (Abd20) did and provides slightly more cost-saving meanwhile. PBS also achieves the best *SFRD* as a value of 0 at this point. **Second**, in a **moderate** scenario (threshold 0.052), PreciseBuildSkip can save 12.9% *Cost Saving* and keep 96% *Observed Failures* compared with Abd20. This shows that PBS can observe same amount of failing builds as Abd20 (96%) but increases *Cost Saving* from 5.2% to 12.9%. Also, PreciseBuildSkip performs better at *SFRD* at this point (0.45 vs. 0.94). **Third**, in a **relaxed** scenario (threshold 0.055), PreciseBuildSkip can save 25.8% cost and observe 87.6% failing builds at the same time. Compared with the existing technique that is best at cost-saving (Abd19), PBS can observe more failing builds (Abd19 81%) and more *Cost Saving* as well (Abd19 22.3%). Besides, PBS at this point also achieves a smaller value of *SFRD* (0.52) than Abd19 does (0.96). **Fourth**, in a **more relaxed** scenario (threshold 0.059), PreciseBuildSkip can save 34.8% cost and observe 81% failing builds at the same time. Compared with Abd19, PBS can observe same amount of failing builds (81%) but increases *Cost Saving* from 22.3% to 34.8%. Besides, PBS also achieves a lower *SFRD* (0.55). We lastly find that all variants we point out above have a better performance on *SFRD* than all existing techniques.

Finally, we observed that PBS_RB works well as a rule-based technique that is easy to use and requires no training data. It achieves the *SFRD* of 0.5 which is better than all existing techniques. The performance of PBS_RB is very similar to the performance of PBS at threshold 0.059 and it can save 32% of cost saving while observing 83% of build failures.

Published version available at: https://doi.org/10.1016/j.jss.2022.111292

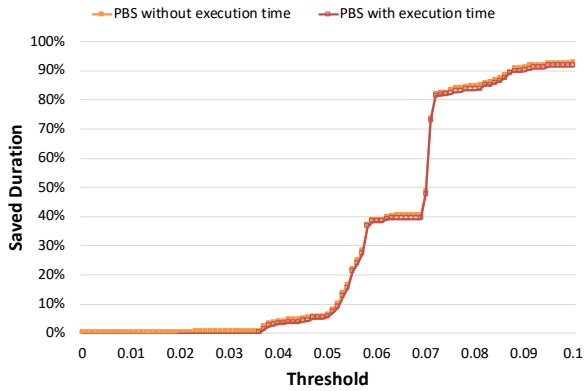


Figure 6: Build time saved by PreciseBuildSkip including and excluding its execution time.

### *7.4. Results for RQ7: How much overhead does PreciseBuildSkip add to build duration?*

We answered this question by comparing the build time saved by PreciseBuildSkip, with and without accounting for its execution time. We plot the results for this research question in Figure 6. This figure shows the median value for each metric across studied projects. The Y axis represents the metric for evaluation and the X axis is the prediction sensitivity for PreciseBuildSkip.

We see in Figure 6 that accounting for PreciseBuildSkip's execution time has negligible impact on the cost that it saves in terms of build duration. PreciseBuildSkip's execution duration generally takes only 0.5% of the saved build duration, *e.g.,* PreciseBuildSkip saved 5.5% of build duration at threshold 0.047, 5.1% after we deduct its execution time. Furthermore, as PreciseBuildSkip's prediction threshold increases, its overhead decreases, *i.e.,* its execution time becomes a smaller and smaller proportion of its build time saved as it saves more and more build time.

## 8. Experiment 2: Evaluating the impact of CI-Run rules in PreciseBuildSkip

In this experiment, we evaluate the impact of CI-Run rules on our approach (among specific variants pointed in §7.3). We aim to understand how our approach performs with and without CI-Run rules in term of saved cost and kept value.

### *8.1. Research Method*

We use the same data set and simulation process as Experiment 1. We also use the same three measurement metrics (*Cost Saving*, *Observed Failures* and *SFRD*) that the variants would provide in practice for evaluation.

#### *8.1.1. Studied PreciseBuildSkip (PBS) variants*

We evaluate PreciseBuildSkip with other variants of it, including rule-based variants and variants without CI-Run rules.

**PBS_Safe:** The safe variant of our original approach (threshold 0.047), keeping all build failures observed and saving as much cost as possible, same as the first point in §7.3.

**PBS_IC_Safe:** The safe variant (threshold 0.135) of incomplete version of our approach using only CI-Skip rules (except AllPassingTest) as features, saving similar amount of cost as PBS_Safe.

**PBS_Moderate:** The moderate variant of our original PBS (threshold 0.052), observing as many failing builds as Abd20 did, same as the second point in §7.3.

**PBS_IC_Moderate:** The moderate variant (threshold 0.16) of incomplete version of our approach using only CI-Skip rules (except AllPassingTest) as features, saving similar amount of cost as PBS_Moderate.

**PBS_Relaxed:** The relaxed variant of our original approach (threshold 0.055), observing as many failing builds as Abd19 did, same as the third point in §7.3.

**PBS_IC_Relaxed:** The more relaxed variant (threshold 0.163) of incomplete version of our approach using only CI-Skip rules (except AllPassingTest) as features, skipping similar amount of builds as PBS_Relaxed.

**PBS_More_Relaxed:** The more relaxed variant of our original PBS (threshold 0.059), saving similar amount of cost as Abd19 did, same as the third point in the result of RQ5.

**PBS_IC_More_Relaxed:** The more relaxed variant (threshold 0.164) of incomplete version of our approach using only CI-Skip rules (except AllPassingTest) as features, skipping similar amount of builds as PBS_More_Relaxed.

**PBS_RB:** Our rule-based approach included in Experiment 1.

**PBS_RB_IC:** The incomplete version (no CI-Run rules) of PBS_RB, skipping rules when any of the CI-Skip rules is fulfilled.

### *8.2. Results for RQ8: What is the impact of including CI-Run rules as features in PreciseBuildSkip?*

We plot the results of this experiment in Figure 7 for the evaluation of all variants. The boxes in these box plots for each metric represent its distribution of values for all the studied projects. We discuss our observed differences in results in terms of absolute percentage point differences over the median value of each metric across projects. All differences in this result are statistically significant ($p\_value < 0.01$).

We can observe that the variants of PBS_IC in general observe less build failures and save similar or less amount of cost as their corresponding PBS variants. Among them, PBS_IC_Safe can observe 97.2% failing builds and it can save 5.1% of cost saving, which means it performs worse in both metrics than its corresponding PBS variant (5.5%, 100%). Compared with PBS_Moderate (12.9% *Cost Saving* and 96% *Observed Failures*), PBS_IC_Moderate performs worse in both metrics (11.6% and 92.4%). Also,we

Published version available at: https://doi.org/10.1016/j.jss.2022.111292

compare PBS_IC_Relaxed and PBS_Relaxed and find that the former's *Cost Saving* (20.3%) is lower than the latter's (25.8%) and *Observed Failures* is lower as well (85.4% vs. 87.6%). Besides, PBS_IC_More_Relaxed achieves 23.8% in *Cost Saving* and 82% in *Observed Failures* while PBS_More_Relaxed achieves 34.8% and 81% respectively.

We then make observations on *SFRD*. We can find that all variants of PBS_IC has higher *SFRD* than their corresponding PBS variants. Since one approach has a better ability to distinguish failing and passing builds if it has a lower value of *SFRD*, we can conclude that PBS variants can discriminate failing builds more accurately than PBS_IC variants. Among them, PBS_IC_Safe also has a value of 0.74 *SFRD* which is worse than PBS_Safe. Besides, PBS_IC_Moderate and PBS_IC_Relaxed also have a higher *SFRD* (0.93 and 0.96) compared to their corresponding variants (0.45 and 0.55). Finally, PBS_IC_More_Relaxed's *SFRD*'s value reaches 1 which means it performs same as randomly pick.

Therefore, given that the variants of PBS_IC have lower values of *Observed Failures* with similar values of *Cost Saving* and higher values of *SFRD* than the corresponding variants of PBS, we can reach a conclusion that CI-Run rules are able to complement CI-Skip rules and supplement our approach to better discriminate failing and passing builds.

Finally, we make observations to compare PBS_RB and its corresponding technique, PBS_RB_IC. We can find that PBS_RB_IC has a higher value in *Cost Saving* and a lower value in *Observed Failures* (47% and 46%), giving it a high value of 1 in *SFRD*. This shows that CI-Run rules are also essential when be applied our rule_based techniques by complementing CI-Skip rules to better discriminate failing and passing builds.

## 9. Experiment 3: Evaluating PreciseBuildSkip when trained on Builds affected by Build-selection

After build selection techniques have been used for some time, the available training data (build executions and their outcomes) would only contain *selected builds*, *i.e.,* only the builds that the build-selection technique decided to run. To understand the impact on PreciseBuildSkip's effectiveness of being trained on such *selected builds*, we performed this experiment.

### *9.1. Research Method*

We use the same research method as Experiment 2, except for the details below.

#### *9.1.1. Studied Techniques*

In this experiment, we study the following techniques: PBS_Safe, PBS_Moderate, PBS_Relaxed, PBS_More-relaxed, Abd20, and Jin20, as described in Experiment 2 (see §8.1.1). We omit two of the techniques that we studied

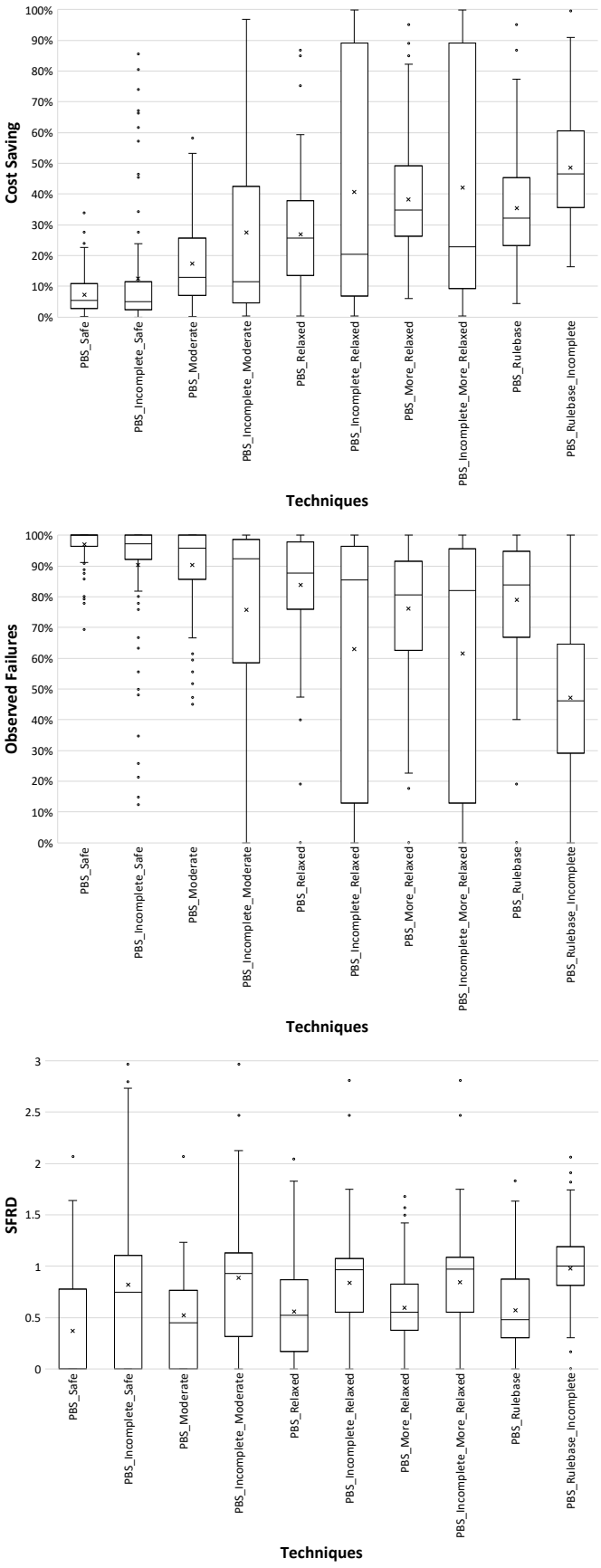


Figure 7: Cost saved and value kept by evaluated PreciseBuildSkip variants including and excluding CI-Run rules.

Published version available at: https://doi.org/10.1016/j.jss.2022.111292

in earlier experiments — PBS_RB and Abd19 — because they are not affected by training on selected builds — they are rule-based and thus do not use a training step.

*9.1.2. Training and Testing*

We followed a different training and testing process in this experiment.

First, for each studied technique, we simulated having applied it to the whole dataset. We did that by executing the technique for build selection over every build of every project, as described for Experiment 1 (see §7.1.2). We refer to the outcome of this step as the *selected-builds dataset* for that technique.

Then, for each technique, we simulated training it in projects that had already applied build selection. We achieved that by again applying the training-testing steps for Experiment 1 (see §7.1.2), but this time taking its training folds from its *selected-builds dataset* and the testing folds from the original dataset.

*9.2. Results for RQ9: How much cost-saving and safety does PreciseBuildSkip provide when trained on projects that use build selection?*

We plot the results for this research question in Figure 8. The boxes in these box plots for each metric represent its distribution of values for all the studied projects. The Y axis represents the metric for evaluation and the X axis is the studied technique variant. For ease of comparison, we represent side by side the results of each technique when trained on the original dataset — using the technique's original name — and when trained on its *selected-builds dataset* — adding to its name the *_Selected* suffix.

We see in Figure 8 that all techniques provided very similar results when trained on projects that used build selection than when they were trained on projects that did not. Thus, training them on data that had already been modified by their build selection had a negligible impact on their effectiveness.

We believe that this is because most techniques are generally conservative in skipping builds — they are more likely to decide to run a build than to skip it. As a result, the impact that they had when applied to produce the *selected-builds dataset* was limited enough to only negligibly impact their effectiveness when they used it for training.

In more detail, PBS_Safe_Selected obtained the same median Observed Failures (100%) and SFRD (0), but decreased its Cost Saving from 5.5% to 5.3%. PBS_Moderate_Selected had the same median Observed Failures (96%), but decreased its Cost Saving from 12.9% to 12.4%. PBS_Relaxed_Selected obtained the same median Observed Failures (87.6%), but increased its Cost Saving from 25.8% to 27.1%. PBS_More_Relaxed_Selected had less median Observed Failures from 81% to 80%, but its Cost Saving increased from 34.8% to 37.9%. Abd20_Selected had less median Observed Failures from 96% to 95%, but

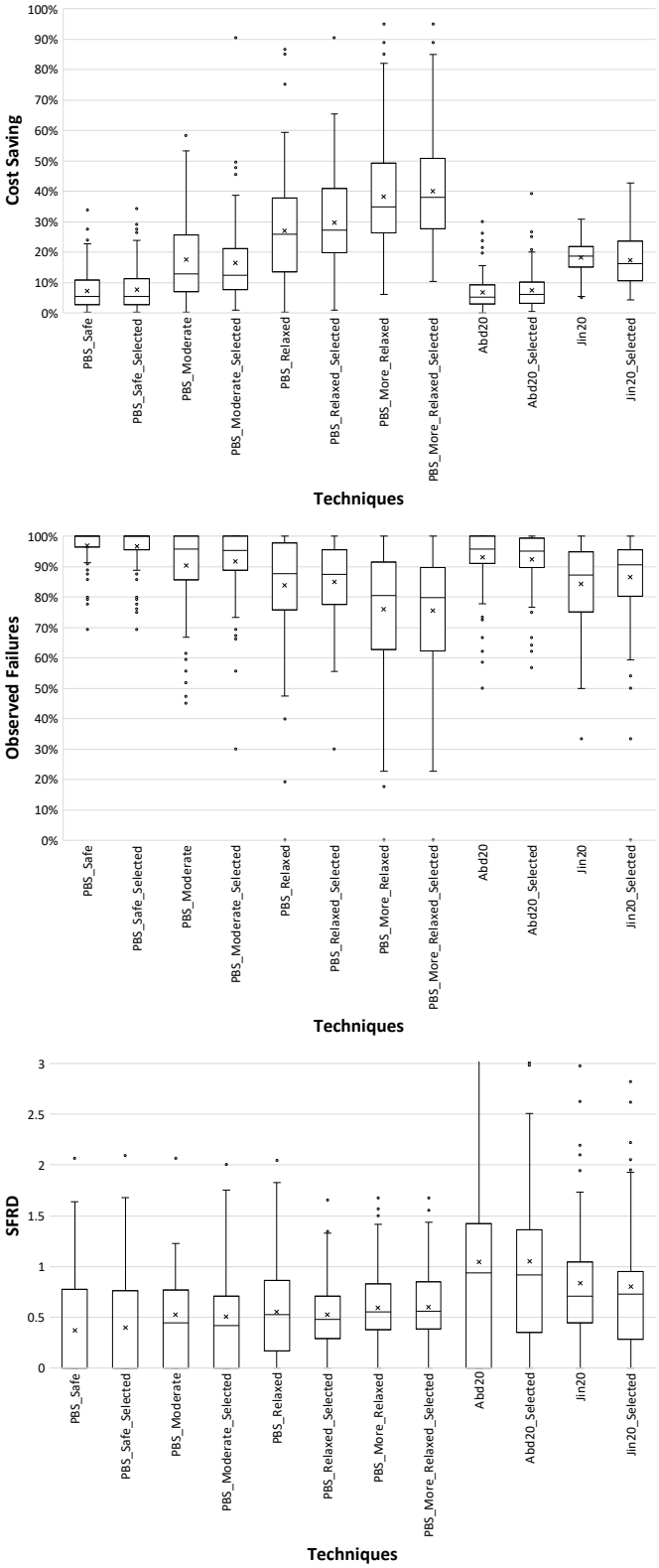


Figure 8: Cost saved and value kept by evaluated techniques when being trained under pre-selected data.

Published version available at: https://doi.org/10.1016/j.jss.2022.111292

its Cost Saving increased from 5.2% to 6%. Jin20_Selected obtained more median build failure observations from 87% to 90.5%, but its Cost Saving dropped from 18.6% to 16.3%.

## 10. Implications

### *10.1. For practitioners.*

From the findings of Empirical Study 1, we find that developers' intuitions may not always be correct, *i.e.,* skipping builds based on their favors may result in missing build failure observations. Therefore, developers should be more cautious when skipping builds by CI-Skip rules. Instead, developers may be able to refer to CI-Run rules and make their decisions based on both CI-Skip rules and CI-Run rules.

We believe that the largest barrier for adopting a CI build selection approach is that developers may be afraid of skipping failing builds. In other words, the concern of delaying failing build observation can be the main reason that build selection approach is not adopted. This implies the motivation for a build selection technique with no mispredictions. Thus, we propose PreciseBuildSkip as a precise technique that minimizes the observed build failures of build selection while providing some cost-savings at the same time.

In contrast, other developers may be looking for a way to reduce CI's high-cost barrier [7] to adopt it, even if it means observing build failures less quickly. PreciseBuildSkip provides configurations with a more liberal sensitivity for these developers: save the cost of 35% of their builds and still observe 81% failing builds with no delay (and the remaining 19% with a 1-build delay). Besides, when there is no training data available, developers can still get benefit from PreciseBuildSkip by using its rule-based version (PBS_RB). Furthermore, our novel metric, *SFRD*, is able to provide developers a chance to pick preferable build selection techniques in a more comprehensive way.

### *10.2. For researchers.*

From the result of RQ5 and RQ6, we can find that SubsequentFailures (subsequent failures) is the main CI-Run rule that makes CI-Skip rules invalid. This is because when the build has already been broken, the only way to turn it to pass is to fix the defect, rather than make any safe changes. Existing work [3][63] also found that the build is hard to transit status, *i.e.,* failing builds are likely to be followed by another build failure. This implies SubsequentFailures could be an important feature when detecting defects.

In this study, we tried different ways to take advantage of SubsequentFailures. We firstly used it as a feature for our predictor. However, the last build status is only available when the last build is executed. Therefore, when the predictor becomes less sensitive to the failing builds, *i.e.,* the threshold increases which means less failing builds are observed, SubsequentFailures is more often not updated in time and this makes the predictor harder to predict a failing build. That's why the predictor almost predicts every build to pass when the threshold is 0.1. If we take an alternative approach — execute the subsequent build of a failing build normally until we find a pass instead of triggering the predictor every time, the curve can be flattened. However, we will have less cost-saving, since we execute one passing build after a failing build anyway. As a result, we decided to use SubsequentFailures as a feature and let developers tune the technique in a tinier range.

Finally, we observe that if we keep most failure observations, the cost-saving remains low. This implies there is an opportunity for more CI-Skip rules coming out to contribute to cost-saving. For example, builds with changes on some specific subsystem of the source code is likely to be builds that can be safely skipped. Also, different projects may have different preferences on choosing CI-Skip rules and CI-Run rules, *e.g.,* faults caused by IncreasingPlatforms may be acceptable to some projects. Besides, since AllPassingTest works well in §4, there are other ways to approximate it, *e.g.,* test selection techniques [13] can predict the result of tests. If all tests are predicted to pass, then AllPassingTest is valid.

## 11. Threats to Validity

### *11.1. Construct Validity*

We use metrics as proxies to represent the *value* — observation of build failures — and *cost* — build execution — in CI. However, these are metrics that developers have reported as describing the value and cost of CI, *e.g.,* [4, 35, 2, 64], and are metrics that other existing approaches for saving cost in CI have used, *e.g.,* [13, 1]. We didn't include the metric designed in previous work [3] as the harmonic mean of *Cost Saving* and *Observed Failures* since this metric dislikes the scenario where *Observed Failures* is maximized (difference between it and *Cost Saving* is relatively big) which is the goal of this paper. Instead, we designed a new metric in this paper called *SFRD* to compare build selection techniques in a more comprehensive way. We also didn't include the metric of failing build delay [3], since the delay of skipped failing builds should be 1 by nature in our scenario because there is not enough space for long delay by relatively low cost saving.

Another threat to construct validity is whether developers want to skip passing builds to save cost in CI. Our work targets those developers that want to skip passing builds to reduce the cost of CI, but we recognize that not all developers may want this. Some developers may have different preferences and thus may prefer to use other techniques, *e.g.,* approaches that predict build outcome but do not skip any build [37, 35].

Reducing the cost of CI by skipping passing executions is a common strategy in the research literature. Many

Published version available at: https://doi.org/10.1016/j.jss.2022.111292

approaches have been proposed to skip the execution of passing tests [8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20] and passing builds [3]. Furthermore, when asked about the characteristics of the builds that they manually skip, developers for the most part describe builds that will likely pass, *i.e.,* they skip builds with: non-source code changes, with no test coverage, with trivial source code changes, or with other likely-to-pass characteristics, *i.e.,* refactoring changes [1].

Finally, developers may want to also skip other builds, in addition to passing builds (*e.g.,* builds that they know will fail [1]). In such situations, developers may still apply our proposed approach to skip passing builds, and combine it with other approaches to skip other kinds of builds.

### *11.2. Internal Validity*

To guard internal validity, we carefully tested our evaluation tools on subsets of our dataset while developing them. Our analysis could also be influenced by incorrect information in our analyzed dataset. Our results may be affected by flaky tests causing spurious failing builds — particularly when calculating the AllPassingTest and NoReachableTest CI-Skip rules. However, CI is expected to function even in the presence of flaky tests, since most companies do not consider it economically viable to remove them, *e.g.,* [13, 65]. Also, since flaky tests tend to be a minority of the wider population of tests (according to existing studies [8]) and since we did not include AllPassingTest as a feature for PreciseBuildSkip, we believe that flaky test had a limited influence on our experiments.

Another threat to internal validity comes from the fact that we defined CI-Run rules according to our personal experiences. So, there may be other CI-Run rules that could be useful in complementing CI-Skip rules, but that we did not report in this study. However, the CI-Run rules that we defined covered the large majority of failing builds under CI-Skip rules ($> 70\%$). Still, to better understand the extent of this threat, we further investigated the failing builds under CI-Skip rules for which none of our CI-Run rules applied, *i.e.,* the "Other" category. We found that these failing builds rarely followed common patterns, and when they did, they appeared in only a handful of instances, making it hard for them to generalize (see Section 5.2.1). In the light of these observations, it seems that defining additional CI-Run rules on top of the ones included in this project increasingly becomes an effort of capturing corner cases that are specific to a given software project or given development habits. Therefore, for those software projects that wanted to improve the safety of our approach even further by defining even more CI-Run rules, we recommend them to define additional CI-Rules that capture the corner cases that are specific to their software project and practices.

Our decision of defining NoReachableTest to consider only static dependencies may have slightly limited PreciseBuildSkip's results. Some static dependencies in the source code may never be triggered during its execution. This means that NoReachableTest could have some false positives, potentially making PreciseBuildSkip execute some builds with test cases that depend on the changes, but that will not trigger them during the software's execution. Therefore, the alternative of applying dynamic dependency analysis could have made PreciseBuildSkip save even more cost. However, such optimization decisions can have hidden costs [66]. Applying dynamic analysis is more computationally expensive, which could also cancel some of the cost savings provided by its higher accuracy.

The accuracy of the tools that we used to run static analysis (SciTools Understand [59] and rubrowser [60]) could also have impacted the ability of PreciseBuildSkip to save cost via the NoReachableTest CI-Skip rule. We used these tools to capture static dependencies between source code classes. SciTools Understand captured the following dependencies among classes: "calls", "implements", "includes/imports", "inherits", "inits", "modifies", "overrides", "sets", "throws", and "uses". Rubrowser captured all modules and classes definitions, and all constants that are listed inside a module/class and linked them together. Then, we used the resulting graph of classes and dependencies among them from these tools to determine if the changed files were reachable by the a project's test cases. A possible threat to validity is that bugs in the source code of these tools may make them miss some dependencies (false negatives), or capture some dependencies that do not really exist (false positives), which could affect the accuracy with which PreciseBuildSkip applied NoReachableTest to skip builds. Also, since both SciTools Understand [59] and rubrowser [60] capture static dependencies, they both can have false positives (capturing dependencies that will not be executed in runtime). Such false positives would cause PreciseBuildSkip save less cost than it could, if we had used dynamic analysis tools.

Another threat could be the risk of over-fitting in our empirical study 2 (§5), since we performed it over our complete data set — since we aimed to increase the generalizability of our observed correlated features. To address the over-fitting risk, we repeated our study on the chronologically earlier half of data for our features through stratified random sampling [67] on number of builds. The selected features remained the same. Furthermore, PreciseBuildSkip is a cross-project predictor that is not affected by the threat of violating the chronological order of builds that are used to train and test, since it was trained in different projects than it was tested. We also increase our internal validity by following the existing techniques' instructions to replicate their techniques, including using the same machine learning algorithm.

### *11.3. External Validity*

To increase external validity, we studied a popular dataset that is prevalent among continuous integration studies: TravisTorrent [68]. TravisTorrent was created in 2016, but it continues to be studied in many research projects. Many of the techniques to save cost in CI

Published version available at: https://doi.org/10.1016/j.jss.2022.111292

[1, 63, 3, 54] were originally evaluated on TravisTorrent projects (some as recently as 2020). Additionally, we extensively curated TravisTorrent, removing: toy projects following standard practice [34, 35], unusable projects for test-granularity techniques, and cancelled builds as in past work [33, 58, 57]. After this curation process, our studies involved 82,427 builds from 100 different software projects. This high number of studied builds and projects gives us a higher confidence that the results of this work are likely to generalize to other projects.

The projects we chose were all Java or Ruby projects (18% of projects are Ruby projects), because there are no projects with other programming languages in the data set. Although these two programming languages are popular, different CI habits in other languages may provide slightly different results to the ones in this study.

Also for external validity, we decided to not remove tangled commits from our studied dataset. This is so that our study reflected the impact of CI-Skip rules and of our proposed technique in a realistic scenario, *i.e.,* in software projects that in practice may contain tangled commits. While tangled commits are deviations from the best practice of keeping commits small and single-purpose, they are a common occurrence in practice. Past work found that up to 16% of change sets associated with bug reports address multiple concerns [69]. This way, our study better captures the diversity of changes that go through CI in practice.

Another threat to external validity questions whether the thresholds used to highlight different configurations of PBS (Safe, Moderate, Relaxed, and More relaxed) will generalize to other projects. We do not present these thresholds as absolute values to be reused across software projects. These values may not generalize to other projects. We simply highlight them to allow for comparison with existing work. Our advice for developers using PBS in their software project is to empirically customize its prediction thresholds to their preference, for their software project.

Finally, our cost-saving technique may not be perfectly suitable for software projects that seldom build following any CI-Skip rules. However, this happens rarely according to our dataset.

## 12. Conclusions and Future Work

In this article, we aimed to maximize build failure observation and save cost in CI. To achieve this goal, we firstly studied the safety of CI-Skip rules and found that these rules are not perfectly safe. We then developed a set of CI-Run rules that make those rules invalid. We studied these CI-Run rules and found that they are correlated with failing builds under CI-Skip rules. Then we encoded our findings into PreciseBuildSkip, a novel build selection technique that can capture the majority of failing builds and provide cost-saving at the same time. Finally we evaluated our approach and compared it with existing techniques.

PreciseBuildSkip's variants improved existing approaches in term of Observed Failures and Cost Saving, *i.e.,* PreciseBuildSkip is able to save cost in a safer way. We highlight two specific variants that we posit will be popular: the safe one, which saves 5.5% builds and generally captures all of failing builds, and a version that is better at Cost Saving: saves 35% of builds while keeping 81% of failing build observations. Nevertheless, PreciseBuildSkip provides many other trade-offs that could be desirable in different environments. In the future, we will work on extending PreciseBuildSkip's algorithm with other machine learning algorithms and more CI-Skip rules to gain more benefit in cost-saving while keeping capturing all failing builds. Besides, we will find other ways to characterize failing builds based on the test execution history and habits (*e.g.,* [70, 71, 72]), source code history (*e.g.,* [73, 74, 75, 76, 77]), or the rationale of the commits being built (*e.g.,* [78]).

## 13. Acknowledgements

This material is based upon work supported by the National Science Foundation under award CCF-2046403, and by Universidad Rey Juan Carlos under the International Distinguished Researcher award C01INVESDIST.

## References

[1] R. Abdalkareem, S. Mujahid, E. Shihab, J. Rilling, Which commits can be ci skipped?, IEEE Transactions on Software Engineering (2019).

[2] M. Fowler, M. Foemmel, Continuous integration, ThoughtWorks) http://www. thoughtworks. com/Continuous Integration. pdf 122 (2006) 14.

[3] X. Jin, F. Servant, A cost-efficient approach to building in continuous integration, in: Proceedings of the 42th International Conference on Software Engineering, 2020, pp. 13–25.

[4] M. Hilton, T. Tunnell, K. Huang, D. Marinov, D. Dig, Usage, costs, and benefits of continuous integration in open-source projects, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ACM, 2016, pp. 426–437.

[5] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, D. Dig, Trade-offs in continuous integration: assurance, security, and flexibility, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, 2017, pp. 197–207.

[6] G. Pinto, M. Rebouças, F. Castor, Inadequate testing, time pressure, and (over) confidence: a tale of continuous integration users, in: Proceedings of the 10th International Workshop on Cooperative and Human Aspects of Software Engineering, IEEE Press, 2017, pp. 74–77.

[7] D. G. Widder, M. Hilton, C. Kästner, B. Vasilescu, A conceptual replication of continuous integration pain points in the context of travis ci, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, 2019, pp. 647–658.

[8] K. Herzig, M. Greiler, J. Czerwonka, B. Murphy, The art of testing less without sacrificing quality, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 1, IEEE, 2015, pp. 483–493.

Published version available at: https://doi.org/10.1016/j.jss.2022.111292

[9] John O'Duinn , The financial cost of a checkin, [Online; accessed 25-January-2019] (2013).
URL https://https://oduinn.com/2013/12/13/the-financial-cost-of-a-checkin-part-2/

[10] S. Elbaum, G. Rothermel, 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, 2014, pp. 235–245.

[11] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, J. Micco, Taming google-scale continuous testing, in: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), IEEE, 2017, pp. 233–242.

[12] A. Shi, S. Thummalapenta, S. K. Lahiri, N. Bjorner, J. Czerwonka, Optimizing test placement for module-level regression testing, in: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, 2017, pp. 689–699.

[13] M. Machalica, A. Samylkin, M. Porth, S. Chandra, Predictive test selection, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), IEEE, 2019, pp. 91–100.

[14] C. Zhu, O. Legunsen, A. Shi, M. Gligoric, A framework for checking regression test selection tools, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, 2019, pp. 430–441.

[15] L. Zhang, Hybrid regression test selection, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, 2018, pp. 199–209.

[16] M. Gligoric, L. Eloussi, D. Marinov, Practical regression test selection with dynamic file dependencies, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, 2015, pp. 211–222.

[17] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: a survey, Software Testing, Verification and Reliability 22 (2) (2012) 67–120.

[18] S. Yoo, M. Harman, Pareto efficient multi-objective test case selection, in: Proceedings of the 2007 international symposium on Software testing and analysis, ACM, 2007, pp. 140–150.

[19] G. Rothermel, M. J. Harrold, A safe, efficient regression test selection technique, ACM Transactions on Software Engineering and Methodology (TOSEM) 6 (2) (1997) 173–210.

[20] G. Rothermel, M. J. Harrold, Analyzing regression test selection techniques, IEEE Transactions on software engineering 22 (8) (1996) 529–551.

[21] R. Abdalkareem, S. Mujahid, E. Shihab, A machine learning approach to improve the detection of ci skip commits, IEEE Transactions on Software Engineering (TSE) (2020) To Appear.

[22] Anonymous, Minimizing the Side Effect of Cost-saving Build Selection in Continuous Integration, available at https://doi.org/10.5281/zenodo.4007140 (Aug. 2020). doi:10.5281/zenodo.4007140.
URL https://doi.org/10.5281/zenodo.4007140

[23] A. Miller, A hundred days of continuous integration, in: Agile 2008 Conference, IEEE, 2008, pp. 289–293.

[24] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. Di Penta, S. Panichella, A tale of ci build failures: An open source and a financial organization perspective, in: 2017 IEEE international conference on software maintenance and evolution (ICSME), IEEE, 2017, pp. 183–193.

[25] C. Zhang, B. Chen, L. Chen, X. Peng, W. Zhao, A large-scale empirical study of compiler errors in continuous integration (2019).

[26] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, M. Di Penta, How open source projects use static code analysis tools in continuous integration pipelines, in: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), IEEE, 2017, pp. 334–344.

[27] K. V. Paixão, C. Z. Felício, F. M. Delfim, M. de A Maia, On the interplay between non-functional requirements and builds on continuous integration, in: Proceedings of the 14th International Conference on Mining Software Repositories, IEEE Press, 2017, pp. 479–482.

[28] M. Cataldo, J. D. Herbsleb, Factors leading to integration failures in global feature-oriented development: an empirical analysis, in: Proceedings of the 33rd International Conference on Software Engineering, ACM, 2011, pp. 161–170.

[29] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, R. Bowdidge, Programmers' build errors: a case study (at google), in: Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 724–734.

[30] N. Kerzazi, F. Khomh, B. Adams, Why do automated builds break? an empirical study, in: Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on, IEEE, 2014, pp. 41–50.

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

[32] T. Rausch, W. Hummer, P. Leitner, S. Schulte, An empirical analysis of build failures in the continuous integration workflows of java-based open-source software, in: Proceedings of the 14th International Conference on Mining Software Repositories, IEEE Press, 2017, pp. 345–355.

[33] K. Gallaba, C. Macho, M. Pinzger, S. McIntosh, Noise and heterogeneity in historical build data: an empirical study of travis ci, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ACM, 2018, pp. 87–97.

[34] M. R. Islam, M. F. Zibran, Insights into continuous integration build failures, in: Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on, IEEE, 2017, pp. 467–470.

[35] A. Ni, M. Li, Cost-effective build outcome prediction using cascaded classifiers, in: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), IEEE, 2017, pp. 455–458.

[36] F. Hassan, X. Wang, Hirebuild: An automatic approach to history-driven repair of build scripts, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, 2018, pp. 1078–1089.

[37] F. Hassan, S. Mostafa, E. S. Lam, X. Wang, Automatic building of java projects in software repositories: A study on feasibility and challenges, in: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, 2017, pp. 38–47.

[38] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, V. Filkov, Quality and productivity outcomes relating to continuous integration in github, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, 2015, pp. 805–816.

[39] D. Ståhl, J. Bosch, Experienced benefits of continuous integration in industry software product development: A case study, in: The 12th iasted international conference on software engineering,(innsbruck, austria, 2013), 2013, pp. 736–743.

[40] M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä, T. Männistö, The highways and country roads to continuous deployment, Ieee software 32 (2) (2015) 64–72.

[41] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, B. Vasilescu, The impact of continuous integration on other software development practices: a large-scale empirical study, in: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, IEEE Press, 2017, pp. 60–71.

[42] B. Vasilescu, S. Van Schuylenburg, J. Wulms, A. Serebrenik, M. G. van den Brand, Continuous integration in a social-coding world: Empirical evidence from github, in: 2014 IEEE International Conference on Software Maintenance and Evolution, IEEE, 2014, pp. 401–405.

[43] W. Felidré, L. Furtado, D. A. Da Costa, B. Cartaxo, G. Pinto, Continuous integration theater, in: Proceedings of the 13th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2019, p. 10.


Published version available at: https://doi.org/10.1016/j.jss.2022.111292

[44] J. Liang, Cost-effective techniques for continuous integration testing (2018).

[45] T. A. Ghaleb, D. A. da Costa, Y. Zou, An empirical study of the long duration of continuous integration builds, Empirical Software Engineering (2019) 1–38.

[46] M. Tufano, H. Sajnani, K. Herzig, Towards predicting the impact of software changes on building activities, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), ICSE '19, 2019.

[47] X. Jin, F. Servant, What helped, and what did not? an evaluation of the strategies to improve continuous integration, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE, 2021, pp. 213–225.

[48] X. Jin, F. Servant, Cibench: a dataset and collection of techniques for build and test selection and prioritization in continuous integration, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), IEEE, 2021, pp. 166–167.

[49] Q. Luo, K. Moran, D. Poshyvanyk, M. Di Penta, Assessing test case prioritization on real faults and mutants, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2018, pp. 240–251.

[50] S. Mostafa, X. Wang, T. Xie, Perfranker: prioritization of performance regression tests for collection-intensive software, in: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2017, pp. 23–34.

[51] D. Marijan, A. Gotlieb, S. Sen, Test case prioritization for continuous regression testing: An industrial case study, in: 2013 IEEE International Conference on Software Maintenance, IEEE, 2013, pp. 540–543.

[52] S. Elbaum, A. G. Malishevsky, G. Rothermel, Test case prioritization: A family of empirical studies, IEEE transactions on software engineering 28 (2) (2002) 159–182.

[53] G. Rothermel, R. H. Untch, C. Chu, M. J. Harrold, Prioritizing test cases for regression testing, IEEE Transactions on software engineering 27 (10) (2001) 929–948.

[54] J. Liang, S. Elbaum, G. Rothermel, Redefining prioritization: continuous prioritization for continuous integration, in: Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 688–698.

[55] A. Celik, A. Knaust, A. Milicevic, M. Gligoric, Build system with lazy retrieval for java projects, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2016, pp. 643–654.

[56] A. Gambi, Z. Rostyslav, S. Dustdar, Improving cloud-based continuous integration environments, in: Proceedings of the 37th International Conference on Software Engineering-Volume 2, IEEE Press, 2015, pp. 797–798.

[57] M. Rebouças, R. O. Santos, G. Pinto, F. Castor, How does contributors' involvement influence the build status of an open-source software project?, in: Proceedings of the 14th International Conference on Mining Software Repositories, IEEE Press, 2017, pp. 475–478.

[58] R. Jain, S. K. Singh, B. Mishra, A brief study on build failures in continuous integration: Causation and effect, in: Progress in Advanced Computing and Intelligent Engineering, Springer, 2019, pp. 17–27.

[59] SciTools Understand, Understand static code analysis tool, `https://scitools.com/`, [Online; accessed 02-March-2020] (2020).

[60] Emad Elsaid, Rubrowser (ruby browser), `https://github.com/emad-elsaid/rubrowser`, [Online; accessed 21-January-2022] (2019).

[61] Wikipedia contributors, Cold start (computing) — Wikipedia, the free encyclopedia, [Online; accessed 21-February-2019] (2019).
URL `https://en.wikipedia.org/w/index.php?title=Cold_start_(computing)&oldid=883021431`

[62] A. I. Schein, A. Popescul, L. H. Ungar, D. M. Pennock, Methods and metrics for cold-start recommendations, in: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval, 2002, pp. 253–260.

[63] F. Hassan, X. Wang, Change-aware build prediction model for stall avoidance in continuous integration, in: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, 2017, pp. 157–162.

[64] P. M. Duvall, S. Matyas, A. Glover, Continuous integration: improving software quality and reducing risk, Pearson Education, 2007.

[65] J. Micco, The state of continuous integration testing at google (2017).

[66] X. Jin, F. Servant, The hidden cost of code completion: Understanding the impact of the recommendation-list length on its efficiency, in: Proceedings of the 15th International Conference on Mining Software Repositories, 2018, pp. 70–73.

[67] W. G. Cochran, Sampling techniques-3 (1977).

[68] M. Beller, G. Gousios, A. Zaidman, Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration, in: Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on, IEEE, 2017, pp. 447–450.

[69] K. Herzig, A. Zeller, The impact of tangled code changes, in: 2013 10th Working Conference on Mining Software Repositories (MSR), IEEE, 2013, pp. 121–130.

[70] A. Gautam, S. Vishwasrao, F. Servant, An empirical study of activity, popularity, size, testing, and stability in continuous integration, in: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), IEEE, 2017, pp. 495–498.

[71] A. M. Kazerouni, C. A. Shaffer, S. H. Edwards, F. Servant, Assessing incremental testing practices and their impact on project outcomes, in: Proceedings of the 50th ACM Technical Symposium on Computer Science Education, 2019, pp. 407–413.

[72] A. M. Kazerouni, J. C. Davis, A. Basak, C. A. Shaffer, F. Servant, S. H. Edwards, Fast and accurate incremental feedback for students' software tests using selective mutation analysis, Journal of Systems and Software 175 (2021) 110905.

[73] F. Servant, J. A. Jones, History slicing, in: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), IEEE, 2011, pp. 452–455.

[74] F. Servant, J. A. Jones, History slicing: assisting code-evolution tasks, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, 2012, pp. 1–11.

[75] F. Servant, J. A. Jones, Chronos: Visualizing slices of source-code history, in: 2013 First IEEE Working Conference on Software Visualization (VISSOFT), IEEE, 2013, pp. 1–4.

[76] F. Servant, J. A. Jones, Fuzzy fine-grained code-history analysis, in: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, 2017, pp. 746–757.

[77] F. Servant, Supporting bug investigation using history analysis, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2013, pp. 754–757.

[78] K. A. Safwan, F. Servant, Decomposing the rationale of code commits: the software developer's perspective, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 397–408.


Published version available at: `https://doi.org/10.1016/j.jss.2022.111292`