Cost-saving in Continuous Integration: Development, Improvement, and Evaluation of Build Selection Approaches

Xianhao Jin

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science and Applications

April 29, 2022

Blacksburg, Virginia

Keywords: Continuous Integration, Build Strategies, Maintenance Cost

Cost-saving in Continuous Integration: Development, Improvement, and Evaluation of Build Selection Approaches

Xianhao Jin

(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 CI systems in millions of dollars. In this dissertation, I propose a collection of novel build selection approaches that are able to save the cost of CI. I also propose the first exhaustive comparison of techniques to improve CI including build and test granularity approaches. I firstly design a build selection approach (SmartBuildSkip) for CI cost reduction in a balanceable way. The evaluation of SmartBuildSkip shows that it can save a median 30% of builds by only incurring a median delay of 1 build in a median of 15% failing builds under its most conservative configuration. To minimize the delayed failure observation, I then propose the second build selection approach (PreciseBuildSkip) that can save cost without delaying failure observation. We find that PreciseBuildSkip can save a median of 5.5% of builds while capturing the majority of failing builds (100% in median) from the evaluation. After that, I evaluate the strengths and weaknesses of 10 techniques that can improve CI including SmartBuildSkip. The findings of the comparison motivate my next work to design a hybrid technique (HybridBuildSkip) that combines these techniques to produce more cost saving while keeping a low proportion of failing builds that are delayed in observation. Finally, I design an experiment to understand how different weights of test duration among the whole build duration can influence the cost saving of build and test selection techniques.

Cost-saving in Continuous Integration: Development, Improvement, and Evaluation of Build Selection Approaches

Xianhao Jin

(GENERAL AUDIENCE ABSTRACT)

Modern software developing teams commonly use the continuous integration as the practice of automating and testing the integration of code changes from multiple contributors into a single software project. The best practice of continuous integration requires this process happens as frequently as possible because the bugs can be found earlier and easier before the change sets grow too large. However, continuous integration process can be time-consuming and in most cases the code change is bug-free. This means that developers may have to wait for a long time only to get a result with no actionable feedback. Thus, in this dissertation, I present multiple selection approaches to selectively execute the continuous integration process based on the prediction of the outcome - if the outcome is predicted to be passing with no actionable feedback, the approach will decide to skip the current execution. The evaluation result shows that my approaches can save the cost of continuous integration while keeping the value of it (finding bugs earlier).

Dedication

Dedicated to my families. Thank you for all your support along the way.

Acknowledgments

To Shuning: Thank you for your love, for your support, and for every minute that we spend together. Thank you for coming to the U.S. with me and changing your major of Japanese language. Thank you for accompanying with me during the tough pandemic time. Thank you for every sweet word and every smile from you in my daily life.

To my parents: Thank you for your unconditional love and support. Thank you for your encouragement for every decision I made. Thank you for your understanding, even sometimes it meant your son would live on the opposite side of the earth for many years.

To my SeaLab mates: Thank you for answering my questions and for all the group meetings and discussions.

To my committee members: Thank you for your timely feedback and helpful suggestions. I am particularly grateful to Drs. Na Meng, Eli Tilevich, B. Aditya Prakash, and Sebastian Elbaum, for valuable insights that guide me to improve this dissertation and further study relevant research topics.

To my advisor, Dr. Francisco Servant: Thank you for inviting me to your research lab. Before coming here, I knew little about doing research and the academic life. Thank you for showing me how a creative, meticulous and intelligent researcher who has a thirst for knowledge should be like. Thank you for every advice and encouragement from you about my academic and daily life. I appreciate your guiding me to find correct research directions and your answers to my questions during our discussions. I can never accomplish this without your help and support.

To all sources that provided me financial support: I am thankful for the financial support from multiple sources for my research: National Science Foundation CAREER CCF-2046403 and International Distinguished Researcher award C01INVESDIST by Universidad Rey Juan Carlos. I would also like to thank Virginia Tech Computer Science department that offered me admission with funding when needed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem statement

Continuous integration (CI) is a popular practice in modern software engineering that encourages developers to build and test their software in frequent intervals [24]. While CI is widely recognized as a valuable practice, it also incurs a very high cost — mostly for the computational resources required to frequently run builds [41, 43, 44, 77, 110]. Overall, adopting CI can be very expensive. Google estimates the cost of running its CI system in millions of dollars [44], and Mozilla estimates theirs as $201,000 per month [55]. For smaller-budget companies that have not yet adopted CI, this high cost can pose a strong barrier.

There are many existing research approaches to save cost in CI, including techniques to make CI builds faster by accelerating preparation phase [11, 28] or running fewer tests [67]. In contrast, I propose a collection of novel build selection strategies that focus on skipping builds that are predicted to pass to reduce the cost of CI. The goal of my proposed techniques is to **execute fewer builds**, while **running as many failing builds as early as possible**. The rationale behind my strategy is: skip builds that are predicted to pass and execute builds that are likely to fail. I posit that the value of CI lies in the observation of failures and its cost lies in the build executions.

Figure 1.1 depicts an example timeline of CI builds including passing builds (Build 1 and 5 -

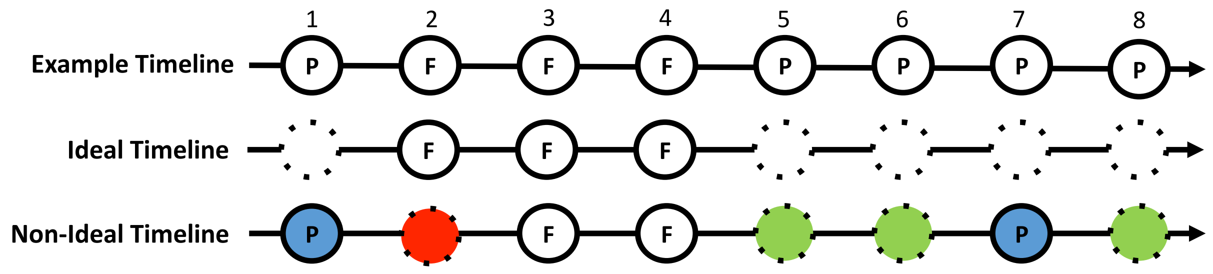


Figure 1.1: Example CI timeline. Circles with character P represent passing builds, *e.g.,* Build 2, and the character F represents a failing build outcome. Dashed circles represent skipped builds.

8) as the cost of CI and failing builds (Build 2 - 4) as the value of CI which are not desirable to skip. Each build is composed of many phases including build preparation and test. In the ideal timeline, all builds are perfectly predicted so all passing builds are skipped and all failing builds are executed. However, in the non-ideal timeline, passing builds can be falsely labeled as failing builds and thus get executed (Build 1 and 7). Worse still, failing builds can be falsely predicted as passing builds and thus get skipped (Build 2). The former causes a reduction of cost saving. The latter results in a more serious problem, dividing skipped builds into two groups: passing ones (green dashed circles) and failing ones (red dashed circles). These failing builds that are skipped by mispredictions can cause a delay of failure observation, *e.g.,* failing Build 2 can only be observed at the time when Build 3 executes. If a technique achieves higher cost saving (skips more builds), it is more likely to skip more failing builds at the same time, *i.e.,* there exists a trade-off. To address this problem, my work focuses on maximizing skipped passing builds while minimizing skipped failing builds.

In light of this trade-off, this dissertation investigates two questions:

(1) How can build selection approaches maximize their benefit and minimize their side effect?

(2) How effective are build selection approaches compared with other CI-improving techniques?

1.2 Thesis

Cost in Continuous Integration can be saved via automated build selection techniques in ways that:

1. balance cost saving and delay of failure observation, or
2. reduce delay of failure observation, and
3. enhances the ability of cost saving.

1.3 Research Contributions and Applications

This dissertation has following contributions:

Contribution 1: Balance cost saving and delay of failure observation. (published at ICSE'20 [51])

I created the first approach (SmartBuildSkip) as a technique to skip builds automatically by predicting build outcomes, to help developing team save the computational cost of CI and waiting time of the CI outcome. The design of SmartBuildSkip is based on the novel conceptual separation of build failures into first and subsequent failures, to improve the effectiveness of build prediction models. To motivate my design, I performed two empirical studies, of the prevalence of build passes over build failures, and of subsequent failures over first failures. I also studied the factors that predict first failures. Furthermore, I performed an evaluation of the extent to which SmartBuildSkip can save cost in CI while keeping most of its value, with the ability of customizing its cost-value balance.

Contribution 2: evaluate existing CI-improving techniques. (published at ICSE'21 [52, 53] and ESEC/FSE'21 [48])

I then performed the first comprehensive evaluation of CI-improving techniques including SMARTBUILDSKIP from different settings to understand their benefits. The evaluation includes a replication of 14 variants of 10 CI-improving techniques from 4 technique families. I used a collection of metrics to measure the performance of CI-improving techniques over various dimensions (computational-cost reduction, missed failure observation, and early feedback). To compare the techniques in the same environment, I extended the popular Travis Torrent dataset [7] with: detailed test and commit, and dependencies information. The contribution also includes the findings that can provide evidence for researchers to design future CI-improving techniques.

Contribution 3: Reduce delay of failure observation. (accepted at JSS'22 [54]))
To minimize the side-effect of mispredictions of failing builds by build-selection approaches (found in results of Contribution 1 and 2), I proposed PRECISEBUILDSKIP as a technique that provides cost-savings in CI while capturing an overwhelming majority of failing builds. PRECISEBUILDSKIP is designed by exploring CI-Skip rules [2] and corresponding CI-Run rules. To motivate the design, I also performed the empirical studies about the evaluation of CI-Skip rules and the relationship between CI-Skip and CI-Run rules. I also designed a novel metric that is able to compare build-selection techniques in a more comprehensive way. Besides, I performed an evaluation to compare PRECISEBUILDSKIP with existing techniques including SMARTBUILDSKIP in the context of CI cost-saving.

Contribution 4: Enhance the ability of cost saving by combining build and test selection approaches. (under review at TOSEM)
Since the existing no-delay build selection technique (PRECISEBUILDSKIP) is not able to provide high cost savings, I designed a hybrid technique — HYBRIDCISAVE that takes advantage of all existing work's strengths to enlarge the cost-saving significantly while keeping a low side-effect based on the previous comparison result. The technique is a hybrid method based

on the prediction results of existing techniques, predicts build outcomes across granularity, and combines both build and test selection strategies to allow skipping full and partial build. I finally performed a study to examine the impact of the technique execution time on the cost saving.

1.4 Structure

The rest of this dissertation is organized as follows. Chapter 2 provides the necessary background required to understand the proposed research work. Chapter 3 describes SmartBuildSkip as the first build selection approach to balance cost saving and failure observation. Chapter 4 presents the first comprehensive evaluation of CI-improving techniques under the same context. Chapter 5 describes PreciseBuildSkip as the first build selection approach to minimize the delayed failure observation. Chapter 6 describes HybridCISave as the first hybrid build selection approach and presents a synthetic study to understand how the proportion of test time to total build time can influence the cost saving ability. Chapter 7 outlines the future work directions, and Chapter 8 presents concluding remarks.

Chapter 2

Background and Related Work

In this chapter, we firstly introduce definitions and technical background required to understand this dissertation. We then discuss related work.

2.1 Definitions and Technical Background

2.1.1 Continuous Integration

Continuous integration (CI) is a DevOps [17] software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run. Continuous integration aims to solve the problem that developers might work in isolation and checked in their changes only after its full completion, resulting in a time-consuming merging and accumulated bugs without correction. Continuous integration benefits the software developing team by improving developers' productivity, finding and addressing bugs earlier and accelerating the delivery process. The best practices of continuous integration requires developers to commit early and often. Well-known examples of CI services are Jenkins[1], Travis[2], CircleCI[3] and AppVeyor[4] [34]. CI services can also be built-in in social coding platforms such as GitHub and GitLab [14]. Besides, big tech companies such

[1]https://www.jenkins.io/
[2]https://travis-ci.org/
[3]https://circleci.com/
[4]https://www.appveyor.com/

as Google and Facebook have their own designed continuous integration system.

A full CI build comprises 1) a traditional build and compile phase, 2) a phase in which automated static analysis tools (ASATs), and 3) a testing phase, in which unit, integration, and system tests are run [6]. In the practice, the build can include multiple jobs and these jobs can be executing in a parallel way. Any of failures happening in any of these three phases can make the build broken, *i.e.,* errored or failed. In this dissertation, we refer to *build* as the whole process of a full CI build including all of these three phases. We refer to *test* as the test suites executed in the test phase of the process, *i.e.,* phase 3.

2.1.2 Regression Test Selection (RTS)

Regression test selection analyzes incremental changes to a codebase and chooses to run only those tests whose behavior may be affected by the latest changes in the code [33]. By focusing on a small subset of all the tests, the testing process runs faster and can be more tightly integrated into the development process. Regression test selection aims to reduce the infrastructure costs of testing changes submitted by developers, as well as to speed up delivery of correctness signal. A typical RTS technique requires two dimensions of information: (1) the test dependency information on an old program version, (2) the changed program elements. Then, a safe RTS technique selects any test whose dependencies overlap with the changed program elements as the affected tests, since missing any of those tests may fail to detect some regression bugs [118]. RTS techniques can be categorized as dynamic [32, 35, 74, 82, 84] and static [57, 59] techniques based on how the test dependencies are collected. RTS can also be categorized as basic-block-level [35, 74, 84], method-level [82], file-level [32, 59], and module-level [97, 106] techniques based on the granularities of changed elements. RTS techniques are used in continuous integration environment to find changes that introduce regression faults [98]. In this dissertation, we consider regression test selection techniques as test suite level, which means these techniques can selectively executing some test suites in the test phase of the continuous integration context.

2.1.3 Test Case Prioritization (TCP)

Test case prioritization is a technique that prioritizes the execution order of the test cases and it is developed in order to run test cases of higher priority so that as many distinct faults as possible are detected early in the execution of the test suite. The prioritization of TCP can depend on the probability of fault existence or the coverage of the functions and statements [18]. The former may be reflected by the historical fail ratio of the test, the relationship between the change set and the test or the authorship of the commit. The latter can be reflected by the distinction among tests and the execution time of the test. Compared with RTS techniques, TCP techniques don't remove any tests and thus can be safer. TCP was a rich research area even before continuous integration became a common practice, *e.g.,* [85]. TCP techniques are commonly used in continuous integration to re-order tests to identify an "ideal" order of test cases that maximizes specific goals, such as early fault detection [64]. In this dissertation, we consider regression test selection techniques to be executed test suite level, which means these techniques can prioritize the test suites that are going to be executed in the test phase of the continuous integration environment.

2.1.4 Build Prioritization and Selection

Build prioritization technique [63] takes advantage of the ideas from TCP techniques that faults can be detected earlier by prioritizing element execution orders. Thus, build prioritization technique performs on the granularity of build/commit level. It aims to address the problem of TCP techniques in CI environment that TCP techniques cannot produce large reductions in feedback time and it is not practical to prioritize tests across builds. Since the arriving time difference among builds or commits, build prioritization technique is only triggered when multiple builds are waiting for available work units when all computing resources are occupied. The prioritization rationale is similar to TCP techniques, *i.e.,* commonly depending on the historical failure ratio.

Build selection techniques [1, 2, 51] selectively executes builds or commits that should have triggered the continuous integration system. These techniques aim to execute a subset of builds to save the computational cost as well as developers' waiting time for feedback. Compared to RTS techniques, build selection approaches skip executions in the granularity of builds instead of tests. This brings them more benefits of cost saving along with higher possibility to delay failure observation. Build selection approaches skip builds or commits based on different criterion: some decide to skip builds that are likely to be skipped by developers while others prefer to skip builds that likely to be passing. This reflects different understanding of what builds should be skipped. The decision of skipping one build is based on the prediction of whether the build satisfies the requirement to be skipped. To achieve this goal, these techniques explore factors to make predictions.

2.2 Related Work

2.2.1 Empirical Studies of CI and its Cost

Multiple researchers focused on understanding the practice of CI, studying both practitioners *e.g.,* [44] and software repositories [108]. Vasilescu *et al.* studied CI as a tool in social coding [107], and later studied its impact on software quality and productivity [108]. Zhao *et al.*

studied the impact of CI in other development practices, like bug-fixing and testing [119]. Stahl *et al.* [101] and Hilton *et al.* [44] studied the benefits and costs of using CI, and the trade-offs between them [43]. Lepannen *et al.* similarly studied the costs and benefits of continuous delivery [61]. Felidré *et al.* [22] studied the adherence of projects to the original CI rules [24]. Other recent studies focused on the difficulties [77] and pain points [110] of CI.

The high cost of running builds is highlighted by many empirical studies as an important problem in CI [41, 43, 44, 77, 110] — which reaches millions of dollars in large companies, *e.g.,* at Google [44] and Microsoft [41].

2.2.2 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.,* [30, 105]. Thus, most of the approaches that reduce the cost of CI aim at making builds faster by running fewer test cases on each build. Some approaches use historical test failures to decide which tests to run [19, 41] Others run tests with a small distance with the code changes [69] or skip those testing unchanged modules [97]. Recently, Machalica *et al.* predicted test case failures using a machine learning classifier [67]. These techniques are based on the broader field of regression test selection (RTS) *e.g.,* [20, 32, 76, 83, 84, 114, 115, 118, 122]. 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 not building when builds only include non-code changes and predicting which builds may be decided to skip by developers [1, 2, 86]. Our techniques predict build outcomes for any kind of changes (code and non-code). Thus, our work complements existing techniques to reduce cost in CI, and could potentially be applied in addition to them.

A related effort for improving CI aims at speeding up its feedback by prioritizing its tasks. The most common approach in this direction is to apply test case prioritization (TCP) techniques *e.g.,* [18, 19, 66, 68, 71, 85] so that builds fail faster. Another similar approach achieves faster feedback by prioritizing builds instead of tests [63]. In contrast, our work focuses on saving cost in CI by skipping tasks instead of prioritizing them. Prioritization-based techniques increase feedback speed but do not focus on saving cost, *i.e.,* all builds still get executed, and all passing tests get executed if no test failure is observed.

Finally, other complementary efforts to reduce build duration have targeted speeding up the compilation process *e.g.,* [11, 26] or the initiation of testing machines *e.g.,* [28].

2.2.3 Characterizing Failing Builds

Multiple studies investigated the reasons why builds fail. Some studies [70, 109] found that the most common build failures were compilation [117], unit test, static analysis [116], and server errors. Paixão *et al.* [75] studied the interplay between non-functioal requirements and failing builds. Other studies found factors that contribute to build failures: architectural dependencies [10, 89] and other more specific factors, such as the stakeholder role, the type of work item and build [56], or the programming language [6]. Other less obvious factors that could cause build failures are build environment changes or flaky tests [80]. Rausch *et al.* [80] also found that build failures tend to occur consecutively, which Gallaba *et al.* [27] describe as “persistent build breaks”. These observations inform our findings about subsequent build failures that they would be numerous, easy to anticipate and able to break CI-Skip rules.

Other studies found change characteristics that correlate with failing builds, such as: number of commits, code churn [45, 80], number of changed files, build tool [45], and statistics on the last build and the history of the committer [73]. In our study of SMARTBUILDSKIP, we separate failing builds into *first failures* and *subsequent failures.* We found that *first failures* are predicted by some of the factors that predict all builds (line, file, and test churn, and number of commits), but also by factors that were not found to correlate with all builds (project size, age, and test density). In our study of PRECISEBUILDSKIP, we found that some of these features can be used as CI-Skip rules but can still break the build even the change is considered safe.

Finally, other studies investigated the noise of build breakage data [31] and characteristics of build failures outside the CI context [37, 79, 104]

2.2.4 Predicting Failing Builds

Some works aimed at predicting build outcomes in industrial settings where continuous integration was not yet adopted. These techniques mostly approached this problem using machine learning classifiers, *e.g.,* measuring social and technical factors and using decision trees [36]; applying social network analysis and measuring socio-technical factors [58, 112]; and using code metrics on incremental decision trees [23].

In the continuous integration context, Ni and Li [73] predict build outcomes using cascade classifiers measuring statistics about the last build and the committer of the current build. Xie and Li [113] use a semi-supervised method over change metrics and the last build's outcome. Hassan and Wang [38] use a predictor over the last build's status and type. Since all these predictors rely on the outcome of the last build to be known, their prediction power may be limited in a cost-saving context, where the last build means the last build that was executed. In contrast to these predictors, SMARTBUILDSKIP is not affected by how *stale* the last build status is, since it does not rely on it for its prediction. Abdalkareem *et al.* designed techniques to predict builds that are likely to be skipped using CI-Skip rules [1, 2]. However, CI-Skip rules are not evaluated themselves and builds under those rules can also be failing builds. PreciseBuildSkip takes advantage of both CI-Skip rules and their exceptions to minimize the mispredictions and thus can capture the majority of failing builds.

2.2.5 Evaluation frameworks for similar techniques

Multiple research works focus on comparing cross-tool performance with an evaluation framework. Zhu *et al.* [122] propose a regression test selection framework to check the output against rules inspired by existing test suites for three techniques. Leong *et al.* [60] propose a test selection algorithm evaluation method and evaluate five potential regression test selection algorithms, finding that the test selection problem remains largely open. Najafi *et al.* [72] studied the impact of test execution history on test selection and prioritization techniques. Luo *et al.* [66] conduct the first empirical study comparing the performance of eight test prioritization techniques applied to both real-world and mutation faults and find that the relative performance of the studied test prioritization techniques on mutants may not strongly correlate with performance on real faults. Lou *et al.* [65] systematically created a taxonomy of existing works in test-case prioritization, classifying them in: algorithms, criteria, measurements, constraints, scenarios, and empirical studies.

Differently to these works, our study in this dissertation specifically targets the context of CI, and it has a broader focus than test prioritization or selection. Our study is the first to compare all the techniques proposed to reduce time-to-feedback or cost in CI, including prioritization and selection techniques, at test and build granularities. We performed observations comparing across 2 goals, 3 dimensions, 10 metrics, 2 granularities, and 10 techniques. Most of our observations required comparisons at broad scope. For example: we revealed the need for a new incentive in test selection to skip full test suites (to also save build-preparation time), which would not be relevant in studies outside the scope of CI.

Chapter 3

SmartBuildSkip: balance cost saving and failure observation delay

In this chapter, we aim to **reduce the high cost of CI** while **keeping as much of its value** as possible, *i.e.,* **balance the cost saving and delay of failure observation**. The cost of CI is commonly defined by the cost of builds [44, 73], and its value is defined by its ability to reveal problems early [16, 24]. Thus, we aim to reduce the cost of CI by **running fewer builds**, while **running as many failing builds as early as possible**. Our goal also responds to the need to run fewer builds that developers frequently express in Q&A websites[100], which they currently may approach by using CI plug-ins [13, 47, 103] to manually skip builds that they deem "safe", *e.g.,* changes in README files.

Existing research approaches to save cost in CI include the automatic detection of such non-code changes [2] and techniques to make CI builds faster [41, 67]. In contrast, our proposed approach focuses on skipping builds that are predicted to pass in more complicated cases — for any kinds of changes that happened between builds. Our approach complements existing techniques and could potentially be applied in combination with them.

We propose SmartBuildSkip, a novel approach to reduce the cost of CI based on automatic build-outcome prediction — by skipping builds that it predicts will pass, and running builds that it predicts will fail. Our strategy is motivated by two hypotheses: H_1: *Most builds in CI return a passing result.* We expect that software changes will generally be done carefully, making passing builds more common than failing builds. By this hypothesis, skipping passing builds would produce large cost savings. H_2: *Many failing builds in CI happen consecutively after another build failure.* One of the strongest predicting factors in existing build-outcome predictors is the previous build result [38, 73, 113]. Also, Rausch *et al.* observed build failures mostly occur consecutively in a small number of Java projects [80]. By this hypothesis, most failing builds could be easily predicted — since most follow another build failure.

Thus, SMARTBUILDSKIP differentiates between ***first failures*** and ***subsequent failures***, following a two-phase process. First, SMARTBUILDSKIP uses a machine-learning classifier to *predict* build outcomes to catch *first failures*. After it observes a *first failure*, it then *determines* that all *subsequent builds* will fail — until it observes a build pass and then changes its operation to predicting again. This strategy aims to address the limitations of existing build-prediction approaches [38, 73, 113], which strongly rely on the outcome of the last build, and *predict* outcome for *all builds* — likely incorrectly predicting some *first* and *subsequent failures*.

Lastly, we propose SMARTBUILDSKIP as a **customizable** technique, in order to help software developers with different cost-saving trade-off needs, *e.g.,* preferring modest effort savings and low delays in observing build failures, or preferring high effort savings with a longer delay to observe build failures.

We performed two empirical studies and two experiments. First, we empirically studied the hypotheses that motivate SMARTBUILDSKIP. Second, we empirically studied the features that predict *first failures*, to inform SMARTBUILDSKIP's predictor. Third, we performed an experiment to evaluate SMARTBUILDSKIP's ability to predict *first* and *all failures* in a dataset of 359 software projects and another one of 37 projects. Fourth, we performed another experiment to measure the cost savings that SMARTBUILDSKIP would produce in our studied datasets. In our experiments, we compared SMARTBUILDSKIP's performance with the state-of-the-art build prediction technique, HW17 [38]. HW17 makes machine-learning predictions for all builds, using both historical and contemporary build information. To the extent of our knowledge, HW17 is the build-prediction technique that currently provides the highest precision and recall.

In our experiments, we compared SmartBuildSkip's performance with the state-of-the-art build prediction technique, HW17 [38]. HW17 makes machine-learning predictions for all builds, using both historical and contemporary build information. To the extent of our knowledge, HW17 is the build-prediction technique that currently provides the highest precision and recall.

SmartBuildSkip provides two major strengths over HW17: (1) SmartBuildSkip runs predictions only for *first failures*, and determines that all subsequent builds fail until a pass is observed. (2) SmartBuildSkip predicts based only on features describing the current build and the project (using no features about the previous build). We found that this strategy was more effective at predicting both *first* and *subsequent failures* (see §3.5). Additionally, we found that, by not relying on the outcome of the previous build, SmartBuildSkip was much more effective in practice. Since the previous build was often skipped and its outcome unknown, HW17 was negatively impacted, but not SmartBuildSkip (observed in §3.6).

The results of our studies support our hypotheses — build passes are numerous (median 87% of all builds), and *subsequent failures* are also a high proportion of *all build failures* (median 52%). In our experiments, SmartBuildSkip significantly improved the accuracy of the state-of-the-art build predictor — up to median 8% F-measure for *first failures*, and up to median 52% F-measure for *all failures.* Finally, SmartBuildSkip's predictions resulted in high savings of build effort that could be customized for developers with different preferred trade-offs, *i.e.,* faster observation of build failures vs. higher savings in build effort. In its most conservative configuration, SmartBuildSkip saved a median 30% of all builds by

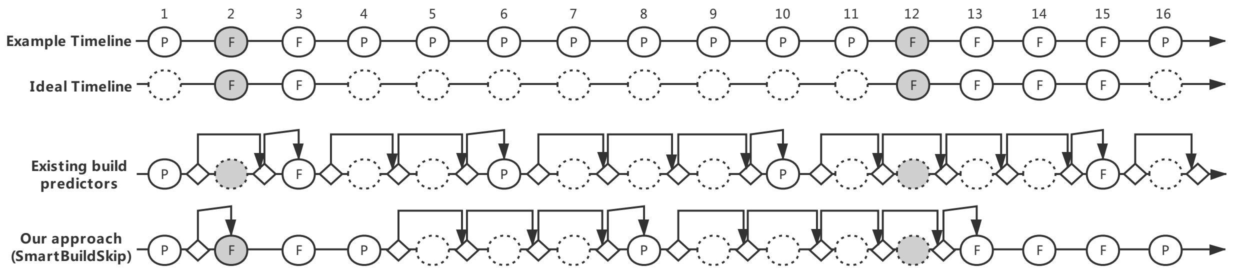


Figure 3.1: Motivating example timeline. *first failures* are highlighted in gray. In an ideal timeline, we would skip all passing builds and run all failing builds. Existing approaches predict outcome for every build. Our approach predicts build outcome if the last build passed. After observing a failure, it continues building until a pass is observed and it goes back to predicting.

only incurring a median delay of 1 build in a median 15% build failures. In a more cost-saving-focused configuration, SMARTBUILDSKIP saved a median 61% of all builds by incurring a 2-build delay for 27% of build failures.

This work provides following **contributions**:

- The conceptual separation of build failures into *first* and *subsequent failures*, to improve the effectiveness of build prediction models.

- Two studies, of the prevalence of build passes over build failures, and of *subsequent failures* over *first failures.*

- A study of factors that predict *first failures.*

- SMARTBUILDSKIP, a customizable, automatic technique to save cost in CI by predicting build outcomes, that can be applicable with or without training data, and that improves the prediction effectiveness of the state-of-the-art.

- A collection of simple predictors, based on factors that predict *first failures*, that can be applied as a rule-of-thumb, with no adoption cost.

- An evaluation of the extent to which SMARTBUILDSKIP can save cost in CI while keeping most of its value, with the ability of customizing its cost-value trade-off.

3.1 Motivating Hypotheses

We motivate our hypotheses and our proposed approach with an example. Figure 3.1 depicts an example timeline of builds, the ideal timeline in which we would save most effort, the timeline produced after applying a state-of-the-art build prediction technique, and the timeline produced after applying our approach SmartBuildSkip. The example timeline shows a numbered sequence of builds in CI. We depict passing builds as circles with a P and failing builds as circles with an F. The ideal timeline shows the outcome that an ideal technique would achieve — skipping every passing build and building all failing builds. We depict skipped builds with a dashed empty circle. This ideal timeline depicts our goal of saving cost in CI by running as few builds as possible while running as many failing builds as possible.

We propose SmartBuildSkip following two main hypotheses: H_1**: Most builds in CI return a passing result.** If this was true, our strategy of predicting build outcomes and skipping those expected to pass would provide substantial cost savings — since passing builds would be a majority and they would be skipped. H_2**: Many failing builds in CI happen consecutively after another build failure.** If true, if we built an automatic approach that predicted that subsequent builds to a failing build will also fail, we would correctly predict a substantial portion of failing builds.

First failures* vs. *subsequent failures. Assuming that our hypothesis H_2 would be supported, we also propose the distinction between *first failures* — the first build failure inside a sequence of build failures — and *subsequent failures* — all the remaining consecutive build failures in the sequence. Figure 3.1 highlights *first failures* with gray fill.

Limitations of existing work. Figure 3.1 also illustrates the limitations of applying existing build predictors (e.g., [38, 73, 113]) to the problem of saving cost in CI by skipping passing builds. The timeline for “existing build predictors” uses a diamond to depict the prediction of the outcome of an upcoming build. If the upcoming build is predicted to pass, the technique skips it and transitions to predict for the next build. We depict this with an arrow leaving the diamond and going into the next diamond, *e.g.,* in build 2. If the upcoming build is predicted to fail, it is executed. We depict this with an arrow leaving the diamond and going into the next build. We posit that existing predictors, by not distinguishing *first* and *subsequent failures*, likely provide limited accuracy for both.

Limited prediction of first failures. We posit that existing predictors will rarely correctly predict *first failures*, because they strongly rely on the status of the previous build for prediction. *first failures* are preceded by a build pass, by definition. However, we expect that it’s more often build passes that are preceded by a build pass. Thus, after observing a build pass, we expect that existing predictors will more likely predict another build pass to follow — likely not catching many *first failures*. SMARTBUILDSKIP, in turn, does not suffer from this limitation, since it does not rely on the outcome of the last build for its prediction.

Limited prediction of subsequent failures. Since existing techniques predict outcome for all builds — even after observing a *first failure*, they may incorrectly predict some *subsequent failures* to pass. SMARTBUILDSKIP, in turn, will correctly anticipate *subsequent failures*, since it does not make predictions for them. Instead, it *determines* that subsequent builds to a failure will also fail.

3.2 Our approach: SMARTBUILDSKIP

We designed SMARTBUILDSKIP by following the two hypotheses that we described. We also include the timeline produced by SMARTBUILDSKIP for our motivating example in Figure 3.1.

SmartBuildSkip's overall strategy. SmartBuildSkip follows a two-phase strategy. In its first phase, SmartBuildSkip predicts the outcome of the next build based on a set of predicting features. If the build is predicted to pass, it is not executed — its cost is saved — and SmartBuildSkip will predict again for the next build. An example is build 5 in Figure 3.1. If the build is predicted to fail, SmartBuildSkip executes it and checks its outcome. If the actual outcome of the executed build is pass, SmartBuildSkip will predict again for the next build — as in build 8 in Figure 3.1. If the actual outcome of the executed build is fail, SmartBuildSkip will shift to its second phase — as in build 2 in Figure 3.1. In its second phase, SmartBuildSkip *determines* that all subsequent builds will fail and thus executes them until the build passes, after which it returns to the first phase — as in builds 2–4 in Figure 3.1.

The benefit of this two-phase strategy is that we expect SmartBuildSkip to be more successful at identifying both *first failures* and *subsequent failures*, by treating them separately. We expect it to predict *first failures* better than existing techniques, since we train SmartBuildSkip's predictor using features that specifically predict *first failures*. We also expect it to accurately predict most *subsequent failures* by determining that all builds after a failing one will also fail.

The downside of this approach is that, by continuously building after observing a *first failure*, one false positive is guaranteed for every sequence of failures — as in builds 5 and 8 in Figure 3.1. However, we believe that this downside is smaller than the benefit that SmartBuildSkip gets from its overall strategy. Besides, existing predictors will also likely incur in these false positives because they strongly rely on the last build status — which in these cases is a bad predictor. Finally, we argue that these *first-pass* builds are valuable for practitioners, because they inform them of when they have fixed the problem that caused the build to fail.

SMARTBUILDSKIP's Variants. We propose two variants of SMARTBUILDSKIP. Both variants use a random forest classifier to predict builds. Since our focus is to correctly predict failing builds, and since we expect CI build output to often be imbalanced, SMARTBUILDSKIP trains with a class weight of 20:1 in favor of failing builds.

SMARTBUILDSKIP-WITHIN: This variant is trained in the past builds within the same software project in which it is applied. It uses the build features that we reported in 3.1.4.

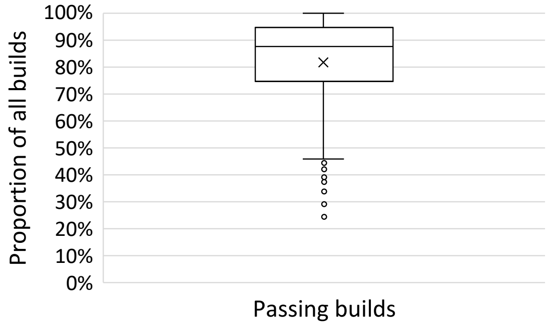
SMARTBUILDSKIP-CROSS: This variant is trained in the past builds of different software projects than the one in which it will be applied. It uses the build features as well as the project features that we report in 3.1.4. We propose this variant to help with the cold-start problem [111] in software projects for which only a few builds have been executed and they would not be enough to provide high-quality predictions.

Studied subjects. We perform our study over the TravisTorrent dataset [6], which includes 1,359 projects (402 Java projects and 898 Ruby projects) with data for 2,640,825 build instances. 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 1,000 lines of source code, which is a criteria applied in multiple other works [45, 73]. After this filtering, we obtained 274,742 builds from 359 projects (53,731 failing builds). We focused our study on builds with passing or failing result, rather than error or canceled — since they can be exceptions or may happen during the initialization and get aborted immediately before the real build starts. 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 [27, 123]. We did a preliminary investigation of these builds and found that these jobs with the same build ID normally share the same build result and build duration. Thus, as many existing papers have done [27, 46, 81], we considered these jobs as a single build.

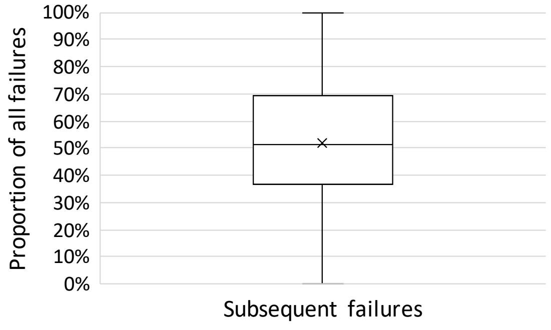
3.3 Evaluating our Motivating Hypotheses

We first evaluate our motivating hypotheses to understand if our approach to save build effort in CI is promising. Our first hypothesis posits that passing builds will be numerous — and thus skipping them would provide high build-effort savings in CI. We measured the ratio of passing builds to all builds in each studied project, and we show the distribution of such ratios in Figure 3.2a. From the result, the passing builds represented a very large proportion — with a median 88% (and a mean 84%) of all builds passing. This result supports our hypothesis that skipping passing builds would strongly save build effort in CI, since they generally represented a large portion of the executed builds. Furthermore, this result also shows the upper bound for how many builds could be saved — given a "perfect" technique that would correctly predict every single passing build.

Our next hypothesis posits that *subsequent failures* will be numerous — and thus predicting that subsequent builds to a failing build will also fail would correctly predict a substantial portion of failing builds. We measured the proportion of *subsequent failures* to *all failures* for each project (*e.g.,* in a build history **P-F-F-F-P**, the ratio of *subsequent failures* to *all failures* is 2/3). Figure 3.2b supports the hypothesis that *subsequent failures* are numerous, *i.e.,* there are many of them. A high number of projects had a high (*i.e.,* , not low) ratio of *subsequent failures*: >52% for 50% of projects, and >38% for >75% of projects. Thus, our approach would correctly predict a high proportion of *all build failures*, since we expect it to correctly predict all *subsequent failures.* Once it observes a failure, it would correctly predict all the subsequent ones.



(a) Proportion of passing builds



(b) Proportion of subsequent failures

Figure 3.2: Hypotheses evaluation.

3.4 Characterizing First Failures

We found that *subsequent failures* are numerous and easy to predict. Next, we will focus on predicting *first failures*. To inform our prediction technique, we perform a second empirical study to identify features that characterize them.

3.4.1 Research Method

We study two different kinds of features to characterize *first failures*: build features and project features. As build features, we selected all the features included in TravisTorrent that previous studies found to be correlated with *all build failures*, *e.g.,* [45, 80]. Our goal was to study whether such features are also correlated with *first failures*. Then, to be able to address the cold-start problem [111], we also created four project features that could be used for cross-project predictions. Our intuition is that project features would aid the classifier in "adapting" its trained model across projects of different characteristics — since projects using continuous integration are diverse [29]. To the extent of our knowledge, no previous work studied the correlation between *all build failures* (or *first failures*) and these project features (as defined by us, with a single value per project). We list in Table 3.1 the features that we studied, along with a brief description.

Build features will be useful to train our approach with past builds from the same software project. To identify build features that have a relationship with *first failures*, we first removed *subsequent failures* from our studied dataset. Then, we measured the correlation between the ratio of *first failures* to all builds (which now only included *first failures* and passing builds) and each studied build feature in each studied project. For each value of a build feature in a project, we measured the ratio of *first failures* to all builds that have that value for that feature in the project. For continuous features, such as src_churn, we use the Pearson correlation coefficient as effect size and its corresponding p-value for the significance test. For categorical variables, such as week_day, we measure effect size using Cramér's V and we use Pearson's $\mathcal{X}^2$ for the statistical significance test.

Project features will be useful to train our approach with past builds from other software projects. When no (or few) past builds are available for a software project, we could use past builds from different software projects to train our predictor. This situation is known in machine learning as the cold-start problem [111]. In such cases, our predictor will use project features to learn how representative past builds from other projects are for the project for which not enough past builds existed. As we did to study build features, we also removed *subsequent failures* to study project features. Then, we measured the correlation across projects between the value of each project feature and the project's ratio of *first failures* to all builds. Since all features were continuous, we applied Pearson's correlation coefficient and decided statistical significance for $p < 0.05$.

Table 3.1: Features studied for correlation with *first failures.*

Build features

Feature	**Short Description**
src_churn (SC)	The number of changed source lines since the last build.
file_churn (FC)	The number of changed source files since the last build.
test_churn (TC)	The number of changed test lines since the last build.
num_commit (NC)	The number of commits since the last build.
Project_performance_short (PS)	The proportion of passing builds in the recent five builds.
Project_performance_long (PL)	The proportion of passing builds in the whole previous builds.
Time_frequency (TF)	The time gap (hour) since the last build.
Failure_Distance (FD)	The number of builds since the last failing build.
Week_day (WD)	The weekday [0, 6] (0 being Monday) of the build.
Day_time (DT)	The time of day [0, 23] of the build.

Project features

Feature	**Short Description**
Team_size (TS)	The median number of developers over the project's CI usage history.
Project_size (PS)	The median number of executable production source lines of code in the repository over the project's CI usage history.
Project_age (PA)	The time duration between the first build and the last build for that project.
Test_density (TD)	The median number of lines in test cases per 1000 executable production source lines over the project's CI usage history.

3.4.2 Result

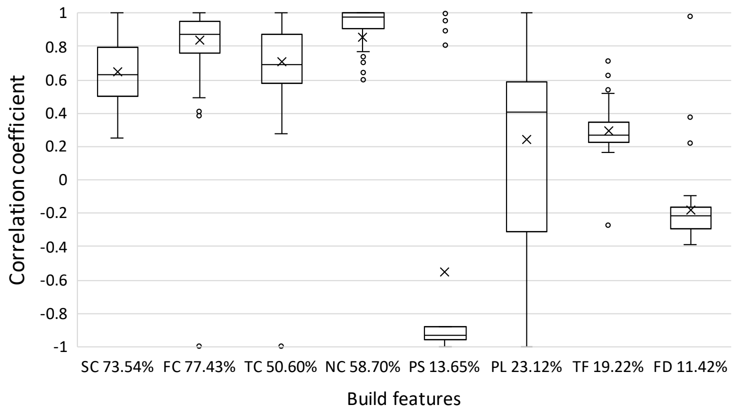
Build features. We show in Figure 3.3a the correlation between different build features and the ratio of *first failures*. Each box in the box plot represents the distribution of correlation coefficients between a feature (see Table 3.1) and the ratio of *first failures*, for all the projects for which that feature's correlation was statistically significant ($p < 0.05$). We report the percentage of projects for which a feature's correlation was statistically significant in its label in the X axis.

We observe that different build features were differently related to *first failures* For example, PS (project_performance_short) had a median correlation of -0.94, which means that the build was more likely to pass when there are more passing builds in its last five builds and it has a strong correlation. However, this correlation was only statistically significant in 13.65% of projects.

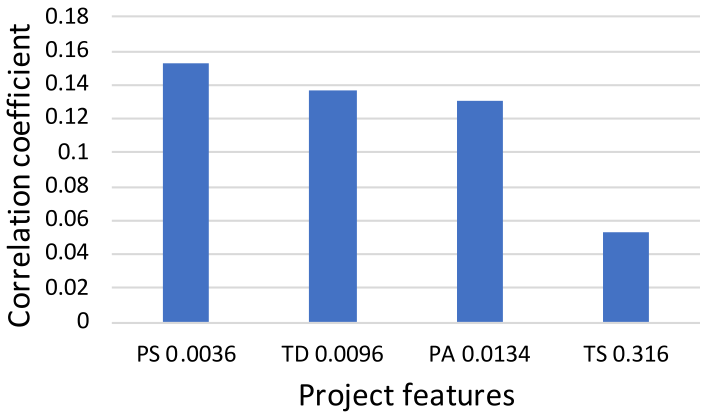
For the design of our technique, we will train on the features that had a strong correlation with the ratio of *first failures* and their results were statistically significant in at least 50% of projects. Four features had these characteristics, the numbers of: changed lines (SC), changed files (FC), changed test lines (TC), and commits since the last build (NC).

A clear implication of these build features being related to *first failures* is that, as changes accumulate in code — measured as any of these four build features — without a failing build being observed, the likelihood of the next build to fail becomes increasingly high. For the two categorical features (WD and DT), the results are statistically significant in only 10.36% and 12.32% of all projects, and their corresponding mean values of Cramér's V are 0.1308 and 0.2483.

Another interesting observation is that most of the build features that did not show strong statistical correlation with *first failures* are those that intuitively would be strongly correlated



(a) SC, FC, TC, NC had a statistically significant correlation for more than 50% of projects.



(b) The correlation was statistically significant for PS, TD, PA.

Figure 3.3: Correlation between (a) build (b) project features and ratio of first failures

with *subsequent failures* instead. That is, *subsequent failures* happen after a particularly short number (zero) of failing builds (FD), after a particularly low proportion of passing to failing builds (PS, PL), and probably a particularly short time after another build (TF). Intuitively, *first failures* would not particularly have any of these characteristics.

Project features. We use a bar chart to show each project feature and its corresponding correlation coefficient. The value following the name of each project feature represents its corresponding p-value. We found three project features for which *first failures* were more prevalent, *i.e.,* for which the project feature increased and its difference is statistically significant (Figure 3.3b): test density (TD), project size (PS), and project age in CI (PA). These are the features that we will use to design our technique to train across projects.

In simpler words, we observed that our studied projects had a larger ratio of *first failures* when they had larger test cases, more lines of code, or had been using CI for longer. This could mean that, as software projects mature, more bugs affect their builds and/or they get better at catching them. We posit that our observation is likely a combination of both phenomena — intuitively, larger projects have more points of failure and larger test suites are better at catching problems. Still, to understand the underlying causes of our observation in depth, further research would be necessary.

3.5 Evaluating Build-Failure Prediction

In our second empirical study, we discovered features that predict *first failures.* Next, we use them in SmartBuildSkip to evaluate it. We evaluate SmartBuildSkip in two experiments that complement each other. First, we evaluate its effectiveness for predicting build failures, and then we evaluate the cost reduction that its predictions provide in practice.

3.5.1 Research Method

We evaluate the prediction effectiveness of SmartBuildSkip in comparison to the state-of-the-art build-prediction technique: HW17 [38]. To better understand the benefit of SmartBuildSkip's two-stage design, we separately evaluate predictions for *first failures* and *all failures.* We evaluate both techniques over our dataset, and we measure their prediction effectiveness using precision, recall, and F1 score. We tested our results for statistical significance with a two-tailed Wilcoxon test, and decided statistical significance for $p < 0.05$.

To provide a point of reference for this evaluation, we replicated the state-of-the-art build prediction technique: **HW17** [38]. We use the acronym **HW17** to refer to it — the first letter of the authors' last names and its publication date — since the authors did not assign it a specific name. To the extent of our knowledge, **HW17** is the existing build prediction technique that provided the highest precision and recall.

We perform 8-fold cross validation, also to study the same conditions in which **HW17** was evaluated. Thus, we randomly divided our dataset into 8 subsets of builds, *i.e., folds*, iteratively using one of them as our test set and the remaining ones as our training set, until we have used every fold as test set.

SMARTBUILDSKIP-WITHIN: Our proposed approach described, trained in the same software project, using the predicting build features that we discovered.

SMARTBUILDSKIP-CROSS: Our proposed approach described, trained in other software projects, using the predicting build features and project features that we discovered.

HW17-WITHIN: The state-of-the-art build predictor, trained in the same software project.

HW17-CROSS: The state-of-the-art build predictor, trained in different software projects.

Dependent Variables. We used three metrics to evaluate our studied techniques: precision, recall, and F1 score. We calculated the value of these metrics for each studied software project, first for the set of *first failures*, and then for the set of *all failures*. We measured precision as the number of correctly predicted build failures divided by the number of builds that the technique predicted as build failures. We measured recall as the number of correctly predicted build failures divided by the number of actual build failures. We measured F1 score as the harmonic mean of precision and recall.

3.5.2 Result

We plot the results of this experiment in Figure 3.4 for the prediction of *first failures*, and in Figure 3.5 for the prediction of *all failures*. The boxes in these box plots for each dependent variable 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.

Predicting first failures. SmartBuildSkip improved HW17's median precision by 3% for its Within approach and by 9% for its Cross approach. SmartBuildSkip also improved HW17's median recall by 4% for its Within approach and by 7% for its Cross approach. These differences were statistically significant ($p < 0.05$). We posit that SmartBuildSkip-Cross provided an even higher improvement because its training set was much larger — encompassing multiple projects — and because build features likely vary little from project to project. These findings validate our hypothesis that separately predicting *first failures* is more effective than training a predictor based on features from *all failures*.

Predicting all failures. SmartBuildSkip improved HW17's median precision by 16% for its Within approach and was 9% worse for its Cross approach. It also improved HW17's median recall by 28% for its Within approach and by 68% for its Cross approach. These differences were statistically significant ($p < 0.05$). We posit that SmartBuildSkip's precision and recall are now much higher than HW17's because it is much better at predicting *subsequent failures*. We also observed that both techniques generally improved both their precision and recall. We believe that this is due to the increase in the number of failing builds in the dataset — after adding *subsequent failures*), allowing all techniques to learn them better. This is particularly acute for SmartBuildSkip's Cross variants, which became much more inclined to predict build failures after being trained with much more data (across projects), which

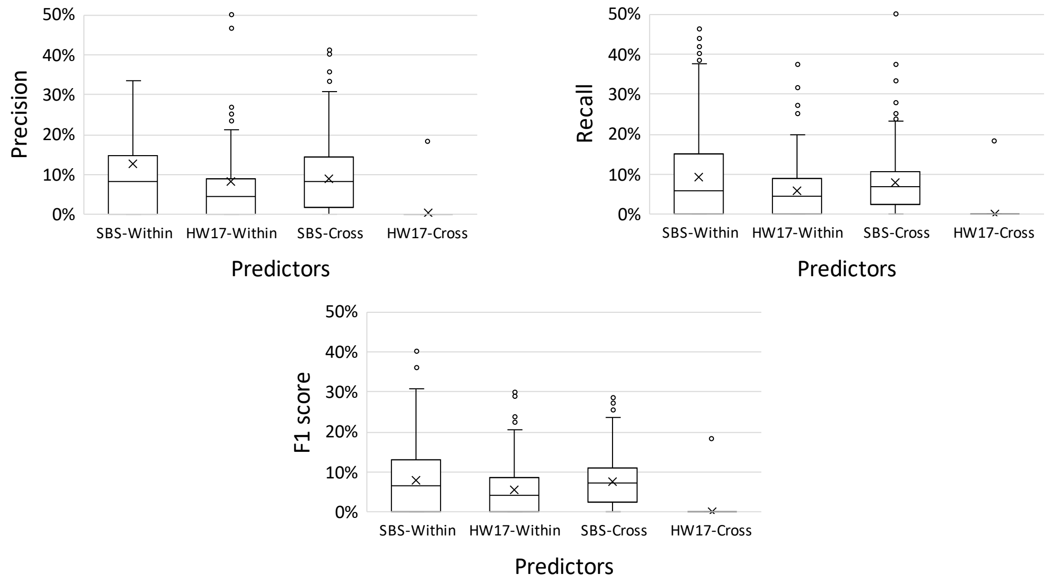


Figure 3.4: Performance comparison on predicting *first failures*

dramatically increased its recall, but reduced its precision. These findings also validate our hypothesis that choosing to always build after a failure is a highly successful strategy to predict *subsequent failures*.

3.6 Evaluating CI Cost Reduction

After finding that SmartBuildSkip improves the precision and recall of the state-of-the-art build predictor, we measure the cost reduction that it would provide in practice.

3.6.1 Research Method

We now simulate the more realistic scenario in which the builds that are skipped are not available for training. When a predictor predicts the upcoming build as a pass, we skip the build, and accumulate the value of the build-level features for the next coming build. We only update the information connected to the last build when the predictor actually decides

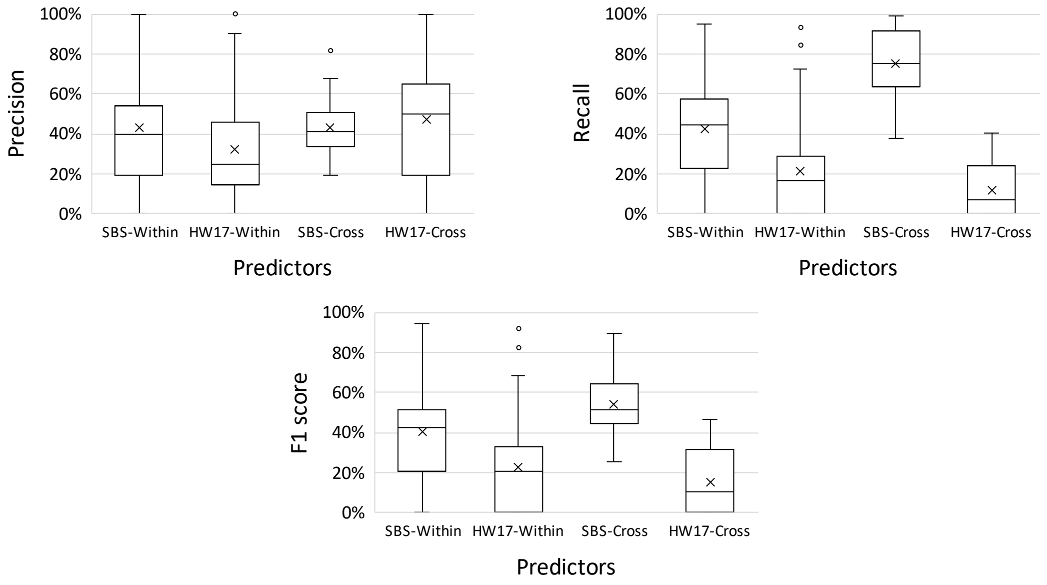


Figure 3.5: Performance comparison on predicting *all failures*

to build. In this context, we measure four metrics for each evaluated technique: how many builds it saves, how many failing builds are observed immediately (and how many with a delay), the delay length of delayed failing builds, and a new metric to measure the balance between failing build observation delay and build execution saving.

Independent Variable: Technique. We evaluate the same four predictors as in Experiment 1, in addition to a new collection of techniques that we call *rule-of-thumb techniques.* In the spirit of cost-saving, we propose this additional collection of techniques because of their low adoption cost. These rule-of-thumb techniques are based on the individual build features. They simply decide to skip builds when the given feature value is below a certain threshold. We propose these techniques as a potentially "good-enough" alternative for software teams that do not have the resources to implement and adopt SmartBuildSkip, or for them to use in the time period while they are implementing it. Finally, we also include a "Perfect" technique that would skip all passing builds and run all failing builds — as a reference for how many builds could be desirably skipped.

Independent Variable: Prediction sensitivity. Our simple techniques need a threshold to be applied, i.e., they are defined as *"predict build failures when the feature value is over X"*. In a similar manner, SmartBuildSkip can be also configured for different thresholds of prediction sensitivity. Thus, we also evaluate these techniques for multiple thresholds of sensitivity. Only when the possibility predicted by the classifier for the coming build to become a failure is smaller than the threshold, we will predict the build as a pass, which means the smaller the threshold is, the easier we are going to predict builds as failing. Finally, these varied thresholds and prediction sensitivities will allow us to learn different trade-offs that could be achieved in terms of saving cost in CI — skipping builds — without losing too much value — without delaying too many build failures. We evaluated 50 different thresholds (values 1–50), which meant: absolute value for the "rule-of-thumb" techniques, and predicted likelihood (in percentage) of the build to fail for SmartBuildSkip.

Dependent Variables. We measured four metrics in this evaluation: Recall, Failing-build Delay, Saved Builds, and Saving Efficiency. *Recall* is the proportion of failing builds that are correctly predicted and executed, among all failing builds. For each failing build that was incorrectly predicted and skipped, we also measured its *Failing-build Delay*, as the number of builds that were skipped until the predictor decided to run a build again — and then the failure would be observed. We measured *Saved Builds* as the proportion of builds that are skipped among all builds. Finally, we measured *Saving Efficiency* as the harmonic mean of saved-builds and recall, to understand their balance.

3.6.2 Result

We plot the results for our Experiment. Figure 3.6 shows the median value for each metric across studied projects. For Failing-build Delay, it's the median across projects of their median Failing-build Delay. The Y axis is the metric for evaluation and each box contains every project's result. The X axis has different meanings for different techniques: the threshold for rule-of-thumb techniques (e.g., threshold 5 for #src_files means that <5 files were changed in that build), or the prediction sensitivity (in percentage) for the predictors.

We make a few observations from our results. First, SmartBuildSkip-Within achieves the peak *saving efficiency* among all techniques for its 2% sensitivity — saving 61% of all builds, executing 73% of the failing builds immediately, and the remaining ones with a median 2-build delay. If a more conservative approach is sought, SmartBuildSkip-Within's 0% sensitivity would execute 80% of the failing builds (and the remaining ones with a 1-build delay), while still saving 45% of all builds.

HW17 achieved the poorest saving efficiency. As we anticipated in the motivation, HW17 predicted most builds to pass because it relied too much on the status of the last build. It saved a large amount of builds, but it also executed very few failing builds as a result.

Finally, our rule-of-thumb techniques provided acceptable results. Thus, a software team looking for a simple mechanism to save effort by skipping builds in CI could simply skip those builds that, for example, changed more than 30 lines — which is the highest saving efficiency for *#src-lines*. In our experiments, this threshold saved around 57% builds, executing 60% failing builds (and the remaining ones with an 8-build delay). While this trade-off may not be the most ideal (certainly SmartBuildSkip provides much better trade-offs), it has the advantage that it can be adopted by simply informing developers to follow that rule.

Finally, if more conservative or more risky approaches are preferred, Figure 3.6 shows a wide variety of trade-offs that could be achieved by different techniques and configurations.

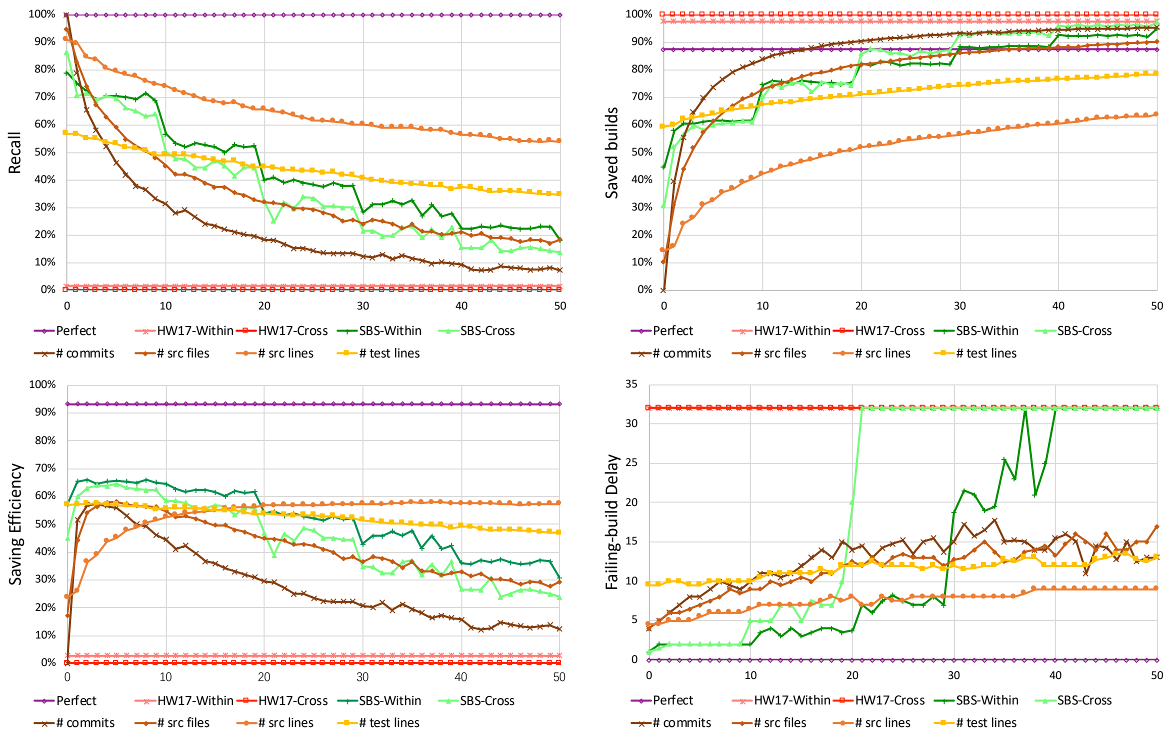


Figure 3.6: Cost saved and value kept by evaluated techniques

3.7 Discussion

Diverse Cost-saving Needs. Different developers will have different preferences in the trade-off between observing failing builds early and saving build effort. Thus, we propose SMARTBUILDSKIP as a customizable solution, with an adjustable prediction sensitivity. Some developers may value observing failing builds early much more than saving cost (but still want to save some cost), *e.g.,* developers at large companies that have been using CI for some time and are exploring ways to reduce its cost (like Facebook [67], Microsoft [41], or Google [19]). These developers could configure SMARTBUILDSKIP in its most conservative sensitivity (0) and save the cost of 30% of their builds while only introducing a 1-build delay in 15% of their build failures.

In contrast, other developers may be looking for a way to reduce CI's high-cost barrier [110] to adopt it, even if it means observing build failures less quickly. These developers could configure SmartBuildSkip with a more liberal sensitivity (2) and save the cost of 61% of their builds and still observe 73% failing builds with no delay (and the remaining 27% with a 2-build delay). In this scenario, SmartBuildSkip dramatically lowers the cost of CI for non-adopters, letting them still get a strong value from it — particularly considering that non-adopters currently do not benefit from CI at all. Furthermore, as developers' budgets increase, they could also adapt the sensitivity of SmartBuildSkip over time to build more and observe failures more quickly.

The Impact of Delayed Failing Builds. Our approach reduces the cost of CI, but it also reduces its value — it delays the observation of some build failures. Some existing techniques target developers who cannot afford a single delayed failing build — by skipping only tests [97] or commits [2] that are guaranteed to pass, *i.e.,* tests for other modules and non-code changes. In exchange for such guarantee, this strategy is limited in how much cost it can save — the number of guaranteed-pass tests and commits.

Our proposed technique targets developers for whom some delay in failure observation is acceptable — as do existing techniques based on test selection. Such techniques, which introduce failure observation delays, are valued and adopted by many large software companies, *e.g.,* Google [18], Microsoft [41], or Facebook [67]. We argue that, for many developers, the cost savings provided by SmartBuildSkip overcome the introduced delay in failure observation — particularly for SmartBuildSkip's most conservative sensitivities, which produce a delay of one or two builds. For context, Herzig *et al.* 's approach [41] (deployed at Microsoft) introduced a delay of 1–3 builds. Ultimately, though, we believe that different developers would prefer different cost-saving trade-offs, which is why we made SmartBuildSkip customizable.

Other Purposes of CI. The main reason for developers to use CI is to *catch bugs earlier* [43], but they also use it to: have *a common build environment*, *make integrations easier*, *enforce a specific workflow*, simplify *testing across multiple platforms*, be *less worried about breaking builds*, *deploy more often*, and have *faster iterations*, [43, 44]. Most (the first four) of these purposes are achieved as soon as CI is adopted, so we do not expect them to be impacted by introducing a cost-saving technique like SmartBuildSkip. However, the last three purposes (and others like safety-checking pull requests) may be impacted, since they benefit from observing build passes. This applies to both our and existing techniques that skip tests or builds.

Still, after adopting a cost-saving technique, developers remain in control of their build frequency. They can always build more frequently by making SmartBuildSkip's prediction sensitivity more conservative, or by simply triggering additional builds on top of the ones that SmartBuildSkip triggers.

Furthermore, SmartBuildSkip provides an additional benefit over existing test-selection-based techniques for purposes that rely on build observations. Test-selection techniques may give a *false sense of confidence* [77] when a build that should have failed instead passes because some of its failing tests were skipped.

When SmartBuildSkip predicts a build that should have failed as passing, it skips it (it does not show it as passing), which provides more transparency about the unknown status of the build — until it eventually fails in a later build.

3.8 Conclusions

In this chapter, we proposed and evaluated SmartBuildSkip, a novel framework for saving cost in CI by skipping builds that it predicts will pass. Our design of SmartBuildSkip is based on two main hypothesis: that build passes are numerous and that many failing builds happen consecutively. We studied these hypotheses and found evidence to support them. Thus, SmartBuildSkip works in two phases: first it runs a machine learning predictor to decide if a build will pass — and skips it — or will fail — and executes it. Whenever it observes a failing build, it determines that all subsequent builds will fail and keeps building until it observes a pass again — and starts predicting again.

With this strategy, SmartBuildSkip improved the precision and recall of the state-of-the-art build predictor (HW17) and cost savings with various trade-offs, since we made it customizable to address the needs of diverse populations of developers. We highlight two specific configurations that we posit will be popular: the most conservative one, which saves 30% builds and only delays the observation of 15% failing builds by 1 build; and a more balanced one that saves 61% of all builds and delays 27% failing builds by 2 builds. Nevertheless, SmartBuildSkip provides many other trade-offs that could be desirable in different environments. SmartBuildSkip provides a novel strategy that complements existing techniques to cost saving in CI that focus on skipping test cases or builds with non-code changes.

Chapter 4

Evaluating CI-improving techniques

We introduced our newly designed technique, SmartBuildSkip in Chapter 3. Given that there are plenty of other techniques that can improve CI, we aim to evaluate their performance with SmartBuildSkip under the same context to compare their strengths and weaknesses.

As software companies adopt CI, they execute builds for many of projects, and they do so very frequently. As workload increases, two main problems appear: (1) the time to receive feedback from the build process increases, as software builds often outnumber the available computational resources — having to wait in build queues, and (2) the computational cost of running builds also becomes very high. Previous studies *e.g.,* [69] have highlighted the long time that developers have to wait to receive feedback about their builds. For example, at Google, developers must wait 45 minutes to 9 hours to receive testing results [63]. Even just the dependency-retrieval step of CI can take up to an hour per build [11]. Regarding the high cost of running builds, that is also highlighted in other studies [41, 43, 44, 77, 110]. The cost of CI reaches millions of dollars, *e.g.,* at Google [44] and Microsoft [41]. While other problems exist for CI, we focus on these two because they are the ones that most existing techniques have focused on addressing. They are also interrelated, since cost-reduction techniques may also reduce time-to-feedback — *e.g.,* skipping some tests may cause other tests to fail earlier.

Multiple techniques have been proposed to improve CI. Most of them have the goal of reducing either its **time-to-feedback** or its **computational cost**. All such techniques consider the observation of build failures to be more valuable than build passes, because failures provide actionable feedback, *i.e.,* they point to a problem that needs to be addressed. **Time-to-feedback-reduction** techniques aim to observe **failures earlier** — by **prioritizing** failing executions over passing ones. These techniques may operate in two different levels of granularity, by prioritizing: test executions *e.g.,* [19], or build executions *e.g.,* [63]. **Computational-cost-reduction** techniques aim to observe **failures only** — by **selectively executing** failing builds only, saving the cost of executing passing ones. They also may operate at two different levels of granularity, selecting: test executions *e.g.,* [67], or build executions *e.g.,* [2].

To the extent of our knowledge, the existing techniques to improve CI have been evaluated under different settings, making it hard to compare them. Previous studies used different software projects, different metrics, and rarely compared one technique to another. However, we expect that different choices of goal, granularity, and technique design will bring different trade-offs. For example, cost-reduction techniques at build-granularity may be more *risky* than a test-granularity one, *i.e.,* it may save more cost when it skips all the tests in a build, but it may also make more mistakes if it skips many failing tests in a build. However, the opposite may be true, if test-granularity cost-reduction techniques also skip a large ratio of full builds (*i.e.,* all the tests in the build). On another example, test-selection techniques may be a good alternative to test-prioritization techniques that also saves cost as an added benefit, or they may instead delay the observation of test failures if they mispredict too many of them. To the best of our knowledge, how these trade-offs manifest in practice is still mostly unknown. Empirically understanding these trade-offs will have valuable practical implications for the design of future techniques and for practitioners adopting them.

In this chapter, we perform the first evaluation of the existing strategies to improve CI. We aim to understand the trade-offs between these techniques for three dimensions: (D1) computational-cost reduction, (D2) missed failure observation, and (D3) early feedback.

For this goal, we performed a large-scale evaluation. We replicated and evaluated all the existing 10 CI-improving techniques from the research literature, representing the two goals (time-to-feedback and computational-cost reduction) and the two levels of granularity (build-level and test-level) for which such techniques have been proposed. We evaluated these techniques under the same settings, using the state-of-the-art dataset of continuous-integration data: TravisTorrent [8]. To be able to study all techniques, we extended TravisTorrent in multiple ways, mining additional Travis logs, Github commits, and building dependency graphs for all our studied projects. Finally, we measured the effectiveness of all techniques with 10 metrics in 3 dimensions. We included every metric that any previous evaluation of our studied techniques used (7), refitted 2 others and designed an additional one.

We analyzed the results obtained by all techniques on all metrics across all 3 dimensions, and we synthesized our observations, to understand which design decisions helped and which ones did not for each dimension. We also extended our experiments as a synthetic study that simulates these studied techniques on different hypothetical projects whose tests within one build occupy different proportions of the whole build process to show the generalizability of our study. Finally, we further reflect on our results to provide a wide set of recommendations for the design of future techniques in this research area.

The main contributions of this work are: (1) the first comprehensive evaluation of CI-improving techniques; (2) a collection of metrics to measure the performance of CI-improving techniques over various dimensions; (3) an extended Travis Torrent dataset with: detailed test and commit, and dependencies information; (4) the replication of 14 variants of 10 CI-improving techniques; (5) evidence for researchers to design future CI-improving techniques.

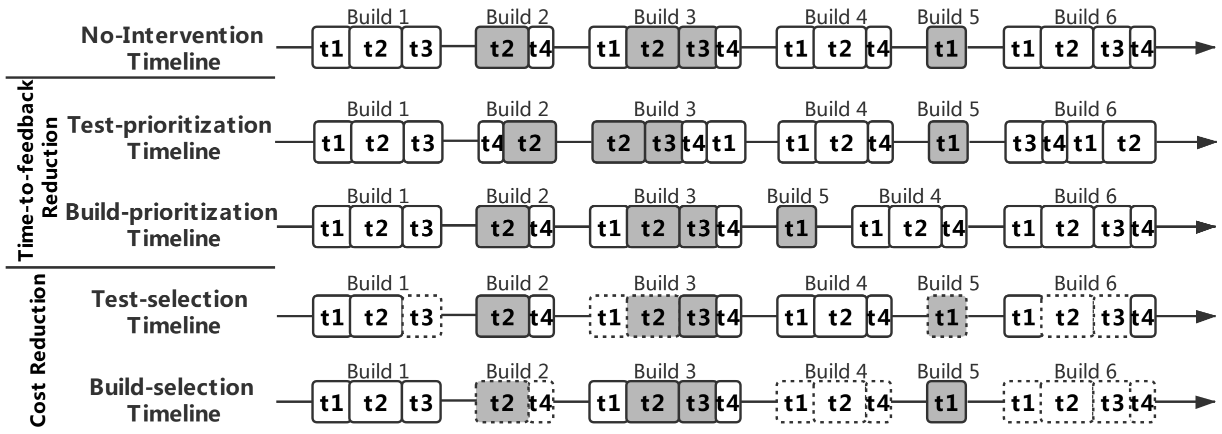


Figure 4.1: Example timeline. Failing tests in gray. Build-selection runs builds fully when it predicts a failing build. Test-selection runs builds partially (for tests that would fail). Build-prioritization changes the build sequence. Test-prioritization changes the test sequence within a build.

4.1 Approaches to Improve Continuous Integration

We summarize technique families in Table 4.1 and discuss each technique in detail in §4.2.3. Figure 4.1 depicts a non-interventional example timeline of builds, a timeline in which a build-selection technique is applied, a timeline produced by build-prioritization technique, a timeline where a test-selection technique is applied, and a timeline with applying a test-prioritization approach. The example timeline shows a chronological numbered sequence of builds in CI. Each build is made up of at least one test. We depict each test suite as a rectangle with a test number (e.g., t1). Failing tests are then highlighted in gray. The length of the rectangle refers to the time duration for the test to be executed. We depict skipped tests with a dashed rectangle. In the most ideal cost-saving scenario, all of the passing tests would be skipped and all of the failing tests would be observed as soon as possible.

4.1.1 Computational-cost Reduction

Test-level granularity

Test-selection techniques [32, 41, 67, 69, 97, 118, 122] aim at automatically detect and label tests that are not going to fail. These test-level approaches collect information from test history and project dependency along with the current commit and use some heuristic models to detect failing tests and skip the others. Figure 4.1 also illustrates how this type of techniques works in the simulation timeline. After a test-selection approach is activated, it selects a subset of tests (e.g., t2 in build #2, t4 in build #4) that it predicts to have a possibility to fail and decides to skip the others (e.g., t3 in build #1, t1 in build #5). For those tests that are not selected in the timeline and get skipped, we depict them as dashed rectangles. In this work, we consider it can skip some builds when it selects no test in those builds.

Build-level granularity

Build-selection techniques [1, 2, 38, 51, 73] aim at automatically detect and label commits and builds that can be CI skipped. Some approaches [38, 51, 73] try to detect failing builds and skip those passing builds to achieve cost-saving. Others [1, 2] aim at identifying commits that can be CI skipped. Figure 4.1 illustrates how they work in the simulation timeline. As a build-level technique, when build-selection approach decides to skip a build (e.g., build #2, #4, #6), normally it skips all of the tests in that build. The inner test sequence is not changed and all of tests are run in an executed build.

4.1.2 Time-to-feedback Reduction

Test-level granularity

Test-prioritization techniques [19, 66, 68, 71, 102] try to give high priority to tests that are predicted to be failed so that developers could be informed in a shorter time. This family of approaches normally rearrange the execution order of tests within a build to make predicted-to-fail tests run earlier by analyzing information such as test failing history and test context. Figure 4.1 depicts an example of how this type of techniques works in the simulation timeline. With a test-level approach being activated, the CI system gives different tests different priorities and firstly executes those tests with a higher priority (e.g., t4 in build #2, t2 in build #3) as well as delays low-priority tests (e.g., t1 in build #3, t2 in build #6). The sequence of test executions in this timeline gets rearranged and the start-time for tests that are more likely to fail move ahead in time. Also, all tests are executed at last.

Build-level granularity

Build-prioritization techniques [63] aim at automatically prioritizes commits that are waiting for being executed. They favor builds with a larger percentage of test suites that have been found to fail recently and builds including test suites that have not been executed recently as an alternative path. Figure 4.1 also shows how this family of techniques works in the simulation timeline. Build-prioritization techniques will only be activated when there is a collision of builds (i.e., there are multiple builds waiting to occupy the limited resource). The technique is build-level so it will not change the inner order of the test executions and it will normally change the sequence of tests across builds when the approach is activated (e.g., build #4, #5). None of tests become dashed in this timeline because they all eventually execute.

4.2 Research Method

In this work, we replicated and evaluated 14 variants of 10 CI-improving techniques, covering their two goals (time-to-feedback and computational-cost reduction) and their two levels of granularity (build-level and test-level) with 1 perfect technique for the ideal timeline. We evaluate them over 100 software projects in TravisTorrent, which we extended to be able to run all such kinds of techniques.

Our goal is to understand the trade-offs between existing CI-improving techniques, and between the metrics that have been used to evaluate them. We perform 2 empirical studies to analyze these trade-offs for the following 3 dimensions of CI-improving techniques, using 10 metrics. We only include selection techniques in Empirical Study 1 since prioritization techniques have no power in cost saving by nature. We involve selection and prioritization techniques in Empirical Study 2 because both of them can have an impact on fault detection, *e.g.,* wrongly-skipped failing builds by selection approaches can cause delay in fault detection.

Empirical Study 1: Cost Saving

D1: Computational-cost Reduction

D2: Missed Failure Observation

Empirical Study 2: Time-to-feedback Reduction

D3: Early Feedback

For each dimension, we study:

RQ1: What design decisions helped this dimension?

RQ2: What design decisions did not help this dimension?

4.2.1 Data Set

We perform our study over the Travis Torrent dataset [6], which includes 1,359 projects (402 Java projects and 898 Ruby projects) with data for 2,640,825 build instances. 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 [45, 73]. To be able to evaluate test-granularity techniques, 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 result, rather than error or canceled — since they can be exceptions or may happen during the initialization and get aborted immediately before the real build starts. 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. We did a preliminary investigation of these builds and found that these jobs with the same build identifier normally share the same build result and build duration. Thus, as many existing papers have done [27, 46, 81], we considered these jobs as a single build. After this filtering process, we obtained 82,427 builds from 100 projects (13,464 failing builds).

To be able to execute all our studied techniques, we extended the information in TravisTorrent of these 100 projects in multiple ways. First of all, we needed to know the duration of each individual test for the comparison and replication. Also, to replicate some techniques, *e.g.,* [19, 41], we needed to capture the historical failure ratio for each individual test. To obtain these information, we built 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. Some techniques, *e.g.,* [2, 67], require 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. Then, we matched each test with its corresponding test file in the project. Finally, to be able to run other techniques, *e.g.,* [32, 67], we built a dependency graph for the source code of each project using a static code analysis tool (Scitool Understand [88]) to determine the paths between the source files and test files.

4.2.2 Evaluation Process

We evaluate the techniques in a real-world scenario, to understand as best as possible the behavior that the techniques would show in practice. We take two measures for that.

First, we respect the original chronological order of build and test operations when training techniques. We achieve that by using an 11-fold, chronological variant of cross-validation. For each project, we split its chronological timeline into 11 folds. We use the first chronological fold only for testing, and we iteratively test the other 10 folds. For each testing fold, we train on all the folds that precede it chronologically. This approach has been used in previous works *e.g.,* [9, 93] to avoid training with information that would not be available in practice, *i.e.,* it happens in the future.

We follow this approach for all the techniques based on machine learning, *e.g.,* [67]. For techniques that do not require training, *e.g.,* [2], we simply execute them over the same last 10 folds. For techniques that train on data from other projects, *i.e.,* for cross-project technique variants, we also executed them over the same last-10-fold timeline — and we divided them into 10 *project* folds to do cross-project cross-validation, *i.e.,* for each project, the technique is trained on 90 other projects and tested on its last 10 fold data.

Second, we respect the real-world availability of information. That is, for selection-based techniques, when a build or test is skipped, the technique will not know its outcome. For techniques that rely on the last build or test outcome *e.g.,* [37], we only inform them of the outcome of the last *executed* build or test. Additionally, when builds are skipped, we

Table 4.1: Studied Techniques.

Goal	Approach	Granularity	Studied Technique
Time to Feedback	Prioritization	Test	PT_Marijan13 [68]
			PT_Elbaum14 [19]
			PT_Thomas14 [102]
		Build	PB_Liang18 [62]
Computational Cost	Selection	Test	ST_Gligoric15 [32]
			ST_Herzig15 [41]
			ST_Mach19 [67]
		Build	SB_Hassan17 [38]
			SB_Abd19 [2]
			SB_Jin20 [51]

accumulate their code changes into the subsequent build.

4.2.3 Replicated Techniques

We replicated and studied all the techniques that have been proposed to improve CI by reducing the time to feedback or reducing its cost. In addition to these, there are other techniques that were proposed before CI and that could also be applied for these two goals: test prioritization techniques, and test selection techniques. Therefore, we also replicated and studied a state-of-the-art technique in each of these two categories that were not originally proposed for CI. We summarize all our studied techniques in Table 4.1.

In total, we studied 10 techniques, across two goals (reducing time to feedback and cost) and two granularities (test and build levels). Since we also studied multiple variants of some techniques, our evaluation included 14 total technique variants. To provide a reference point, we also studied a perfect technique: *Perfect Technique*. It achieves the goal of each metric perfectly — it predicts which tests or builds will fail with 100% accuracy, prioritizing or selecting them perfectly.

We include the detailed description for each technique in §4.3.1 and §4.4.1.

4.3 Empirical Study 1: Cost Saving

4.3.1 Studied Techniques

Test-selection Techniques

We replicated all the test-selection techniques that were proposed for improving CI: ST_Mach19 [67] and ST_Herzig15 [41]. To provide even more context for our study, we also evaluate a state-of-the-art test-selection technique: ST_Gligoric15 [32] — since test-selection techniques have also been proposed outside the context of CI, *e.g.,* [32, 83, 84, 114, 115, 118].

ST_Gligoric15 [32] skips tests that cannot reach the changed files, by tracking dynamic dependencies of tests on files. A test can be skipped in the new revision if none of its dependent files changed. The rationale is that tests that cannot reach changed files cannot detect faults in them.

ST_Herzig15 [41] is based on a cost model, which dynamically skips tests when the expected cost of running the test exceeds the expected cost of removing it, considering both the machine cost and human inspection cost [5, 42]. This technique tends to skip tests that mostly passed in the past or that have long runtime.

ST_Mach19 [67] proposes a Machine Learning algorithm with combined features of commit changes and test historical information. We studied two variants of it: one is trained in the past builds within the same project in which it is applied (*ST_Mach19_W*), and the other is trained in the builds of different software projects than the one in which it will be applied (*ST_Mach19_C*). It uses the following features: file extensions, change history, failure rates, project name, number of tests and minimal distance.

Build-selection Techniques

We then replicated all build-selection techniques that have been proposed for improving CI: SB_Abd19 [2], and SB_Jin20 [51]. To provide even more context for our study, we also replicated a state-of-the-art build-prediction technique: SB_Hassan17 [38].

SB_Hassan17 [**38**] predicts every build's outcome based on the information from last build. Builds can be skipped when they are predicted to pass. In our study, information from the previous build is blinded if the build does not get executed. We study two variants of this technique (*SB_Hassan17_W* and *SB_Hassan17_C*) as we did for *ST_Mach19*.

SB_Abd19 [**2**] uses a rule-based approach to skip commits that only have *safe* changes, *e.g.,* changes on configuration or document files. This technique is expected to capture most failing builds since it only skips builds considered safe to skip.

SB_Jin20 [**51**] aims at saving CI cost by skipping passing builds. Their strategy is to capture the first failing build in a subsequence of failing builds and continuously build until a passing build appears. We replicated this technique under the configuration that provided the optimal effectiveness [51]. We studied three variants of this technique: *SB_Jin20_W* & *SB_Jin20_C* as we did previously, and also a rule-of-thumb variant (SB_Jin20_S) that skips builds with < 4 changed files.

4.3.2 D1: Computational-cost Reduction

We studied four metrics for D1. We plot the result of each metric in a box plot where each box represents the distribution of values for all the studied projects.

Studied Metrics

Build time saved measures the proportion of total build time that is skipped among all build time per project. It was covered in SB_Abd19 [2].

Test time saved measures the same as the previous metric but in terms of test time. The previous work ST_Gligoric15 [32] used this metric in its evaluation. It shows how much time applying a technique could save during the phase of test executions.

Builds number saved measures the proportion of builds that are saved among all builds. It was studied by SB_Abd19 [2] and SB_Jin20 [51]. It represents how many resources could be saved as the number of builds.

Tests number saved measures the same as the previous metric but in term of tests. Previous papers [32, 41] studied this metric. It represents how many resources could be saved during test executions.

Analysis of Results

We plot the results for all techniques for this metric in Figure 4.2.

Comparing Metrics. When we compare the techniques' test number vs. test time saved, most of them saved a very similar ratio of test time than ratio of tests (except ST_Herzig15). When comparing build number vs. build time, build-granularity techniques saved a very similar ratio of build time as of builds. Also, test-granularity techniques saved a larger ratio of build time than of builds. This means that test-granularity techniques save build time when they skip builds partially — when they skipped some of their tests. When comparing test number vs. build number, build-granularity techniques saved a very similar ratio of builds and tests. Also, test-granularity techniques saved a much lower ratio of builds than of tests — some dramatically so (ST_Herzig15 and ST_Mach19_C). This means that test-granularity techniques saved a low ratio of full builds. When comparing test time vs. build time, build-granularity techniques saved very similar ratios of test time and build time. Also, test-granularity techniques saved a much lower ratio of build time than of test time. This observation extends our earlier one: every build that these techniques did not skip fully, and thus did not skip its build-preparation time, reduced their ability to save build time to an important extent.

Comparing Granularities. By comparing test vs. build-granularity techniques, build-granularity techniques generally saved higher build-time cost — except for SB_Abd19. Build-granularity techniques have the advantage of skipping both test-execution and build-preparation time, while test-granularity techniques have the advantage of skipping tests spread over many builds, not only on those that get fully skipped. Our observation implies that skipping full builds was a better strategy for saving cost.

Comparing Techniques. We first observed that SB_Mach19_C and SB_Jin20_C skipped fewer builds than their counterparts that were trained only with data within the same project (SB_Mach19_W, SB_Jin20_W). After having been trained with a more diverse set of build and tests (across many projects), these techniques became less confident to skip them. ST_Herzig15 saved very low ratio of build time despite saving a large ratio of tests. This is because it very rarely skips tests that failed many times in the past — regardless of the code changes in the build. So, within each build, it very rarely skipped the tests with the most past failures — thus very rarely skipping builds fully. SB_Abd19 saved a median 21% build time, which is a relatively high amount, considering that it only skipped builds with non-executable changes, *e.g.,* that only changed formatting or comments. ST_Mach19_W and ST_Gligoric15 skipped a relatively high ratio of build time (competitively with build selection techniques) because they skipped many full builds. This is because they analyze

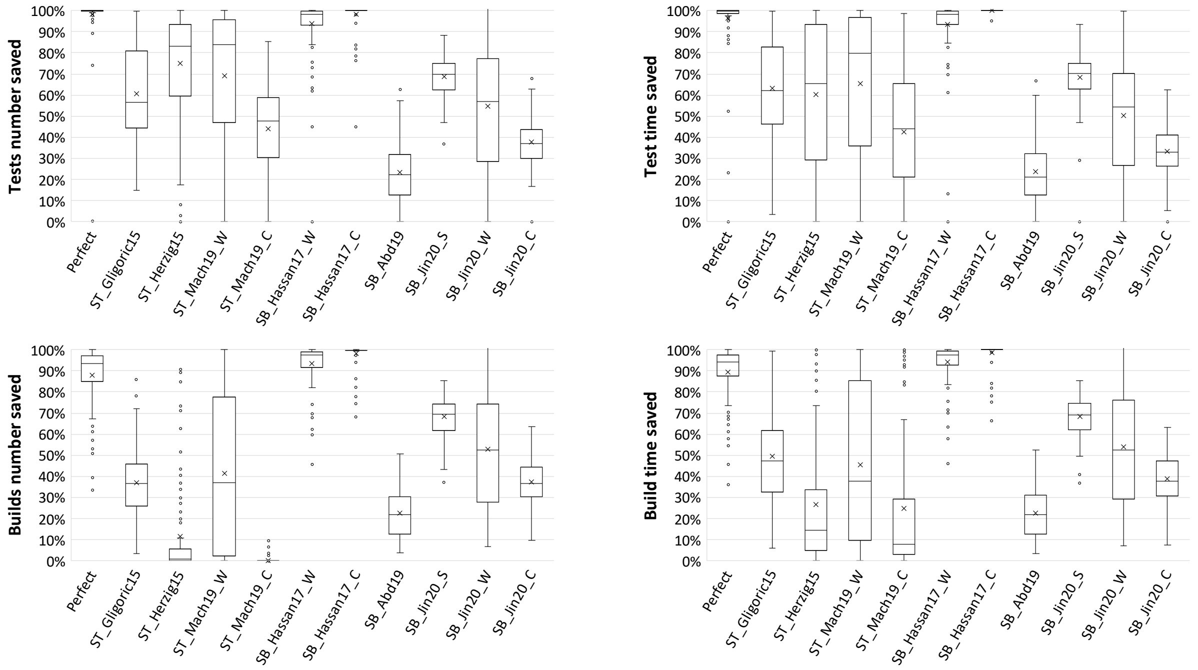


Figure 4.2: Results for Cost Saving Metrics. Prioritization techniques not included, since they do not skip tests/builds.

the relationship between code changes and tests inside a build. ST_Gligoric15 skips all tests that cannot execute the code changes, and ST_Mach19_W considers the distance between the changes and the tests in its predictor. This allows both techniques to fully skip those builds in which no test can execute the code changes — *i.e.,* when only non-executable code was changed, or when no tests exist to execute the changes. SB_Jin20_W and SB_Jin20_S saved high ratios of build time, since they both focused on skipping full builds. While SB_Jin20_S provided higher savings, we expect it to also skip a higher ratio of skipped failing builds (see §4.3.3) — SB_Jin20_S simply skips builds with <4 commits. Finally, SB_Hassan17_W and SB_Hassan17_C skipped too much build time (higher than the perfect baseline). This is because they mostly rely on the status of the previous build, which is unknown if skipped. So, as soon as they observe a passing build, they recurrently skip all subsequent builds.

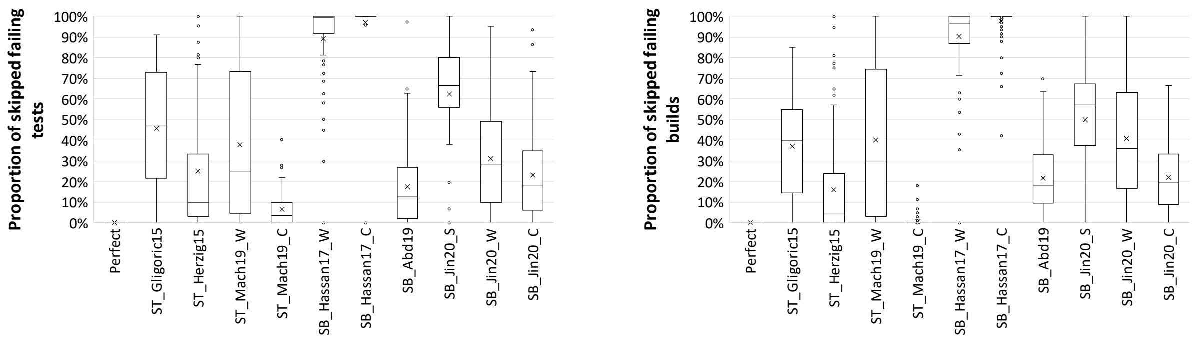


Figure 4.3: Results for Missed Failure Observation Metrics. Prioritization techniques not included, since they do not skip tests/builds.

4.3.3 D2: Missed Failure Observation

Studied Metrics

Proportion of skipped failing tests. This metric measures the undesired side effect of cost-saving techniques skipping some of the failing test cases. It was used by ST_Herzig15 [41].

Proportion of skipped failing builds. This metric measures the proportion of failing builds that are skipped among all failing builds. It was covered in SB_Jin20 [51].

Analysis of Results

We plot the results for all techniques for this metric in Figure 4.3.

Comparing Metrics. All techniques generally skipped a very similar ratio of failing tests than builds, with small differences.

ST_Mach19_C, ST_Herzig15, ST_Gligoric15, SB_Jin20_S skipped a slightly higher ratio of failing tests than builds. This is explained by test-granularity techniques skipping partial builds in addition to full builds, and thus they also skipped a higher ratio of failing tests.

The case of SB_Jin20_S is different: it skipped a higher ratio of tests because it skipped fewer builds with no failing tests — few changed < 4 files.

SB_Abd19, SB_Jin20_C, ST_Mach19_W and SB_Jin20_W skipped a slightly higher ratio of failing builds than tests. This means that these techniques skipped failing builds with lower than average (or no) failing tests, *e.g.,* failing due to configuration or compilation errors (which amount to 35% of failing builds). Finally, SB_Hassan17_C and SB_Hassan17_W skipped most failing (and passing) tests and builds.

Comparing Granularities. Build-granularity techniques generally skipped higher ratios of failing builds and tests than test-granularity techniques — except for SB_Abd19. They generally skipped a higher ratio of all tests and builds.

Comparing Techniques. If we rank techniques on these two metrics of side-effect, we observe that they rank almost exactly in the opposite order as they would according to build time saved (for D1). This shows a clear trade-off between cost-saving and its side effect of skipping failures.

4.4 Empirical Study 2. D3: Time-to-feedback Reduction

In D3, we study how much prioritization techniques advance the observation of failures and how much the side effect in D2 will influence it. So, we study all the time-to-feedback and computational-cost reduction techniques.

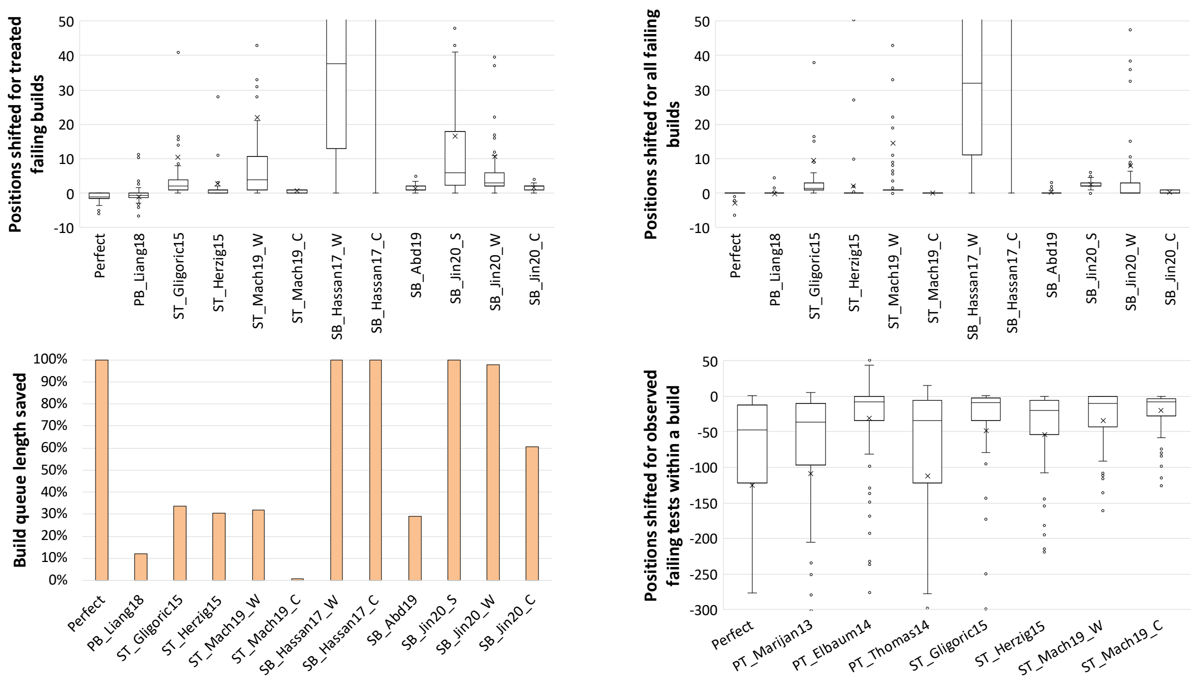


Figure 4.4: Results for Time-to-feedback Reduction Metrics.

4.4.1 Studied Techniques

We only describe here the techniques that we did not describe in earlier sections: prioritization techniques.

Test-prioritization Techniques

For this family of techniques, we replicated all the test-prioritization techniques that were proposed for improving CI: PT_Elbaum14 [19] and PT_Marijan13 [68]. To further extend this study, we also replicated the state-of-the-art test case prioritization (TCP) technique. We chose the technique that provided the highest effectiveness in the most recent evaluation of TCP techniques [66]: PT_Thomas14 [102]. TCP was a rich research area before CI became a common practice, *e.g.,* [18, 71, 85, 102]. We apply these techniques to prioritize tests within each build.

PT_Marijan13 [68] prioritizes tests that failed recently or have a shorter duration. Tests are ordered based on their historical failure data, test execution time and domain-specific heuristics.

PT_Elbaum14 [19] favors tests that failed either recently or a long time ago.

PT_Thomas14 [102] uses topic modeling to diversity the tests that get executed earlier. Every prioritized test is selected if it contains the most different topics from the previous test in its identifiers and comments. The rationale behind this is that similar tests often find similar problems.

Build-Prioritization Techniques

To the extent of our knowledge, only one technique has been proposed to prioritize software builds, PB_Liang18 [63]. **PB_Liang18** [63] executes builds containing a recently-failing and recently-non-executed test in a collision queue. We apply PB_Liang18 to prioritize builds within a build waiting queue, as its previous evaluation did [63]. Queues form when build executions overlap in time.

4.4.2 Studied Metrics

Positions shifted for observed failing tests within a build measures the shifted positions for all observed failing tests (prioritized or not). A similar metric to this one was used in the evaluations of PT_Marijan13 [68], PT_Elbaum14 [19], and PT_Thomas14 [102]. For test-selection techniques, we measure the average number of shifted positions for all remaining tests — when a test is skipped, the next one can now run one position earlier.

Positions shifted for treated failing builds measures the number of builds between every treated (delayed/advanced) failing build's original observation position and its new position. This metric was studied by SB_Jin20 [51]. For test-granularity techniques, this metric is not impacted, since the build is still executed in the same position. For build-selection techniques, we consider that when a build is skipped, it will run as the next build (its tests will run on it).

Positions shifted for all failing builds measures the same as the previous one, but now across all failing builds. PB_Liang18 used a similar metric in its evaluation [63]. Through this metric, we can understand the impact of the previous metric over all builds.

Build-queue-length saved. This is a metric designed by us to measure how applying a technique could relieve the collision problem: when multiple builds are waiting to be executed within a limited resource. We follow the same configuration in PB_Liang18's paper. The build-queue-length refers to the median number of builds waiting ahead for each build in each project. With a pre-experiment on all projects, we find that for only one project - "Rails/Rails", the median value of every build's waiting queue is bigger than 0. Thus, we only report the result for this metric on that project.

4.4.3 Analysis of Results

Comparing Metrics. When comparing positions shifted for treated failing builds vs. all failing builds, for all techniques, the advance (PB_Liang18) or delay (others) that they introduce in the observation of failing builds is much lower when measured across the whole population of failing builds. The upside of this is that the undesired effect of most techniques (*i.e.,* delay of failure observation) is very low across all failing builds (median 0–2 builds). The downside is that the desired effect of PB_Liang18 (*i.e.,* advance of failure observation) is also very low across all failing builds (median 0 builds).

Next, we compare the performance of test selection techniques (*i.e.,* the only overlapping technique family) in the positions that observed failing tests shifted within a build vs. the positions that failing builds shifted across all builds. We observe that test selection techniques provided some advancement in the observation of test failures (lower than most test prioritization techniques), while introducing a very low delay in observation of build failures (median 0–2).

Comparing Granularities. We did not observe a substantial difference when comparing granularities — we observed stronger differences when comparing techniques.

Comparing Technique Strategies. When comparing technique strategies (prioritization vs. selection), test-selection techniques provided some advancement in the observation of failing tests within a build, but test-prioritization techniques provided better results overall (except PT_Elbaum14).

Comparing Techniques. PT_Marijan3 and PT_Thomas14 behave very similarly — despite their different approaches to prioritization — and they are both close to perfect, prioritizing most tests correctly. PT_Elbaum14 provides a lower advancement of test failures (also lower than many test-selection techniques), since it uses a simpler criterion — prioritizing tests that were executed very recently or a long time ago. All test-selection techniques provided a very similar advancement of test-failure observation, except ST_Herzig15 which was slightly better. Interestingly, ST_Herzig15 was one of the techniques with the lowest delay in build-failure observation (median 0 for all failing builds). At the build-granularity, PB_Liang18 had a very low impact in prioritizing builds because builds very rarely occurred concurrently in our dataset — only the Rails project had a meaningful number of concurrent builds. An important metric in PB_Liang18's original evaluation was the savings in the build-queue length. We plot the results for all techniques for this metric in Figure 4.4. Interestingly, we also observed that test-selection and build-selection techniques also had a strong impact in this metric — less so for test-selection techniques and SB_Abd19 because they skip fewer full builds (see §4.3.2). Regarding build-selection techniques, those that saved more builds (see §4.3.2) also saved more in the build-queue-length metric, but also introduced higher delays in build-failure observation.

4.5 Answers for Research Questions and Implications

We synthesize our observations and we lay out their implications to advance this area of research.

4.5.1 D1: Computational-cost Reduction

RQ1: What design decisions did not help?

First, we report on **missed opportunities** for saving more computational cost. Cost-saving techniques focused on skipping passing builds and tests, but they **did not specifically target those that would provide the highest savings**, *i.e.,* slower tests, slower builds, or all tests in a build (in the case of tests-selection). This is demonstrated by the fact that build-granularity techniques saved similar ratios of test number, test time, build number, and build time; and that test-granularity techniques saved similar ratios of test number and test time, and lower ratios of build time than test time.

We also learned that **training cost-saving techniques across projects** harmed their predictions. In other fields, training with data from multiple projects is considered to increase the accuracy of predictors. For cost-saving techniques, though, this exposed the techniques to more diverse sets of failures, making more builds/tests “look like a failure”, resulting on the predictors saving less cost (being less inclined to skip builds and tests).

Test-selection techniques were also limited in the cost that they could save when **they did not target saving full builds** — ST_Mach19_C and ST_Herzig15 saved very low build time despite saving a high ratio of tests. An additional aspect that contributed to ST_Herzig15 saving limited build time (despite saving high number of tests) is that **it only used features characterizing the tests**, but not the code changes in the build — *e.g.,* missing the opportunity to skip full builds for no-code changes.

RQ2: What design decisions helped?

Other design decisions allowed techniques to save high cost. A particularly useful design decision was **trying to predict seemingly-safe builds and tests** — SB_Abd19 saved 21% builds simply by skipping builds with no-code changes, and ST_Gligoric15 saved 36% builds skipping tests that did not cover the code changed in the build.

Another decision that provided high cost savings was to **skip full builds instead of individual tests** — thus also saving build-preparation time. Skipping all tests in a build allows to skip the time to prepare the build (*i.e.,* compilation and other overhead like virtual machine preparation), and we observed that **build-preparation takes a large portion of build time**. An illustrative example is how ST_Gligoric15 and ST_Herzig15 saved about the same ratio of test time, but ST_Gligoric15 saved much higher build time because it saved a much higher ratio of full builds.

Test-selection techniques, however, performed really well in terms of saving a high ratio of tests (84% by ST_Herzig15 and 80% by ST_Machalica_W). This is because they could save some cost spread out across many builds — *i.e.,* **skipping partial builds achieved high cost savings**. However, the test-selection **techniques that skipped full builds also achieved high savings**. Intentionally or not, ST_Gligoric15 saved many full builds by simply skipping all tests that did not cover the changed code. ST_Mach19_W also saved many full builds by approximating the same idea: one of its predictor's features is the distance between the changed code and the test.

Implications for Future Techniques

Our results have multiple implications for the design of future techniques. First, we encourage future techniques to consider **hybrid approaches** to save both full builds and also partial builds, *i.e.,* to save cost at both build and test granularity. Future techniques should also leverage the beneficial factors that we already observed, such as **skipping full builds with no-code changes or no tests to cover them**. To save more full builds, novel prediction features could be designed, **targeting slower builds** if possible — which no existing technique attempts. To save more tests, existing techniques already provide very useful features (saving a high ratio of tests), but other new features could be designed to target **saving more and slower tests**, and considering the **relationships between the tests and the code changes** in the build. Finally, our observations also show that **build time saved** is the metric that most comprehensively shows the cost saved by all existing techniques — even though cross-referencing multiple metrics allows for additional observations, as we did in this study.

4.5.2 D2: Missed Failure Observation

RQ1: What design decisions did not help?

In terms of the proportion of builds and tests that were skipped by cost-saving techniques, we generally observe that **the decisions that made techniques save higher cost also made them make more mistakes**, *i.e.,* skip higher ratios of failing builds and tests. It was also particularly interesting that **seemingly-safe techniques** — SB_Abd19 and ST_Gligoric15 — still **showed pretty high ratios of skipped failing builds and tests**.

Our study thus shows that skipping builds with no-code changes or without tests to execute them is not enough to guarantee that they will not fail. A quick look discovered that the builds and tests skipped by these techniques failed for different reasons, such as configuration or compilation errors (present in 35% of failing builds).

RQ2: What design decisions helped?

One design decision that reduced the skipped failing tests and builds was **training techniques across projects**. All the _C variants skipped lower ratios than their _W counterparts (except SB_Hassan17_C). Also **test-granularity techniques generally skipped lower ratios of failing tests** than build-granularity techniques did of builds.

Implications for Future Techniques

These results imply multiple recommendations for future techniques. First, future techniques should design **novel features to predict failures that are caused by no-code changes**, *e.g.,* configuration changes, to avoid assuming that seemingly-safe builds will not fail. Second, future techniques should attempt to **break this trade-off between saving cost and skipping failures**. Existing techniques generally increase cost savings by also increasing missed failure observations. Future techniques should attempt to improve one of the two dimensions by keeping the other one fixed (or optimal). Finally, future studies should propose **new metrics to better assess the trade-off between cost-saving and skipped-failures** of various techniques — since most techniques succeed in one at the expense of the other. SB_Jin20 [51] proposed the harmonic mean of the two as a balanced metric, but further study is granted to understand whether both should be valued equally or in a weighted manner — particularly considering the much higher ratio of passes to failures in CI datasets.

4.5.3 D3: Time-to-feedback Reduction

RQ1: What design decisions did not help?

Unsurprisingly, **build-selection techniques did not advance the observation of build failures at all**, but at least they introduced very low delays in the observation of failing builds (and also saved some computational cost). Similarly, **test-selection techniques also introduced a small delay in the observation of test failures**. **Build-prioritization also showed very limited advancement in observing failing builds**, but that was mainly because only one of our studied projects (open-source) had some contention in the build queue. We expect that industrial software project would obtain a much higher benefit from this approach. Finally, we also observed that the build-selection techniques that produced **higher cost savings also introduced higher delays in build-failure observation**, showing again the tension between both goals.

RQ2: What design decisions helped?

The best techniques to provide early feedback were test-prioritization techniques. In fact, PT_Thomas14 provided near perfect results. We also found that **test-selection techniques provided lower, but competitive advancement of test failure observation**, while also providing some cost savings. For example, ST_Herzig15 provided high advancement of test-failure observation within a build, with very low delay of build-failure observation, while also saving some computational cost. Similarly, we observed that **build-selection techniques could also provide reductions in build-queue-length that were competitive with build prioritization**.

Implications for Future Techniques

For future techniques, we recommend to **combine test prioritization with test selection**

techniques — since prioritization techniques could stop after the first failure is identified, and save the cost of running the remaining tests. We found that test-prioritization techniques already reached very high results (PT_Thomas14 is near perfect), so the features that they use could be also very useful for test selection to save cost. Conversely, existing test-selection techniques that already perform very well for cost-savings (*e.g.,* ST_Herzig15) could be improved in their ability to advance failure observation. Similarly, we recommend to further study the application of **build-selection techniques to provide early observation of build failures** by reducing the build queue via skipping builds in industrial projects in which parallel build requests are a larger issue. Finally, there is also space to develop new metrics that could capture the balance that techniques provide across all dimensions D1–D3.

4.5.4 Standing on the Shoulders of Giants

Our findings confirm and extend previous work:

D1

Beller *et al.* [6] observed that test time is a low proportion of build time. We extend this observation by finding that our studied test-selection techniques infrequently skipped full (all tests within) builds, which strongly limited their cost-saving ability. We thus recommend test-selection to incentivize skipping full builds to save higher cost in CI.

D2

Jin and Servant [51] observed a trade-off of higher cost savings incurring more missed build failures in their technique. We extend this observation by finding that all our studied techniques were affected by that trade-off (techniques ranked equally by cost savings as by missed failures). We additionally identified clear strategies that made techniques miss fewer failures: training across projects, and operating at test granularity. We also observed that a seemingly-safe technique [2] still missed a high ratio of failures. Finally, we elicited the need for better prediction of safe builds, and new metrics to compare trade-offs.

D3

Herzig *et al.* [41] found that their test-granularity technique incurs low delay in build-failure observations. We extend this observation by finding that all our other studied test-granularity techniques also incur low build-failure-observation delay, measured across all failing builds.

4.5.5 Enhancing Generalizability

The findings above are based on Travistorrent dataset which includes only open source projects. This can cause limitations on generalizability of our findings. Thus, we did an extra experiment to simulate how these techniques perform on other projects or data sets, like what if there is a project with tests that take a much higher proportion among the entire build process, *i.e.,* with a high test weight —- we believe that this is one important factor that can differentiate open source and industrial projects and can influence our findings. Since different test weights of projects are supposed to not influence techniques' performances on failure observations, we mainly focus on the cost-saving abilities of the studied techniques. Therefore, we performed a synthetic study to simulated the studied techniques on projects with different test weights.

In this study, we focused on understanding the techniques' performances on cost-saving and this intuitively excludes both build and test prioritization techniques because they cannot provide cost savings. Besides, we also excluded SB_Hassan17 since it almost skips every build according to our previous experiments. As a result, we included five techniques in this study, two of them are build selection techniques: SB_Jin20 and SB_Abd19, and three of them are test selection approaches: ST_Herzig15, ST_Gligoric15 and ST_Mach19.

We aim to understand how techniques perform in cost saving when the projects have different test weights. In other words, we are studying the relationship between the proportion of test time among build time (time weight) and the cost saving. We measure the cost saving by using the build time saved among all build time as approximation. We didn't include the metrics of failure observation because we assume that this dimension cannot be influenced by different test weights.

Since it is not practical to include project with all different test weights, we used the synthetic study to answer our research question. In the synthetic study, we simulate the studied techniques under projects with different hypothetical test weights. We also assume that the build time is equally distributed across all builds and each technique's behavior will not be impacted by the change of the test weight, *e.g.,* its precision and recall of prediction remains unchanged. The independent variable in this study is the test weight from 0.1 - 0.9 with ten-scale. We didn't include 0 and 1 because the build should include at least some tests to make it worth studying and the tests should not be 100% of the entire build. We also want to note that the test wight in the previous empirical studies is 0.18.

The results are plotted in Figure 4.5. From Figure 4.5, we can observe that the cost saving line for build selection approaches are horizontal because their cost saving abilities don't change with different test weights. SB_Jin20 is able to produce more cost saving than SB_Abd19. We can also observe that test selection approaches can produce higher cost savings when the test weight becomes higher. This is because test selection techniques can skip some tests in a build and when the test weight is higher, this cost saving is enlarged. Since this enlarged cost saving is correlated to how many tests can be skipped in a build, *i.e.,* how many builds are partially skipped, the test selection approach that produces more cost saving as the test weight increases are able to skip more partial builds. Among the test

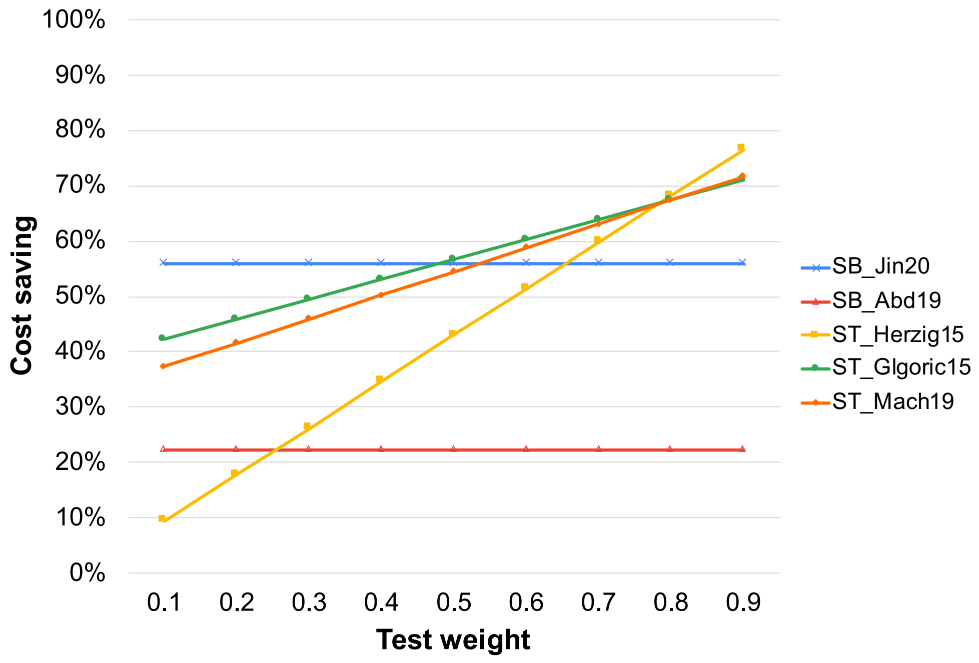


Figure 4.5: Cost saving achieved by studied techniques under different test weight.

selection approaches, we can find that the cost saving ability of ST_Herzig15 is increasing fastest because it can skip most partial builds. This finding confirms our observations in the previous studies. Besides, we can also observe that when the test weight is low, the test selection approaches can still produce some cost saving because they can skip some full builds that include both test phases and other phases. Among them, ST_Gligoric15 skips most full builds and thus it can save most cost when the test weight is relatively low.

When we compare build and test selection approaches in Figure 4.5, we can observe that at the beginning build selection approaches such as SB_Jin20 can produce more cost saving than test selection approaches but test selection approaches' cost saving ability exceeds build selection approaches when the test weight becomes higher, *i.e.,* the lines have crossings. We can observe that when the test weight is higher than 0.25, ST_Herzig15 save more cost than SB_Abd19, when the test weight is higher than 0.5, ST_Herzig15 and ST_Mach19 produces more cost saving than SB_Jin20, and when the test weight is higher than 0.65, the studied test selection approaches can save more cost than all stuided build selection approaches.

Based on these findings, we can conclude that test selection approaches have a higher potential in cost saving when the tests occupy a higher proportion of the entire build. This indicates that in those projects where test density is higher, current test selection approaches may be better at cost-saving since they can skip tests from both passing and failing builds fully and partially. Besides, it also acknowledges the possibility that combines both build and test selection approaches to allow skip full and partial builds to maximize the cost-saving. Furthermore, to better design a technique that can produce sufficient cost saving, we recommend that the technique should focus more on skipping full builds, *i.e.,* skipping all tests in passing builds when the test weight is relatively low and should focus more on skipping partial builds when the test weight becomes higher. Finally, we recommend that future techniques should maximize the recall of passing executions because it benefits the cost saving with no extra side effect. For test selection approaches, the distribution of skipped passing executions across passing and failing builds also matters for cost saving ability.

4.6 Conclusions

In this work, we performed the most exhaustive evaluation of CI-improving techniques to date. We evaluated 14 variants of 10 CI-improving approaches from 4 families on 100 real-world projects. We compared their results across 10 metrics in 3 dimensions. We derived many observations from this evaluation, which we then synthesized to understand the design decisions that helped each dimension of metrics, as well as those that had a negative impact on it. We compared techniques' cost-saving abilities under different test weights. Finally, we provide a set of recommendations for future techniques in this research area to take advantage of the factors that we observe were beneficial, and we lay out also future directions to improve on those factors that were not. We lay out plans to combine approaches at test and build granularities to save further costs, and to combine selection and prioritization approaches to improve on the early observation of failures while also saving some cost. Such techniques could consider additional history-based prediction features, such as the project's code-change history, *e.g.,* [90, 91, 92, 95, 96], since test-execution history was beneficial for some techniques, *e.g.,* [41]. We also discuss the need of future metrics to capture the various characteristics of these techniques in a more holistic way.

Chapter 5

PreciseBuildSkip: reduce delay of failure observation

According to the findings of Chapter 4, one major flaw that existing build selection approaches is the delay of failure observation caused by the misprediction on failing builds. Thus, in this chapter, we aim to explore ways to minimize the delay of failure observation for build selection approaches.

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 [19, 41], that have a long distance with the code changes [69], that test unchanged modules [97], or that are predicted to pass by a machine learning classifier [67]. 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.,* [32, 83, 84, 114, 115, 118, 122].

Another, more recent, approach is to predict and skip passing builds, *e.g.,* [51]. 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 [1, 2]. 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 [2].

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 work.

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 [2]. 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 [2], Abd20 [1], and Jin20 [51].

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 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.

5.1 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 [2]. 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?

5.1.1 Data Set

We performed our study over the Travis Torrent dataset [6], 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 [45, 73]. 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 [27, 46, 81], 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 [2], Abd20 [1], and Jin20 [51]), 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 [2] and Abd20 [1] 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 [88]) 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 static dependency graph using rubrowser [21]. We used a project's static dependency graph to check if there is a path between changed files and test files.

5.2 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 [2] 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

Table 5.1: 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.

builds.

We evaluated CI-Skip rules in two dimensions: cost-saving ability and safety, over a large dataset of continuous integration builds (see §5.1.1). 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.

5.2.1 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 [2], 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 5.1, 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 [2]. 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"* [2]. 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 [2]. 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[1]. 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[1] (*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 [2].

VersionRelease (VR): Developers sometimes skip a release preparation commit. Following Abdalkareem *et al.* 's study [2], 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 strong criterion for anticipating safe builds. Additionally, developers report

[1]A complete list: http://das.encs.concordia.ca/publications/which-commits-can-be-ci-skipped/

skipping builds *"When tests are not written to work for that particular source branch/repo"* [2]. 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.

5.2.2 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 5.1. 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.

Result

Figure 5.1 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

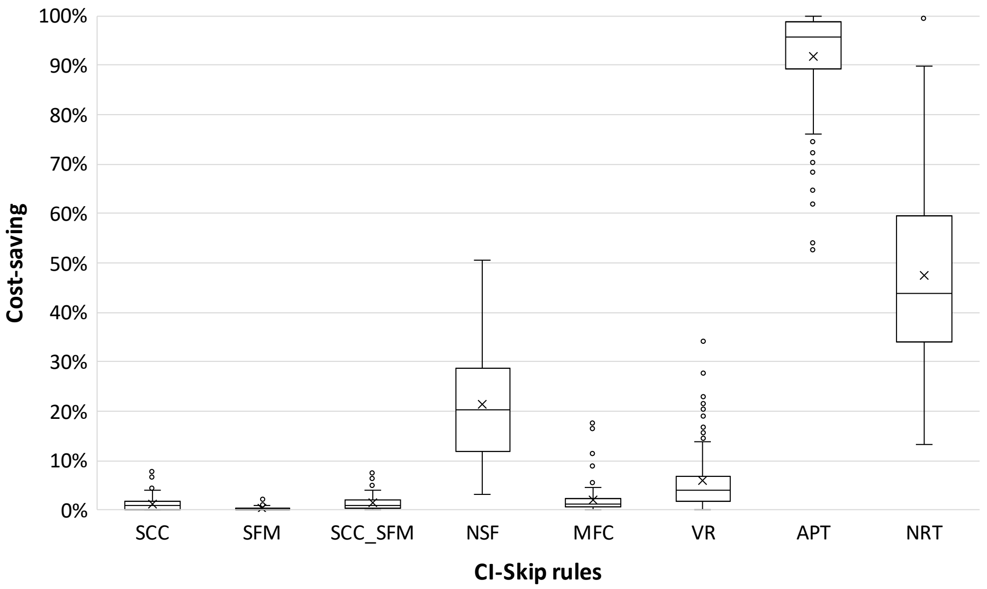


Figure 5.1: Proportion of builds that CI-Skip rules could save.

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 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.

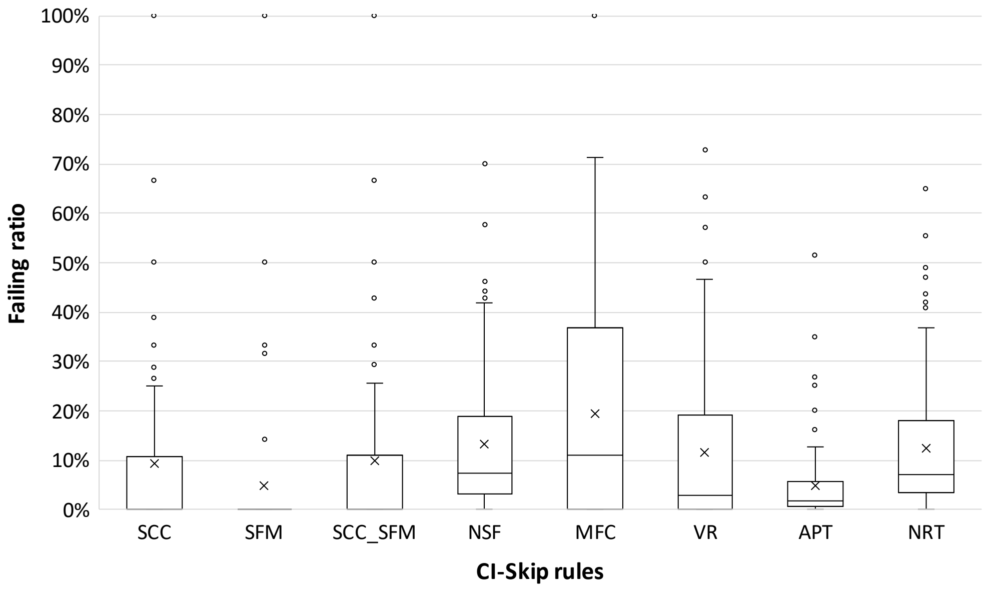


Figure 5.2: Proportion of failing builds among builds under each CI-Skip rule.

5.2.3 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 ratios in Figure 5.2. 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%.

Result

Figure 5.2 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 (§5.2.2). Thus, applying MetaFileChangeOnly manually would not be an effective way to safely skip builds.

We also found that SourceCommentChange, SourceFormatModification and SourceFormat-CommentChange were highly safe (with median 0% failing builds). Unfortunately, they also provide very few opportunities to save cost, as seen in §5.2.2. 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 (§5.4).

5.3 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.

Table 5.2: 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.

5.3.1 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 5.2.

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 [39]. This rule is triggered when a build changes a build-script file (*e.g.,* “pom.xml” or “build.gradle”).

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 [51]. 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.

5.3.2 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 5.3 shows the distribution of any combination of CI-Run rules present in failing builds under CI-Skip rules for any studied project.

Result

In Figure 5.3, 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

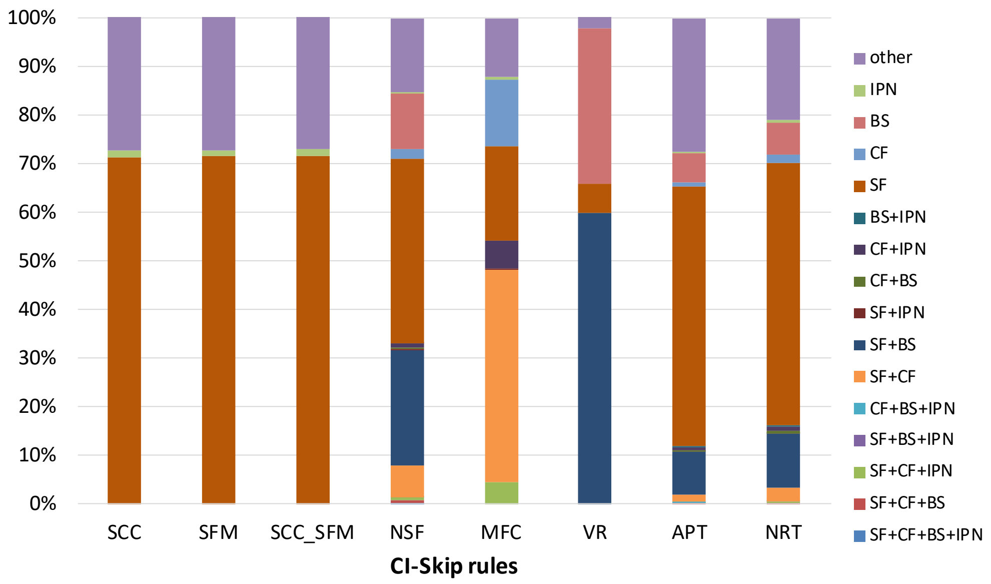


Figure 5.3: Distribution of failing builds captured by CI-Run rules under each CI-Skip rule.

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 5.3. 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).

5.3.3 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.

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.

Result

We report in Table 5.3 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 cor-**

relation 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.

Table 5.3: Correlation between CI-Run rules and failing builds under CI-Skip rules.

		CI_Skip Rules							
		SCC	SFM	SCC_ SFM	NSF	MFC	VR	APT	NRT
Exceptions	BS				0.06				0.02
	CF				0.07	0.11		0.03	0.04
	SF	0.83	0.83	0.82	0.83	0.82	0.8	0.79	0.82
	IPN				0.05	0.07		0.04	0.05

5.4 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 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 [111] in software projects for which only a few builds have been executed and thus they need additional data for training [87]. **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.

5.5 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.

5.5.1 Research Method

We describe the details of our evaluation below.

Studied Techniques

We evaluated PreciseBuildSkip in two versions. First, our proposal, PreciseBuildSkip, using a random-forest classifier (§5.4). 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 §5.4). 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 [51]: 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 [2]: The first rule-based build selection approach based on CI-Skip rules, which uses a subset of our studied CI-Skip rules (SourceCommentChange, SourceFormat-CommentChange, 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 [1]**:** 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.

Training and Testing

We used the same data set as Empirical Study 1 and 2, which includes 82,427 builds from 100 projects (see §5.1.1) We use 10-fold cross validation (each fold has 10 projects) to evaluate machine-learning-based techniques: PRECISEBUILDSKIP and Jin20. Thus, we randomly divide the 100 projects in our dataset into 10 sets (*i.e.,* folds), each one with 10 projects. We then iteratively select each fold as our test set. For each test set, we perform predictions for each build in it, having trained the classifier on the other 9 folds (the other 90 projects), *i.e.,* the testing data will not be known in advance by the predictor.

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 [1], and tested it in ours (see §5.1.1). Rule-based techniques (PBS_RB and Abd19) do not require training. So, we applied them directly to our dataset.

As in past work [51], 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 [51].

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 [1, 2, 51], and the second metric was covered in an existing work [51]. The last metric is designed in this work 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.

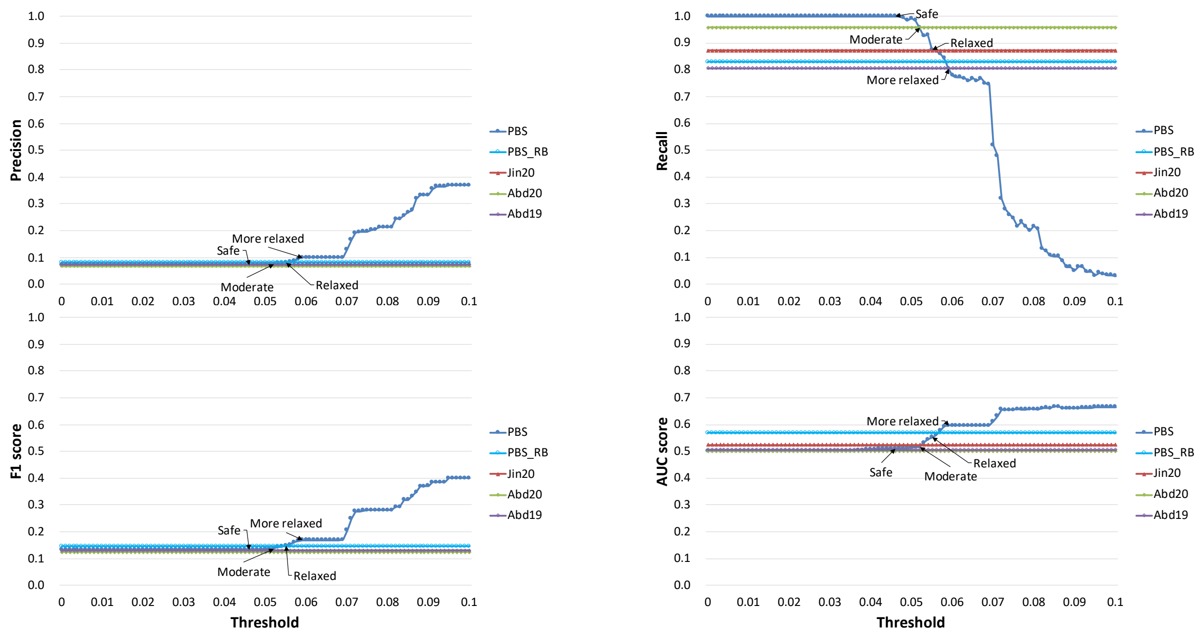


Figure 5.4: Performance comparison on predicting build failures.

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 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.

5.5.2 Results for RQ5: How correct are PreciseBuildSkip's predictions?

Figure 5.4 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 5.4-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 5.4.

In Figure 5.4-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 5.4-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 5.4-F1 score), most techniques achieve low values, as a result of their low precision. We observe a similar effect for AUC scores in Figure 5.4-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.

5.5.3 Results for RQ6: How much cost-saving and safety do PreciseBuildSkip's predictions provide?

We plot the results for this research question in Figure 5.5. 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 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.

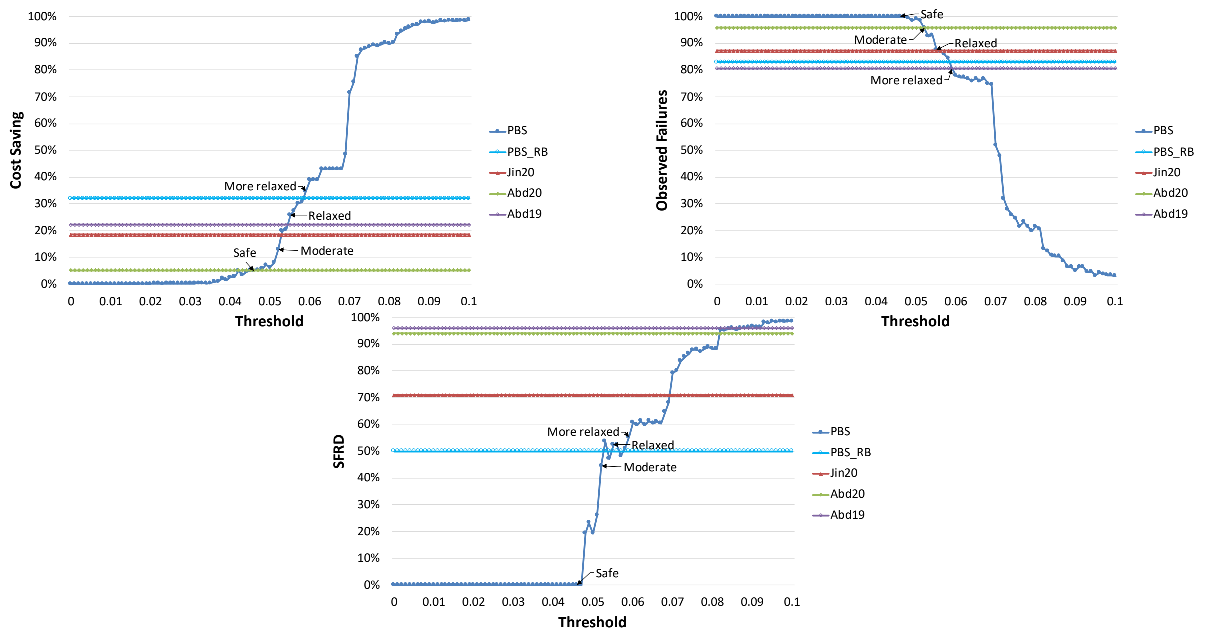


Figure 5.5: Cost saved and value kept by evaluated techniques.

Next, we compare **PreciseBuildSkip** with existing techniques by highlighting the same prediction thresholds discussed for RQ5 (see §5.5.2). **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.

5.5.4 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 5.6. 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 5.6 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.

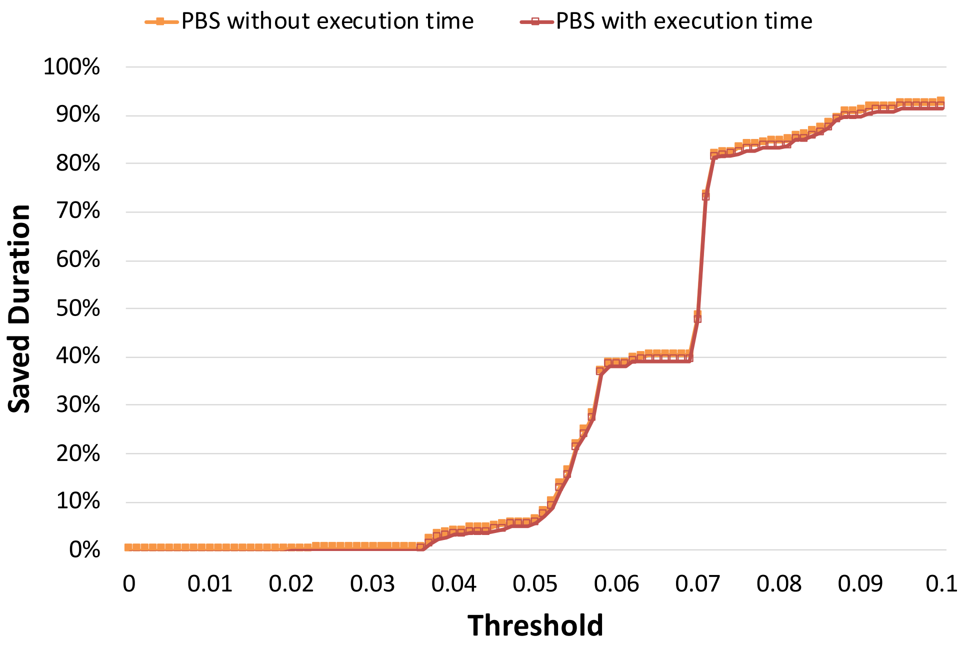


Figure 5.6: Build time saved by PRECISEBUILDSKIP including and excluding its execution time.

5.6 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 §5.5.3). We aim to understand how our approach performs with and without CI-Run rules in term of saved cost and kept value.

5.6.1 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.

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 §5.5.3.

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 §5.5.3.

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 §5.5.3.

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.

5.6.2 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 5.7 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%). PBS_IC_Moderate performs worse in both metrics (11.6% and 92.4%), compared with PBS_Moderate (12.9% *Cost Saving* and 96% *Observed Failures*). We compare PBS_Relaxed and PBS_IC_Relaxed as well 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

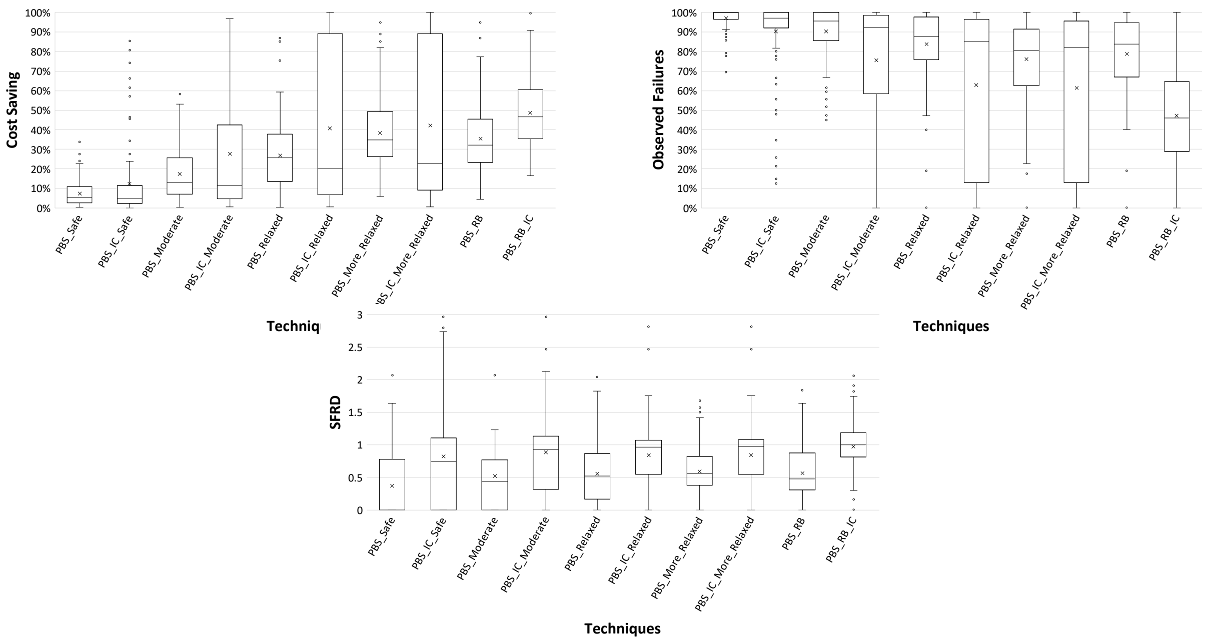


Figure 5.7: Cost saved and value kept by evaluated PreciseBuildSkip variants including and excluding CI-Run rules.

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 PBS_RB_IC, its corresponding technique. 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.

5.7 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.

5.7.1 Research Method

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

Studied Techniques

We study the following techniques: PBS_Safe, PBS_Moderate, PBS_Relaxed, PBS_More-relaxed, Abd20, and Jin20 in this experiment, as described in Experiment 2 (see §5.6.1). We omit two of the techniques that we studied 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.

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,

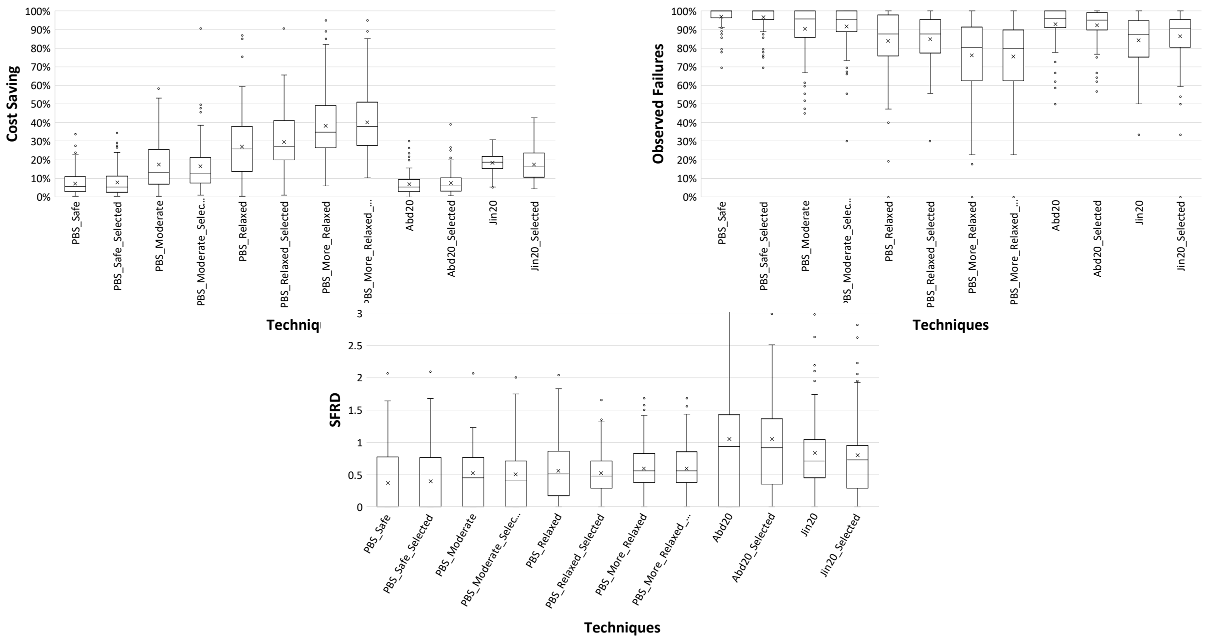


Figure 5.8: Cost saved and value kept by evaluated techniques when being trained under pre-selected data.

as described for Experiment 1 (see §5.5.1). 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 §5.5.1), but this time taking its training folds from its *selected-builds dataset* and the testing folds from the original dataset. However, we still keep the testing data unknown by the predictor.

5.7.2 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 5.8. 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 5.8 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 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%.

5.8 Implications

5.8.1 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 [110] 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.

5.8.2 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 [51][38] 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 §5.2, there are other ways to approximate it, *e.g.,* test selection techniques [67] can predict the result of tests. If all tests are predicted to pass, then AllPassingTest is valid.

5.9 Conclusions

In this work, 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.

Chapter 6

HybridBuildSkip: enhances the ability of cost saving.

The findings of Chapter 5 reflect that the delay of failure observation for build selection approaches can minimized, but the corresponding cost saving ability is also weakened. Therefore, in this chapter, we aim to explore ways to enhance the ability of cost saving while keeping minimizing the delay of failure observation for build selection approaches.

Existing cost-saving techniques for CI selectively skip executions at either the *build* [1, 2, 51] or *test* [32, 41, 67, 98] granularities. The former selectively skip *full builds* — skipping both the build preparation steps and all the test cases for the software. The latter selectively skip *partial builds* — executing the build preparation steps, but skipping some of the test cases.

Existing techniques use a wide variety of heuristics and machine-learning algorithms to predict whether a given build (or test case within a build) is going to pass or fail. According to a recent study [53], some techniques do well in cost-saving while others are better at failure-observation, *i.e.,* there is a trade-off that a technique that saves more cost always observes less failures. In this paper, we define and study two strategies to try to break this trade-off, *i.e.,* to improve both the cost-saving and failure-observation ratios. Our first strategy is to combine multiple existing techniques to take advantage of their own strengths. Our second strategy is to skip passing tests in builds with failing tests (partially skipping them).

We propose the first hybrid CI cost-saving approach, HybridCISave that allows skipping both full and partial builds and allows test-level approaches to decide whether the build-level executions can be skipped or not. HybridCISave is comprised of two phases: it firstly predicts if this build can be skipped fully through a hybrid build selector, HybridBuildSkip, and then for those builds that are decided to not skip, HybridCISave uses a test selector called HybridTestSkip to decide which tests can be skipped (skip partial builds). The rationale of this design is that we assume skipping builds partially can enlarge the cost-saving of build selection approaches while not sacrificing failure observation. HybridBuildSkip combines two sets of techniques: normal build selection techniques and test selection techniques to predict build result. The way how test selection can be used to predict build outcomes is that if all tests in one build are predicted to pass, then the build is considered to pass, otherwise the build is predicted as a failing build. The rationale of this design is that current build selection approaches are mainly focused on features of change size or historical performance, rather than directly predict the test failing possibility which is the main cause of the build failure [6]. With the design of HybridCISave, we evaluated it with all existing build selection approaches and found that HybridCISave outperformed these techniques by saving 14% duration, executing 100% of failing builds and observing 99.4% of test failures.

Our evaluation studied each of our novel ideas separately, and we define individual research questions for them. We performed an extensive study on the impacts of skipping partial builds and predicting build outcomes by test selection approaches. Skipping builds partially can provide large or small cost savings and can also influence failure observations. Thus, we did one experiment to compare HybridCISave and HybridBuildSkip to understand how much extra cost can be saved by the strategy of skipping partial builds and the corresponding impact on failure observation. We found that by skipping some tests from builds, HybridCISave can save 5% more durations and reduce its observed failing tests from 100% to 99.4%.

We then evaluated HybridTestSkip with other existing test selection techniques and found that HybridTestSkip can save 11% of tests while observing 100% of failing tests in median.

To understand the impact of applying test selection techniques to predict build outcomes, we did two experiments. We first compared HybridBuildSkip with and without combining test selection techniques to directly understand the impact of test selection techniques and found that the variant without test selection techniques misses more failures when saving same cost. We then explored further to compare the effectiveness of each feature and studied the information gain of all combined techniques and found that some test selection techniques rank the top in term of the feature importance. Finally, it's possible that the multiple predicting process of HybridCISave would threaten its cost-saving ability. Therefore, we compared the execution time of HybridCISave and its saved duration. We found that the execution time spent to execute HybridCISave was negligible compared to the execution time that it saved. Executing HybridCISave took a median of 0.0116 seconds, while the median build duration in our studied dataset was 441.5 seconds. Besides, we observed that, when accounting for the execution time of HybridCISave, its saved duration decreased only by up to 0.016%. This paper provides the following contributions:

- HybridCISave, the first hybrid selection approach in Continuous Integration that outperforms existing techniques in term of cost saving and failure observation.
- The strategy of skipping both full and partial builds.
- The strategy of applying test selection approaches to predict build outcomes.
- Two sets of studies, of impact of the two previous strategies.
- A study of the impact of the execution time of build selection approaches on its cost saving.

- HybridBuildSkip, the first build selection approach and HybridTestSkip, a hybrid test selection approach that is optimized for maximizing failure observation.

6.1 Our Approach: HybridCISave

We designed HybridCISave which consists of a hybrid build selection approach, HybridBuildSkip (introduced in §6.1.1) and a hybrid test selection approach, HybridTestSkip (introduced in §6.1.2) as shown in Figure 6.1. HybridCISave follows a two-phase strategy. First, HybridCISave uses HybridBuildSkip to predict the outcome of a build based on a set of predicting features (see §6.1.1). If HybridBuildSkip predicts the build to pass, it skips it fully, *i.e.,* all its steps and tests, and its cost is saved. Otherwise, *i.e.,* if HybridBuildSkip predicts the build to fail, the second phase will start to work. In its second phase, HybridCISave uses HybridTestSkip to predict individual test outcomes in this build and will only execute those tests that it predicts to fail — saving the cost of executing those that it predicts to pass. With this two-phase strategy, we expect HybridCISave to achieve higher cost reduction than normal build selection approaches, *e.g.,* HybridBuildSkip (since it now also skips builds partially) for similar ratios of observed failures (since we designed HybridTestSkip to be highly conservative in the way it decides to skip tests).

6.1.1 HybridBuildSkip

We designed HybridBuildSkip aiming to leverage and combine the predictions of the state of the art build selection techniques, but also aiming to adapt test selection techniques to aid in the problem of build selection. Thus, HybridBuildSkip informs its predictions with six different features in three categories: the prediction of three state of the art build selection

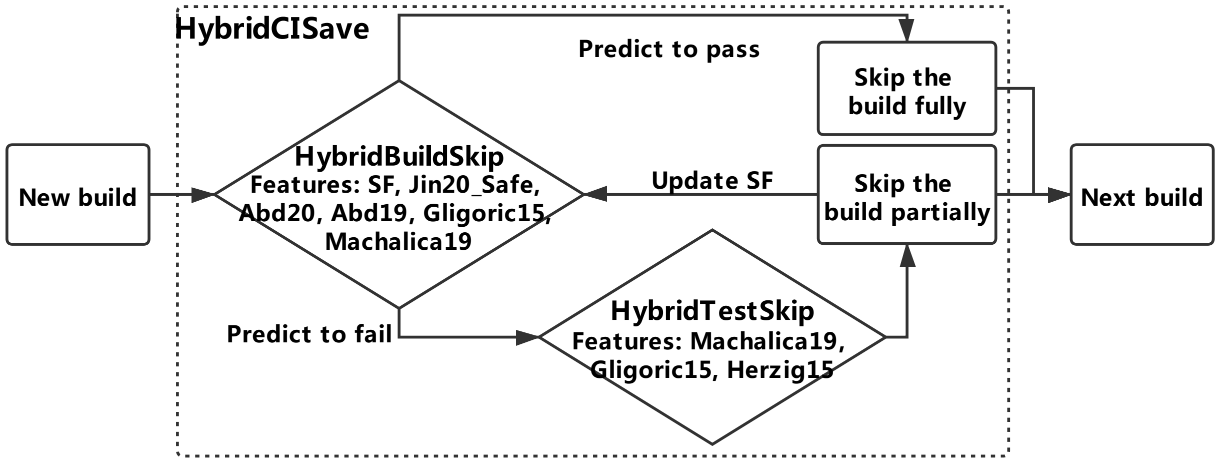


Figure 6.1: Flow chart of the design of HYBRIDCISAVE.

techniques, the adapted prediction of two test selection techniques, and one build feature that was found in previous work [51] to be highly predictive of build outcome: *subsequent failure*.

HYBRIDBUILDSKIP's first 3 features contain the prediction (pass / fail) of all the approaches that were previously proposed to selectively execute builds in CI for cost-reduction (to the extent of our knowledge): Abdalkareem19[2], Abdalkareem20 [1], and Jin20 [51]. Then, in its next 2 features, **HYBRIDBUILDSKIP** adapts the predictions of two state of the art test selection approaches: Gligoric15 [32] and Machalica19 [67]. For each of these approaches, HYBRIDBUILDSKIP adapts their prediction outcome (originally at test granularity) to the build granularity, *i.e.,* **combines selection techniques across granularities**, considering that they predict a build to pass if they predict all its tests to pass. If at least one test is predicted to fail, HYBRIDBUILDSKIP considers that the approach predicted the build to fail. Finally, HYBRIDBUILDSKIP's sixth feature contains a simple prediction: *fail* if the current build is subsequent to another build failure, and *pass* otherwise. Previous studies found this feature to be useful to predict build failures [38, 51, 73]. Thus, all six features are categorical, with a value of 0 if their corresponding technique predicted the build to pass, and 1 if it predicted it to fail.

HybridBuildSkip uses these features to predict whether a given build will pass or fail using a random-forest machine learning classifier. We train HybridBuildSkip as a cross-project predictor, *i.e.,* we train it in the past builds of different software projects than the one in which we apply it. This aids in the cold-start problem [111] in software projects where only a few builds have been executed. HybridBuildSkip then predicts the outcome of each build and only executes those that it predicts to fail. Finally, we make HybridBuildSkip customizable, *i.e.,* we can customize its prediction sensitivity to varying levels of tolerance to skipping failing builds.

Next, we summarize the techniques that inform HybridBuildSkip's features.

F1: Abdalkareem19 [2] is a build selection approach that uses heuristics to skip commits that only have safe changes, *e.g.,* changes on configuration or document files.

F2: Abdalkareem20 [1]: is a build selection approach that uses machine learning to predict builds that are likely to be skipped by developers with features including Abdalkareem19's rules.

F3: Jin20_Safe [51] is a build selection approach that uses a two-phase prediction algorithm. It uses machine learning to predict whether builds will pass based on statistical features, but when a build fails, it predicts all consecutive builds to fail, until one passes and the machine learning algorithm is used for prediction again. We pick the most conservative configuration for this technique to serve as the feature of HybridBuildSkip.

F4: Gligoric15 [32] is a test selection technique that skips tests that cannot reach the changed files, by tracking dynamic dependencies of tests on files. A test can be skipped in the new revision if none of its dependent files changed in the coming build. The rationale is that tests that cannot reach changed files cannot detect faults in them.

F5: Machalica19 [**67**] selects tests to run based on a machine learning algorithm with combined features of commit changes and test historical information. It uses the following features: file extensions, change history, failure rates, project name, number of tests and minimal distance. We ignored the step of determining the flaky tests when replicating.

F6: Subsequent Failures (SF) predicts a build to fail if the previous one failed. This feature is already used as feature in Jin20_Safe, but we emphasize it in HYBRIDBUILDSKIP by treating it as a single feature. We also want to note that this feature does not require future information, *i.e.,* it only updates its value when builds get actually executed.

6.1.2 HYBRIDTESTSKIP

We designed HYBRIDTESTSKIP to allow partially skipping those builds that HYBRIDBUILDSKIP decides not to skip while still maximizing the failure observation. HYBRIDTESTSKIP is a rule-based technique, so it does not require training data. HYBRIDTESTSKIP works as a heuristic test selection that combines multiple test selection techniques that were proposed for the continuous integration context (Machalica19 [67], and Herzig15 [41]) with an additional state of the art test selection technique: Gligoric15 [32]. HYBRIDTESTSKIP is designed to be conservative by predicting a test to pass (and skips it) if all its combined test selection techniques predict it to pass. Otherwise, it predicts it to fail (and executes it). With this design, we aim to minimize its ratio of missed test failures (*i.e.,* test failures that it would predict to pass).

We already described Gligoric15 [32] and Machalica19 [67] in §6.1.1. **Herzig15** [**41**] is based on a cost model, which dynamically skips tests when the expected cost of running the test exceeds the expected cost of skipping it, considering both the machine cost and human inspection cost [42]. We re-used the statistics from the paper such as the false alarm rate and the cost of running machines. This technique tends to skip tests that have long run-time.

6.1.3 Novelty

There are some novelties in the design of HybridCISave. First of all, HybridCISave allows skipping both full and partial builds to enlarge cost-saving. In this way, HybridCISave is able to take advantage of the cost from those passing tests in the executed builds. However, it can also bring some side-effects: the newly skipped tests may be failing tests that can delay the failure observation. Thus, we address this problem by designing HybridTestSkip in a specific way which is the second main novelty. We design HybridTestSkip to be extremely conservative: one test can only be skipped when all existing test selection approaches agree. This reduces its cost-saving ability, but ensures that it can observes the majority of failing tests. Then using it to skip tests in executed builds can enlarge cost saving without sacrificing failure observation. Finally, to better predict build outcomes, HybridBuildSkip combines both existing build selection and test selection approaches as its features. We design HybridBuildSkip to include test selection approaches because test failures are the major failing causes of build failures [6] and test selection approaches can strengthen the new technique's ability to detect test failures.

6.2 Research Questions

We perform three experiments to evaluate HybridCISave, analyze the effectiveness of its components and the impact of the execution time on its saved time. In our experiments, we answer the following research questions:

Experiment 1: Evaluating HybridCISave

RQ1: How effective is HybridCISave saving cost and observing failures, compared to existing build selection approaches?

Experiment 2: Analyzing HybridCISave's components

RQ2: What is the benefit of having a test selection component in addition to a build selection component?

RQ3: What is the benefit of having test selection approaches to predict build outcomes?

RQ4: What is the relative importance of each feature in HybridBuildSkip?

RQ5: How much cost-saving and failure-observation can HybridTestSkip achieve?

RQ6: What is the relative importance of each feature in HybridTestSkip?

Experiment 3: Counting End-to-End time

RQ7: What is the total execution time of HybridCISave and its individual components?

RQ8: How much cost does HybridCISave save in practice if we account for its execution time?

6.2.1 Data Set

We performed our study over the Travis Torrent dataset [6], which includes 1,359 projects (402 Java projects and 898 Ruby projects) with data for 2,640,825 build instances. 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 [45, 73]. To be able to explore 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

[27, 46, 81], 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 (13,464 failing builds and 40% of them happen after another failing build) from 100 projects. The median value of build number per project is 437.

To be able to implement our approach and replicate existing work, we extended the information in TravisTorrent of these 100 projects in multiple ways. First, we built scripts to download the raw build logs from Travis and parse them to extract all the information about test executions, such as test name, duration and outcome. Replicating existing approaches 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. Finally, we built a dependency graph for the source code of each project using a static analysis tool (Scitool Understand [88]) to compute the paths between files for implementing existing techniques. 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 static dependency graph using rubrowser (https://github.com/emad-elsaid/rubrowser). We used a project's static dependency graph to check if there is a path between changed files and test files.

6.3 Experiment 1: Evaluating HybridCISave

In the first experiment, we measure the cost reduction and failure observation that HybridCISave would provide in practice and evaluate it with all existing build selection techniques.

6.3.1 RQ1: How effective is HYBRIDCISAVE saving cost and observing failures, compared to existing build selection approaches?

Research Method

We applied **HYBRIDCISAVE** to predict the build and test outcomes and execute full and partial builds in a large dataset, and we compared its performance with that of all previous build selection techniques.

Metrics. We measured 3 metrics in this evaluation which including: *Saved_Duration*, *Observed_Build_Failures*, and *Observed_Test_Failures*, as in previous work [1, 2, 51, 53]. We measure *Saved_Duration* as the proportion of skipped build duration among all build time. It measures how much a technique reduced computational cost. A technique performs better in this metric if it saves a higher ratio of build duration.

$$Saved\_Duration = \frac{skipped\ duration}{all\ build\ duration}$$

Observed_Build_Failures is measured as the proportion of failing builds that are correctly predicted (*i.e.,* not skipped), among all failing builds. It measures the ability of a technique to not make mistakes (*i.e.,* not skip failing builds). A technique performs better in this metric if it correctly predicts a higher ratio of failing builds.

$$Observed\_Build\_Failures = 1 - \frac{\#\ skipped\ failing\ builds}{\#\ all\ failing\ builds}$$

Observed_Test_Failures is measured as the proportion of failing tests that are observed among all failing tests. It measures the ability of not making mistakes in test granularity. A technique performs better in this metric if it detects a higher ratio of failing tests.

$$Observed\_Test\_Failures = 1 - \frac{\#\ skipped\ failing\ tests}{\#\ all\ failing\ tests}$$

Studied Techniques. We replicated all existing build-selection techniques for comparison with HybridCISave.

HybridCISave: Our proposed approach (see §6.1). Since **HybridBuildSkip** is customizable, we evaluate it for multiple prediction-sensitivity thresholds: 0–100. Higher prediction sensitivities make **HybridBuildSkip** 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 failure observation.

Jin20 [51]: A 2-phase build selection approach, using a random-forest classifier with size features.

Abdalkareem19 [2]: The first rule-based build selection approach based on CI-Skip rules (rules that characterize builds that are likely to be skipped by developers).

Abdalkareem20 [1]: A machine-learning approach (also random-forest classifier) using Abdalkareem19's CI-Skip rules as features. We picked its random-forest variant since it is reported as the best performance classifier for Abdalkareem20 [1].

To compare **HybridCISave** to all variants of Jin20 (Jin20 is also customizable with thresholds), we fit their curves in the *Observed_Build_Failures* metric, to be able to observe their differences in terms of *Saved_Duration* for their variants that provide the same ratio of *Observed_Build_Failures*. We perform this fit by plotting Jin20's variant that has the closest (but smaller) *Observed_Build_Failures* for each studied threshold value of **HybridCISave**.

Training and Testing. We used the data set described in §6.2.1, which includes 82,427 builds from 100 projects. We use 10-fold cross validation to evaluate machine-learning-based techniques: **HybridCISave** and Jin20. Each fold has 10 distinct projects which are randomly assigned. Each build in the testing fold is tested by a classifier trained on the other 90 projects. Abdalkareem20, however, can not be trained in our dataset. Abdalkareem20 trains its classifier with developer-skipped commits, and our dataset has too few of these commits. Thus, we trained Abdalkareem20 in the 10-project dataset in which it was originally evaluated [1], and tested it in ours (see §6.2.1). Rule-based techniques (*e.g.,* Abdalkareem19, Gligoric15) do not require training. So, we applied them directly to our dataset.

As in past work [51], 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.,* SF, 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, as past work did [51].

Results

Figure 6.2 shows the results of all our studied techniques in terms of *Saved_Duration*, *Observed_Build_Failures* and *Observed_Test_Failures.* Each data point in Figure 6.2 represents the median value of one technique's performance on one metric across all 100 projects. In Figure 6.2, we can observe that observed failures generally drop and *Saved_Duration* increases as the prediction threshold increases. This means that higher cost reduction results in fewer observed failures, *i.e.,* there is a trade-off between cost-saving and failure-observation as also observed in previous work [51, 53]. However, HYBRIDCISAVE improves this trade-off in two ways: (1) HYBRIDCISAVE is able to save some cost while keeping 100% of build failure observations: up to 14% of build duration. (2) HYBRIDCISAVE is also able to still observe a moderate amount of build failures while saving almost all passing builds: HYBRIDCISAVE can skip 93% build duration while still observing 40% build failures.

When comparing HybridCISave with the existing technique that observes most build failures — Abd20, we can observe that HybridCISave achieves higher cost-saving (16%) than Abd20 does (5.1%) while observing same amount of failing builds (96%). We then compare HybridCISave with Abd19. We can find that HybridCISave is able to save higher cost (34.3%) than Abd19 (22%) when they observe same amount of build failures (81%). If we compare HybridCISave with the variant of Jin20 that was proposed as conservative (safer) (Jin20_Safe) [51], we can observe that HybridCISave still performs better in *Saved_Duration* (28% vs. 19%) with the same *Observed_Build_Failures* (87%).

We can also observe that HybridCISave outperformed the variants of Jin20, in 3 ways: (1) HybridCISave was able to observe 100% failures and save some cost, while Jin20 observed at most 87% of failing builds, in its most conservative configuration. (2) HybridCISave saved consistently higher cost than Jin20 for similar levels of observed build failures, *i.e.,* the saved duration curve of HybridCISave is predominantly higher than Jin20's. (3) HybridCISave observed up to 40% of build failures when saving most cost (up to 93%). The difference in cost-saving ability between HBS and Jin20 grew even larger after threshold 0.8.

6.4 Experiment 2: Analyzing HybridCISave's components

In the second experiment, we aimed to understand how the strategies of applying test selection approaches to predict build outcomes and skipping full and partial builds influence the performance of HybridCISave. We also studied the effectiveness of each feature in HybridBuildSkip.

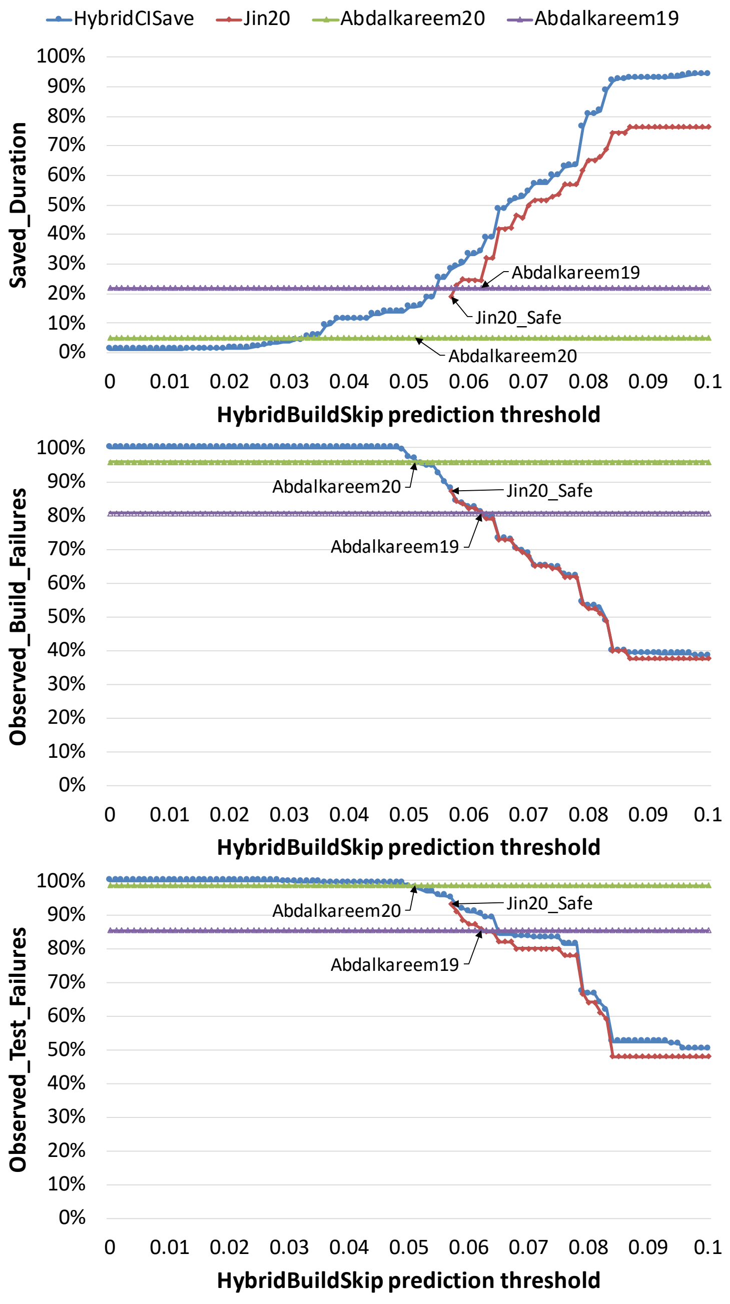


Figure 6.2: Cost saved and value kept by HybridBuildSkip and existing build selection techniques

6.4.1 RQ2: What is the benefit of having a test selection component in addition to a build selection component? & RQ3: What is the benefit of having test selection approaches to predict build outcomes?

Research method

To evaluate the effectiveness of skipping full and partial builds, we compared **HybridBuildSkip** (see §6.1.1) and **HybridCISave**, using the same dataset (§6.2.1) and process described in §6.3.1. This means that we also trained it across-projects with 10-fold cross validation. For this research question, we only compare **HybridCISave** with **HybridBuildSkip**, to ease the interpretation of its added benefit. To further understand the relative importance of combining selection approaches across granularities, we compare **HybridBuildSkip** with its base version - **HybridBuildSkip**-Base that doesn't include test selection techniques as features. To better compare these two techniques, we adjust **HybridBuildSkip**-Base to have closest but smaller failure observations so that we can compare their saved duration as we did for Jin20. The study follows the same set-ups (§6.2.1) and §6.3.1).

We made evaluations in two dimensions: cost-saving and failure-observation. We measured the cost-saving ability with *Saved_Duration*: the proportion of skipped duration among total duration, similarly to how we evaluated **HybridBuildSkip**. Measuring cost saving in terms of saved *duration* (*i.e.,* time) allows us to account in a single metric for saving both full and partial builds. Then, we measured the ability to observe failures using *Observed_Test_Failures*: the proportion of executed failing tests among all failing tests. We only used *Observed_Test_Failures* to be able to account in a single metric for observations of failures, whether they were part of a full build or partially-skipped build. We plot our

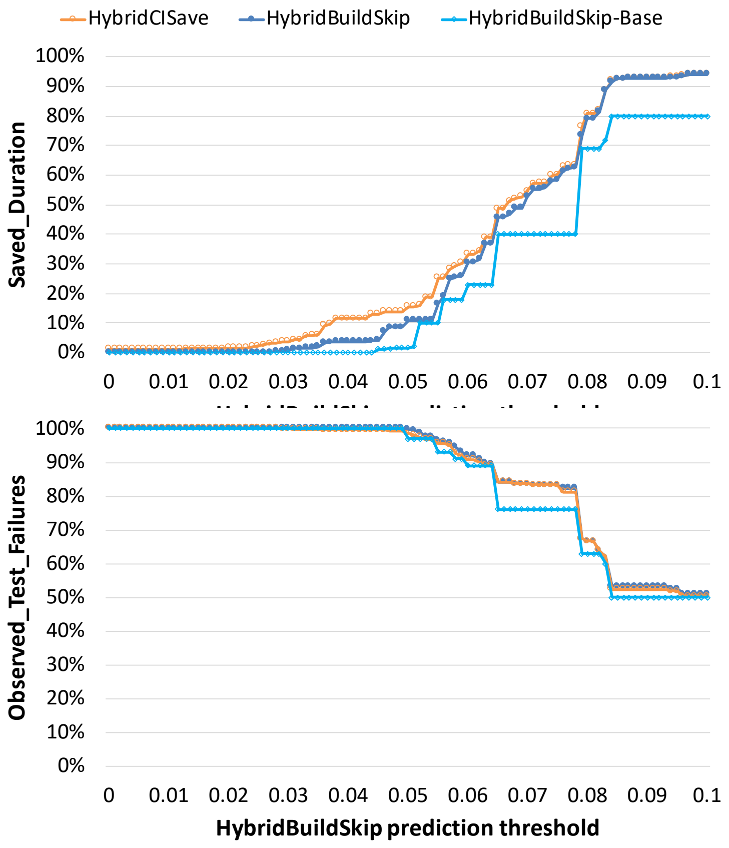


Figure 6.3: Cost saved and value kept by HYBRIDCISAVE and HYBRIDBUILDSKIP

evaluation results in Figure 6.3. This figure shows the median value for each metric across our studied projects, for multiple prediction thresholds.

Results

In Figure 6.3, we observe that HYBRIDCISAVE saved consistently higher cost than HYBRIDBUILDSKIP, with its highest benefit occurring at threshold 0.044, in which it saved 9% higher cost. This shows that the strategy of skipping both full and partial builds is able to increase cost-saving for our build selection approach. We also observed that when the threshold is bigger than 0.8, the benefit becomes negligible because HYBRIDBUILDSKIP is skipping the majority of builds in those range of thresholds and there is little space for HYBRIDTESTSKIP to save. We can also make an observation that when HybridBuildSkip skips no builds (*threshold* < 0.025), HybridCISave is still able to provide some cost-saving (1.4%) by skipping partial builds. This cost-saving is simply produced by its HybridTestSkip component.

Figure 6.3 also shows that HybridCISave and HybridBuildSkip observed similar ratios of test failures, *i.e.,* by adding HybridTestSkip in HybridCISave, we incurred only a minimal decrease in *Observed_Test_Failures*. The largest difference happened at threshold 0.057, in which HybridCISave observed 92.7% of failing tests while HybridBuildSkip observed 94.5%. The smallest difference occured at thresholds 0–0.027, where HybridBuildSkip and HybridCISave detected the same ratio of failing tests: 100%. Therefore, we can conclude that our strategy of skipping full and partial builds was able to save more cost, while observing almost the same ratio of test failures than HybridBuildSkip.

From the comparison of HybridBuildSkip and HybridBuildSkip-Base, we can observe that by adding test selection approaches to predict build outcomes, HybridBuildSkip is able to save more cost when observing the same amount of failing tests. This shows that test selection approaches can be effective for predicting build outcomes. Finally, we want to highlight that, while no previous technique observed 100% failures in our evaluation, both our proposed techniques achieved high ratios of failure observation, while also saving some cost: 9% saved build duration by HybridBuildSkip (with 100% observed failing builds), which HybridCISave improved to 14% saved build duration (with 99.8% observed failing tests).

Table 6.1: Importance of HybridBuildSkip's features.

Feature Name	Information Gain (median value)
F6: Subsequent Failures	0.1590
F4: Gligoric15 [32]	0.0060
F3: Jin20_Safe [51]	0.0040
F2: Abdalkareem20 [1]	0.0038
F1: Abdalkareem19 [2]	0.0034
F5: Machalica19 [67]	0.0028

6.4.2 RQ4: What is the relative importance of each feature in HybridBuildSkip?

Research Method

We applied the Information Gain Attribute Evaluation [4] on all features of HybridBuildSkip for all projects in our dataset. Table 6.1 shows the median value for each feature and its corresponding Information Gain across studied projects from high to low. We applied same methods to measure information gain of features of HybridBuildSkip-Base and we found that the ranking is same so we didn't include it here.

Results

In Table 6.1, we observe that SF was the feature with highest Information Gain value (0.159). This is because failing builds often continued to fail for a few more builds, until developers fixed the problem. This observation confirms the findings from previous work that failing builds are likely to appear as a sequence [51] and the status of the last build is the most effective feature to predict build outcomes [12, 38, 73]. It also shows the benefit of HybridBuildSkip to have included SF as an individual feature. However, we still want to note that even though SF is the most effective feature, only using it is not practical because this information may not be available when the previous builds are skipped. We also find that all other studied features had a small impact on the build outcome prediction. Among them, Gligoric15 had the highest Information Gain value while Machalica19 had the lowest impact. This shows that our design of **HybridBuildSkip** benefited from adapting the outcome of test selection techniques to the build selection problem, and adding them to the combination of its features.

6.4.3 RQ5: How much cost-saving and failure-observation can HybridTestSkip achieve?

Research Method

We used the same dataset and process described in §6.3.1, with two metrics to evaluate **HybridTestSkip** (in the same two dimensions as for evaluating **HybridCISave**, but adapted to the test granularity). We measured **HybridTestSkip**'s cost-saving ability using metric *Skipped_Tests*: the proportion of skipped tests among all tests. We measure **HybridTestSkip**'s ability to observe failures using *Observed_Test_Failures*: the proportion of executed failing tests among all failing tests. We show the distribution of these two metrics across our studied projects in Figure 6.4. Since **HybridTestSkip** is a rule-based technique, we don't apply different thresholds for it.

Results

In Figure 6.4, we can observe that **HybridTestSkip** is able to provide moderate cost savings: a median *Skipped_Tests* ratio of 11.3%. This shows that our strategy of combining test selection techniques in a hybrid way can also provide some cost-savings. We can also see that **HybridTestSkip** could observe a median value of 100% failing tests across our studied

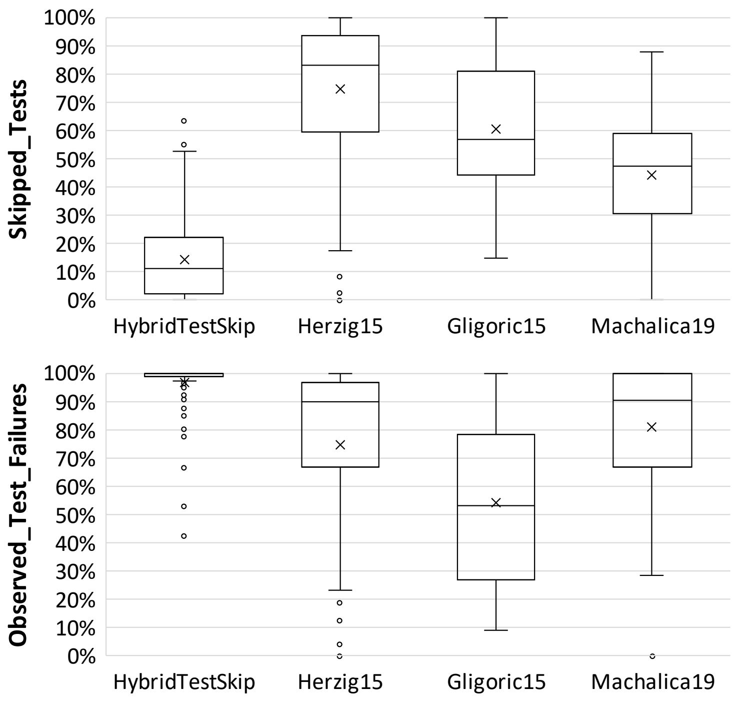


Figure 6.4: Cost saved and value kept by HybridTestSkip

projects. This shows that HybridTestSkip saved cost in a relatively safe way, *i.e.,* skipping some tests while capturing the majority of failing ones. Thus, our design in HybridTestSkip allows it to save some cost in test execution while still maximizing the ratio of test failures that get observed. This design makes it more conservative than its feature techniques —it saves less cost but observes more failures. Alternative designs for HybridTestSkip (*e.g.,* using machine-learning predictors over the same features, or relaxing the number of techniques that need to agree to skip a test) may achieve higher cost savings, but may also cause fewer test failures to be observed (*i.e.,* may skip more failing tests). We chose our current design of HybridTestSkip to prioritize failure observation over cost saving.

Table 6.2: Importance of HYBRIDTESTSKIP's features

Feature Name	Information Gain (median value)
Herzig15 [41]	0.02
Gligoric15 [32]	2.08
Machalica19 [51]	1.63

6.4.4 RQ6: What is the relative importance of each feature in HYBRIDTESTSKIP?

Research Method

We applied FOIL Information Gain Attribute Evaluation on each rule of HYBRIDTESTSKIP for all projects in our dataset. FOIL Information Gain [25] is used to evaluate rule-base classification and it computes the difference in information content of the current rule and its predecessor, weighted by the number of covered positive examples. Table 6.2 shows the median value for each feature and its corresponding FOIL Information Gain across studied projects.

Results

In Table 6.2, we observe that Gligoric15 was the feature with highest FOIL Information Gain: 2.08. In fact, Gligoric15 was also the feature with second highest information gain for HYBRIDBUILDSKIP (after SF). This shows that the strategy chose by Gligoric15 (skipping tests that cannot reach the changed files) is a strong predictor of whether tests will pass or fail. The next most important feature of HYBRIDTESTSKIP was Machalica19, with FOIL Information Gain of 1.63. This shows that Machalica's strategy of applying machine learning and considering historical test information is also a valuable predictor for test outcome. Finally, Herzig15 had a much lower information gain than than the other two features: 0.02.

This may be because it selects test to execute based on a slightly different criterion — using a value-cost formula to decide if the expected cost of running the test exceeds the expected cost of skipping it.

6.5 Experiment 3: Counting End-to-End time

Since executing multiple techniques and combining them could potentially incur high execution times, we study in this last experiment, whether HYBRIDCISAVE's execution time reduces its achieved cost savings. We measured HYBRIDCISAVE's execution time in our Experiments 1 and 2, on a machine with a 2.5 GHz CPU, 32 GB RAM, Ubuntu 16.04.3 LTS and Python 3.5.2.

6.5.1 RQ7: What is the total execution time of HYBRIDCISAVE and its individual components?

Research method

In this experiment, we measured the execution time of HYBRIDCISAVE and all the techniques it uses, including Gligoric15, Jin20_Safe, Abdalkareem20, Abdalkareem19, Machalica19 and Herzig15. We computed their run-time in seconds across studied projects. For those techniques with a machine learning classifier, we only included its prediction time, *i.e.,* we didn't include the training time since the training process can be performed and updated offline (separately from the CI cycle). We also measured the build duration in our studied projects to compare the execution time of these techniques with the time spent in builds.

Table 6.3: Time taken to execute HybridCISave per build.

	Median (s)	Max (s)
HybridCISave (total)	0.011600	0.183400
HybridBuildSkip (total)	0.011600	0.183200
HybridTestSkip (total)	0.007000	0.174500
HybridCISave (self)	0.000011	0.000044
HybridBuildSkip (self)	0.001500	0.002000
HybridTestSkip (self)	0.000044	0.000100
Gligoric15	0.003700	0.153900
Herzig15	0.000001	0.000001
Machalica19	0.002900	0.021700
Abd19	0.000001	0.000001
Abd20	0.001600	0.031600
Jin20_Safe	0.001500	0.002200
Build duration	441.500000	2,151.000000

Result

We report in Table 6.3 the results of this experiment as the median value of min, first quartile, median, mean, third quartile, max values of actual execution time of HybridCISave and its components across our studied projects. We observe that in general the total execution time of HybridCISave is negligible compared to the build duration — the median value of HybridCISave's execution time is 0.0116 seconds and the median value of build duration is 441.5 seconds. This shows that the execution time of HybridCISave has a negligible impact on its achieved cost-reduction. We can also observe that HybridBuildSkip takes much longer time than HybridTestSkip (median 0.0116 seconds vs. 0.007 seconds), since it requires more information and its prediction process is also more time-consuming. This also makes HybridBuildSkip take similar time to HybridCISave.

By comparing the actual execution time of each component in HybridCISave, we observe that test selection approaches generally take longer time. This is because normally there are many tests in one build, so the prediction of test outcomes has to be repeated many times for each build. Among test selection approaches, Gligoric15 takes highest time (median 0.0037 seconds per build). Another test selection approach (Machalica19) takes higher time (median 0.0029) than any other build selection technique. However, Herzig15 and HYBRIDTESTSKIP (self) take less time because they use heuristics, which are less time-consuming. Finally, we found that among build selection techniques, those approaches that require machine learning prediction (Jin20_Safe, Abdalkareem20 and HYBRIDBUILDSKIP (self)) require longer execution time. Amond them, Abdalkareem20 takes highest time (median 0.0016 seconds per build) because its predictor is triggered for every commit in one build which means it needs to predict more times than other build selection approaches. Jin20_Safe takes less time since it is designed to not have to make predictions for every build — once it observes a failing build, it continuously executes the build until it observes a passing build. These results show that the run times of our studied techniques is negligible compared to the build times.

6.5.2 RQ8: How much cost does HYBRIDCISAVE save in practice if we account for its execution time?

Research method

To understand how HYBRIDCISAVE's cost savings change when accounting for its execution time, we plot its results obtained in Experiment 2 for the *Saved_Duration* metric, before (HYBRIDCISAVE) and after (HYBRIDCISAVE_REAL) we deduct HYBRIDCISAVE's execution time from it.

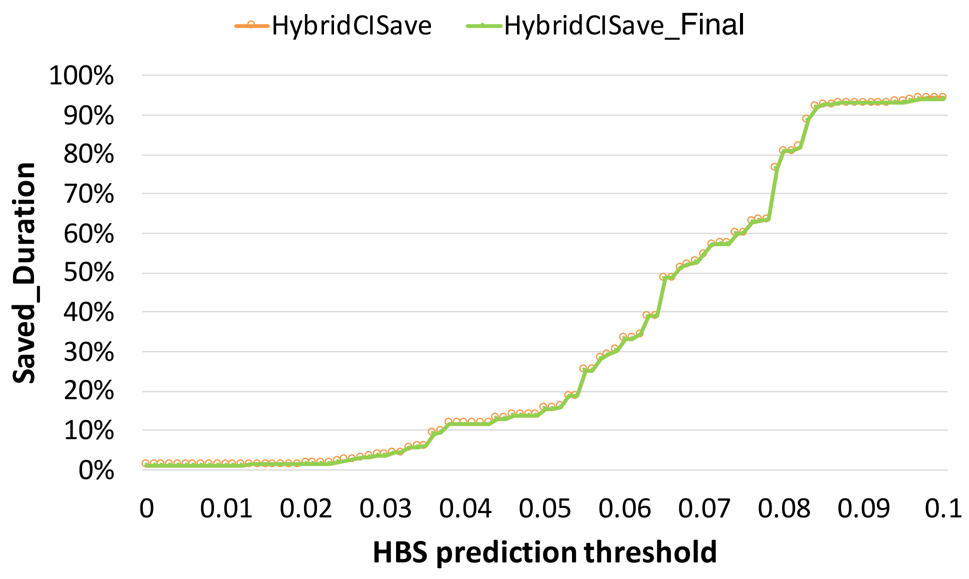


Figure 6.5: Cost saved by **HybridCISave** with or without considering its execution time.

Result

Figure 6.5 shows the median value of *Saved_Duration* across studied projects. We observe that in general the achieved *Saved_Duration* by **HybridCISave_Real** is very close to the cost reduction achieved by **HybridCISave**, which means the execution time of **HybridCISave** has a negligible impact on its achieved cost-reduction. All of the differences are smaller than 0.02%. The biggest difference is 0.016%: at the point of threshold 0.018, the execution time makes the saved duration of **HybridCISave** drop from 1.458% to 1.442%. The smallest difference is 0.001%: at the point of threshold 0.033, the execution time pulls the saved duration down from 5.732% to 5.731%. Finally, for **HybridCISave**'s highest cost-saving while also achieving high ratio of observed failures (threshold 0.047), the difference between **HybridCISave_Real** and **HybridCISave** is only 0.01%. As a result, we conclude that the execution time of **HybridCISave** has a negligible impact on its ability to save cost.

6.6 Discussion

We discuss some interesting points observed in this paper to advance this area of research.

Build failures that are hard to be detected by existing techniques. We further explored the failure reason of 11 failing builds that can only be detected by 1 feature of HYBRIDBUILDSKIP (all of failing builds can be detected by at least one feature). We found that 6 of 11 failing builds only include changes on project configuration files, *e.g.,* pom.xml. This shows that this kind of configuration files can also result in build failures and future techniques can take advantage of that. Besides, we observed that 4 of 11 failing builds only include Travis configuration file (travis.yaml). This indicates that the configuration file of Travis CI can cause failing builds and developers may struggle to write the configurations correctly [110].

Combining approaches in different ways. Prior work combined multiple approaches in many different ways for better performance. For example, Zhang [118] combined regression test selection strategies in both file and method levels. In this work, we take advantage of existing techniques by treating each of them as a feature of our predictor. Instead of manually picking the strength of each technique, we asked the machine learning algorithm to decide how to account for each technique's prediction in the given build. In our experiments, we found that the predictions of all existing build selection approaches have an impact on the eventual prediction of HYBRIDBUILDSKIP. We also found that Subsequent Failure is the most effective feature that also confirms the findings from previous studies [38, 51, 73]. In the future, we will explore other possible ways to combine build selection techniques.

Combining approaches using various machine learning algorithms. Other prior work studied what machine learning classifiers provide the best accuracy for build outcome predictions, *e.g.,* [1]. In this work, we also studied various machine learning algorithms for HybridBuildSkip including Random Forest, AdaBoost and Multilayer Perceptron. We evaluated HybridBuildSkip under these three machine learning algorithms and found that they all have similar performance in terms of cost-saving and failure-observation. We selected Random Forest classifier for our approach because it was the fastest when making its predictions. We will also explore additional algorithms in the future.

Predicting build outcomes through test selection approaches. Prior work [118] combined approaches at file and test levels to achieve better accuracy in test selection. In this work, we also combine approaches from different granularities, *i.e.,* build level and test level. We take advantage of test selection approaches to predict build outcome — if all tests in one build are predicted to be passing and thus can be skipped, we will predict the whole build as a passing build and not execute it. We applied two test selection approaches for HybridBuildSkip and both had some impact on the predictions, especially Gligoric15. In the future, we will also explore using additional test selection techniques for HybridBuildSkip's and HybridCISave's algorithms.

Build selection approach End-to-end execution time. Since HybridBuildSkip and HybridCISave rely on predictions from many other existing approaches, an important question is whether the total execution time of all these techniques could have a relatively big impact on the saved duration achieved by our approaches. However, our experiment 3 showed that the execution time of HybridBuildSkip and HybridCISave (including all their components) is negligible compared with their saved build duration. This motivates the future design for more complex and time-consuming build selection approaches. It also highlights the importance of measuring the execution time of the technique itself in a technique evaluation, to account for its impact on the cost savings achieved.

Aggressive cost-saving for build selection. Different developers will have different preferences in the trade-off between observing failing builds early and saving build effort.

For this reason, prior work [51] and the techniques proposed in this paper are designed as customizable, to cater to different preferences. For aggresive configurations of our approach, HybridCISave is able to save 93% of build duration and still observe 40% of failing builds. Some practitioners may prefer to achieve high cost savings, even if the achieved ratio of observed failures is limited. Future approaches could aim to save cost aggressively first, and then try to increase the ratio of observed build failures at the same time.

Different dimensions to evaluate build selection approaches. Previous (and this) work [1, 51, 53] evaluate build selection approaches in two dimensions: cost-saving and failure-observation. However, since there is a trade-off between these two dimensions, techniques may work well in one dimension, but not the other one. Therefore, there should be an easier way to compare build selection approaches. One way to solve this in future work is to design new metrics. Previous work [51] proposes a balanced metric as the harmonic mean of cost-saving and failure-observation, but there may be better ways to measure this balance. We took a different approach to simplify the comparison between customizable techniques in Experiment 1, in which we first chose the variants that achieved similar ratios of observed failures to then compare their cost-saving ability. This allowed us to use a single metric for comparison. However, not all techniques are customizable, which motivates future work to propose better metrics to compare the trade-off of cost savings and observed failures of different approaches.

Chapter 7

Future Work

As discussed in the chapters above, the research of this dissertation managed to save the cost produced in the continuous integration process with the minimal side effect of delaying failure observations. Through applying our approaches, developers could save the waiting time of the outcome of the continuous integration and the company can save the computational resources for more important and likely-to-fail executions. Nevertheless, there are still many problems remaining unaddressed in the process of continuous integration. For example, some developers may want to keep some passing builds to be executed such as the first passing builds happening right after a sequence of failing builds, while others may want to skip some failing builds because these builds don't contain meaningful changes to fix the defect. Besides, the usage of build selection approaches can raise new problems happening in the developers' programming, *e.g.,* developers may find it harder to locate a bug when some prior builds are skipped, which enlarge the code change sets. Future build selection approaches may also try other algorithms [40, 78] in predicting build outcomes. The followings are some directions that we plan to work on in the future:

Flexible build skip. The research work in this dissertation assumes that failing executions are able to provide actionable feedback and thus can be more valuable than passing executions which are supposed to be skipped for the aim of cost saving. This assumption is confirmed by many other existing work [32]. However, sometimes developers may want to skip some failing builds as well as execute some passing builds as we discussed above. Thus, in the future we plan to propose a flexible build selection approaches that can decide to skip builds based on some of developers' preferences. This technique should also aim to provide options of being triggered autonomously to skip builds or informing developers its prediction results to let developers make their own decisions.

Build selection with trust. One problem can be raised when build selection approaches are applied is that developers may find it hard to trust the prediction outcome of the build selection approach. This is because developers may find the approaches lacking the detailed information and they are fear that failing executions are skipped and can influence the future coding work. To address this problem, we plan to propose a build selection technique that can provide the information of prediction details. It should include information such as the prediction confidence, historical failure ratio and the suspect code churn.

Fault localization under CI context. Another problem that can occur during the application of build selection approaches is that developers may find it hard to locate the bug because the prior builds are skipped, making the change set much bigger. Existing work of fault localization and debugging techniques [3, 15, 92, 94, 120, 121] can detect bugs effectively but they are not proved to work under the continuous integration environment. Therefore, as a future work, we plan to develop a fault localization approach that can be used in continuous integration context to complement the build selection approaches. We also plan to compare the performance of this technique with other traditional fault localization techniques and debugging tools [99] that are not designed under the CI context.

Chapter 8

Summary and Conclusions

Continuous Integration is a popular software engineering process with many benefits such as detecting bugs earlier. The main cost lying in Continuous Integration is the abundant computation cost. To address this problem, we proposed to skip passing executions while keeping failure executions in Continuous Integration because only failing executions can provide actionable feedback which is more valuable. To achieve this goal, we designed multiple build selection approaches as the technique that can automatically decide which builds should be executed. First of all, we designed the first build selection technique — SmartBuildSkip that can save the cost of CI in a balancable way by skipping predicted-to-pass builds. We then compare our technique with existing techniques that could be applied to improve CI and evaluated the strength and the weakness of different strategies as well as giving recommendations. Next, we proposed a build selection — PreciseBuildSkip that minimizes the side-effect of build selection techniques (delayed failure observation) into a median value of 0% while saving some cost. Finally, we developed explore a hybrid selection technique — HybridBuildSkip based on existing build selection techniques. HybridBuildSkip is designed to skip both full and partial builds while minimizing the delayed observations of failing executions at the same time. Some other work [49, 50] are also completed during this journey, but are out of scope of this dissertation.

Bibliography

[1] Rabe Abdalkareem, Suhaib Mujahid, and Emad Shihab. 2020. A Machine Learning Approach to Improve the Detection of CI Skip Commits. *IEEE Transactions on Software Engineering* (2020).

[2] R. Abdalkareem, S. Mujahid, E. Shihab, and J. Rilling. 2019. Which Commits Can Be CI Skipped? *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2897300

[3] Kijin An and Eli Tilevich. 2019. Catch & release: An approach to debugging distributed full-stack JavaScript applications. In *International Conference on Web Engineering*. Springer, 459–473.

[4] B Azhagusundari, Antony Selvadoss Thanamani, et al. 2013. Feature selection based on information gain. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)* 2, 2 (2013), 18–21.

[5] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 433–444.

[6] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. 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, 356–367.

[7] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. 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, 447–450.

[8] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *Proceedings of the 14th working conference on mining software repositories*.

[9] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. 2008. Duplicate bug reports considered harmful… really?. In *2008 IEEE International Conference on Software Maintenance*. IEEE, 337–345.

[10] Marcelo Cataldo and James D Herbsleb. 2011. Factors leading to integration failures in global feature-oriented development: an empirical analysis. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 161–170.

[11] Ahmet Celik, Alex Knaust, Aleksandar Milicevic, and Milos Gligoric. 2016. Build system with lazy retrieval for Java projects. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 643–654.

[12] Bihuan Chen, Linlin Chen, Chen Zhang, and Xin Peng. 2020. BUILDFAST: History-Aware Build Outcome Prediction for Fast Feedback and Reduced Cost in Continuous Integration. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 42–53.

[13] Cloudbee. 2019. Jenkins Enterprise by CloudBees 14.5 User Guide - Skip Next Build Plugin. https://docs.huihoo.com/jenkins/enterprise/14/user-guide-14.5/skip.html. [Online; accessed 27-April-2019].

[14] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social coding in GitHub: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on computer supported cooperative work*. 1277–1286.

[15] Tung Dao, Lingming Zhang, and Na Meng. 2017. How does execution information help with information-retrieval based bug localization?. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 241–250.

[16] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk*. Pearson Education.

[17] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. 2016. DevOps. *Ieee Software* 33, 3 (2016), 94–100.

[18] Sebastian Elbaum, Alexey G Malishevsky, and Gregg Rothermel. 2002. Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering* 28, 2 (2002), 159–182.

[19] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 235–245.

[20] Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Empirically evaluating readily available information for regression test optimization in continuous integration. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 491–504.

[21] Emad Elsaid. 2019. Rubrowser (Ruby Browser). https://github.com/emad-elsaid/rubrowser. [Online; accessed 21-January-2022].

[22] Wagner Felidré, Leonardo Furtado, Daniel Alencar Da Costa, Bruno Cartaxo, and Gustavo Pinto. 2019. Continuous Integration Theater. In *Proceedings of the 13th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 10.

[23] Jacqui Finlay, Russel Pears, and Andy M Connor. 2014. Data stream mining for predicting software build outcomes using source code metrics. *Information and Software Technology* 56, 2 (2014), 183–198.

[24] Martin Fowler and Matthew Foemmel. 2006. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf* 122 (2006), 14.

[25] Johannes Fürnkranz and Peter A Flach. 2003. An analysis of rule evaluation metrics. In *Proceedings of the 20th international conference on machine learning (ICML-03)*. 202–209.

[26] Keheliya Gallaba, Yves Junqueira, John Ewart, and Shane Mcintosh. 2020. Accelerating Continuous Integration by Caching Environments and Inferring Dependencies. *IEEE Transactions on Software Engineering* (2020).

[27] Keheliya Gallaba, Christian Macho, Martin Pinzger, and Shane McIntosh. 2018. 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, 87–97.

[28] Alessio Gambi, Zabolotnyi Rostyslav, and Schahram Dustdar. 2015. Improving cloud-based continuous integration environments. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 797–798.

[29] Aakash Gautam, Saket Vishwasrao, and Francisco Servant. 2017. 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, 495–498.

[30] Taher Ahmed Ghaleb, Daniel Alencar da Costa, and Ying Zou. 2019. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering* (2019), 1–38.

[31] Taher Ahmed Ghaleb, Daniel Alencar da Costa, Ying Zou, and Ahmed E Hassan. 2019. Studying the impact of noises in build breakage data. *IEEE Transactions on Software Engineering* (2019).

[32] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 211–222.

[33] Milos Gligoric, Rupak Majumdar, Rohan Sharma, Lamyaa Eloussi, and Darko Marinov. 2014. Regression test selection for distributed software histories. In *International Conference on Computer Aided Verification*. Springer, 293–309.

[34] Mehdi Golzadeh, Alexandre Decan, and Tom Mens. 2021. *On the rise and fall of CI services in GitHub*. https://doi.org/10.5281/zenodo.5815352

[35] Mary Jean Harrold, James A Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S Alexander Spoon, and Ashish Gujarathi. 2001. Regression test selection for Java software. *ACM Sigplan Notices* 36, 11 (2001), 312–326.

[36] Ahmed E Hassan and Ken Zhang. 2006. Using decision trees to predict the certification

result of a build. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE, 189–198.

[37] Foyzul Hassan, Shaikh Mostafa, Edmund SL Lam, and Xiaoyin Wang. 2017. 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, 38–47.

[38] Foyzul Hassan and Xiaoyin Wang. 2017. Change-aware build prediction model for stall avoidance in continuous integration. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE Press, 157–162.

[39] Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: An automatic approach to history-driven repair of build scripts. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 1078–1089.

[40] Keith Henderson, Tina Eliassi-Rad, Christos Faloutsos, Leman Akoglu, Lei Li, Koji Maruhashi, B Aditya Prakash, and Hanghang Tong. 2010. Metric forensics: a multi-level approach for mining volatile graphs. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. 163–172.

[41] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The art of testing less without sacrificing quality. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 483–493.

[42] Kim Herzig and Nachiappan Nagappan. 2015. Empirically detecting false test alarms using association rules. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 39–48.

[43] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 197–207.

[44] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. 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, 426–437.

[45] Md Rakibul Islam and Minhaz F Zibran. 2017. Insights into continuous integration build failures. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*. IEEE, 467–470.

[46] Romit Jain, Saket Kumar Singh, and Bharavi Mishra. 2019. A Brief Study on Build Failures in Continuous Integration: Causation and Effect. In *Progress in Advanced Computing and Intelligent Engineering*. Springer, 17–27.

[47] Jenkins. 2019. CI Skip Plugin. https://plugins.jenkins.io/ci-skip. [Online; accessed 27-April-2019].

[48] Xianhao Jin. 2021. Reducing cost in continuous integration with a collection of build selection approaches. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1650–1654.

[49] Xianhao Jin and Francisco Servant. 2018. 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*. 70–73.

[50] Xianhao Jin and Francisco Servant. 2019. What edits are done on the highly answered questions in stack overflow? an empirical study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 225–229.

[51] Xianhao Jin and Francisco Servant. 2020. A Cost-efficient Approach to Building in Continuous Integration. In *Proceedings of the 42th International Conference on Software Engineering*. To appear.

[52] Xianhao Jin and Francisco Servant. 2021. 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, 166–167.

[53] Xianhao Jin and Francisco Servant. 2021. 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, 213–225.

[54] Xianhao Jin and Francisco Servant. 2022. Which builds are really safe to skip? Maximizing failure observation for build selection in continuous integration. *Journal of Systems and Software* (2022), 111292.

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

[56] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. 2014. Why do automated builds break? an empirical study. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 41–50.

[57] David Chenho Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. 1995.

Class firewall, test order, and regression testing of object-oriented programs. *JOOP* 8, 2 (1995), 51–65.

[58] Irwin Kwan, Adrian Schroter, and Daniela Damian. 2011. Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. *IEEE Transactions on Software Engineering* 37, 3 (2011), 307–324.

[59] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An extensive study of static regression test selection in modern software evolution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 583–594.

[60] Claire Leong, Abhayendra Singh, Mike Papadakis, Yves Le Traon, and John Micco. 2019. Assessing transition-based test selection algorithms at Google. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 101–110.

[61] Marko Leppänen, Simo Mäkinen, Max Pagels, Veli-Pekka Eloranta, Juha Itkonen, Mika V Mäntylä, and Tomi Männistö. 2015. The highways and country roads to continuous deployment. *Ieee software* 32, 2 (2015), 64–72.

[62] Jingjing Liang. 2018. COST-EFFECTIVE TECHNIQUES FOR CONTINUOUS INTEGRATION TESTING. (2018).

[63] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. 2018. Redefining prioritization: continuous prioritization for continuous integration. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 688–698.

[64] Jackson A Prado Lima and Silvia R Vergilio. 2020. Test Case Prioritization in Continu-

ous Integration environments: A systematic mapping study. *Information and Software Technology* 121 (2020), 106268.

[65] Yiling Lou, Junjie Chen, Lingming Zhang, and Dan Hao. 2019. A survey on regression test-case prioritization. In *Advances in Computers*. Vol. 113. Elsevier, 1–46.

[66] Qi Luo, Kevin Moran, Denys Poshyvanyk, and Massimiliano Di Penta. 2018. Assessing test case prioritization on real faults and mutants. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 240–251.

[67] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive test selection. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. IEEE Press, 91–100.

[68] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. 2013. Test case prioritization for continuous regression testing: An industrial case study. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 540–543.

[69] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming google-scale continuous testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. IEEE Press, 233–242.

[70] Ade Miller. 2008. A hundred days of continuous integration. In *Agile 2008 Conference*. IEEE, 289–293.

[71] Shaikh Mostafa, Xiaoyin Wang, and Tao Xie. 2017. Perfranker: Prioritization of performance regression tests for collection-intensive software. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 23–34.

[72] Armin Najafi, Weiyi Shang, and Peter C Rigby. 2019. Improving test effectiveness using test executions history: An industrial experience report. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 213–222.

[73] Ansong Ni and Ming Li. 2017. Cost-effective build outcome prediction using cascaded classifiers. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*. IEEE, 455–458.

[74] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. *ACM SIGSOFT Software Engineering Notes* 29, 6 (2004), 241–251.

[75] Klérisson VR Paixão, Crícia Z Felício, Fernanda M Delfim, and Marcelo de A Maia. 2017. 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, 479–482.

[76] Cong Pan and Michael Pradel. 2021. Continuous test suite failure prediction. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 553–565.

[77] Gustavo Pinto, Marcel Rebouças, and Fernando Castor. 2017. 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, 74–77.

[78] B Aditya Prakash, Jilles Vreeken, and Christos Faloutsos. 2014. Efficiently spotting the starting points of an epidemic in a large graph. *Knowledge and information systems* 38, 1 (2014), 35–59.

[79] Noam Rabbani, Michael S Harvey, Sadnan Saquif, Keheliya Gallaba, and Shane McIntosh. 2018. Revisiting" Programmers' Build Errors" in the Visual Studio Context. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 98–101.

[80] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 2017. 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, 345–355.

[81] Marcel Rebouças, Renato O Santos, Gustavo Pinto, and Fernando Castor. 2017. 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, 475–478.

[82] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G Ryder, and Ophelia Chesley. 2004. Chianti: a tool for change impact analysis of java programs. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 432–448.

[83] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing regression test selection techniques. *IEEE Transactions on software engineering* 22, 8 (1996), 529–551.

[84] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 6, 2 (1997), 173–210.

[85] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering* 27, 10 (2001), 929–948.

[86] Islem Saidani, Ali Ouni, and Wiem Mkaouer. 2021. Detecting skipped commits in continuous integration using multi-objective evolutionary search. *IEEE Transactions on Software Engineering* (2021).

[87] Andrew I Schein, Alexandrin Popescul, Lyle H Ungar, and David M Pennock. 2002. Methods and metrics for cold-start recommendations. In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*. 253–260.

[88] SciTools Understand. 2020. Understand Static Code Analysis Tool. https://scitools.com/. [Online; accessed 02-March-2020].

[89] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 724–734.

[90] Francisco Servant. 2013. Supporting bug investigation using history analysis. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 754–757.

[91] Francisco Servant and James A Jones. 2011. History slicing. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 452–455.

[92] Francisco Servant and James A Jones. 2012. History slicing: assisting code-evolution tasks. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.

[93] Francisco Servant and James A Jones. 2012. WhoseFault: Automatic Developer-to-

Fault Assignment through Fault Localization. In *International Conference on Software Engineering*. 36–46.

[94] Francisco Servant and James A Jones. 2012. WhoseFault: automatic developer-to-fault assignment through fault localization. In *2012 34th International conference on software engineering (ICSE)*. IEEE, 36–46.

[95] Francisco Servant and James A Jones. 2013. Chronos: Visualizing slices of source-code history. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 1–4.

[96] Francisco Servant and James A Jones. 2017. Fuzzy fine-grained code-history analysis. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 746–757.

[97] August Shi, Suresh Thummalapenta, Shuvendu K Lahiri, Nikolaj Bjorner, and Jacek Czerwonka. 2017. Optimizing test placement for module-level regression testing. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 689–699.

[98] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and improving regression test selection in continuous integration. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 228–238.

[99] Myoungkyu Song and Eli Tilevich. 2009. The anti-goldilocks debugger: helping the average bear debug transparently transformed programs. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. 811–812.

[100] Stack Overflow contributors. 2019. Skip travis build if an unimportant file changed. https://stackoverflow.com/questions/48455623/skip-travis-build-if-an-unimportant-file-changed [Online; accessed 21-February-2019].

[101] Daniel Ståhl and Jan Bosch. 2013. 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)*. 736–743.

[102] Stephen W Thomas, Hadi Hemmati, Ahmed E Hassan, and Dorothea Blostein. 2014. Static test case prioritization using topic models. *Empirical Software Engineering* 19, 1 (2014), 182–212.

[103] Travis. 2019. Skipping a build. https://docs.travis-ci.com/user/customizing-the-build/#skipping-a-build. [Online; accessed 27-April-2019].

[104] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2017. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process* 29, 4 (2017), e1838.

[105] Michele Tufano, Hitesh Sajnani, and Kim Herzig. 2019. 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)* (Montral, Canada) *(ICSE '19)*. 4 pages.

[106] Marko Vasic, Zuhair Parvez, Aleksandar Milicevic, and Milos Gligoric. 2017. File-level vs. module-level regression test selection for. net. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 848–853.

[107] Bogdan Vasilescu, Stef Van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark GJ van den Brand. 2014. Continuous integration in a social-coding world: Empirical evidence from GitHub. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 401–405.

[108] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 805–816.

[109] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. 2017. 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, 183–193.

[110] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. 2019. 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, 647–658.

[111] Wikipedia contributors. 2019. Cold start (computing) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Cold_start_(computing)&oldid=883021431 [Online; accessed 21-February-2019].

[112] Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. 2009. Predicting build failures using social network analysis on developer communication. In *Proceedings*

of the 31st International Conference on Software Engineering. IEEE Computer Society, 1–11.

[113] Zheng Xie and Ming Li. 2018. Cutting the Software Building Efforts in Continuous Integration by Semi-Supervised Online AUC Optimization.. In *IJCAI*. 2875–2881.

[114] Shin Yoo and Mark Harman. 2007. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 140–150.

[115] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.

[116] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. 2017. 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, 334–344.

[117] Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. 2019. A Large-Scale Empirical Study of Compiler Errors in Continuous Integration. (2019).

[118] Lingming Zhang. 2018. Hybrid regression test selection. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 199–209.

[119] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. 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, 60–71.

[120] Hao Zhong and Na Meng. 2017. An empirical study on using hints from past fixes. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 144–145.

[121] Hao Zhong and Na Meng. 2018. Towards reusing hints from past fixes. *Empirical Software Engineering* 23, 5 (2018), 2521–2549.

[122] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A framework for checking regression test selection tools. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 430–441.

[123] Mahdis Zolfagharinia, Bram Adams, and Yann-Gaël Guéhénuc. 2017. Do Not Trust Build Results at Face Value-An Empirical Study of 30 Million CPAN Builds. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 312–322.