HybridCISave: A Combined Build and Test Selection Approach in Continuous Integration

XIANHAO JIN, Virginia Tech
FRANCISCO SERVANT, Universidad de Málaga

Continuous Integration (CI) is a popular practice in modern software engineering. Unfortunately, it is also a high-cost practice—Google and Mozilla estimate their CI systems in millions of dollars. To reduce the computational cost in CI, researchers developed approaches to selectively execute builds or tests that are likely to fail (and skip those likely to pass). In this article, we present a novel hybrid technique (HybridCISave) to improve on the limitations of existing techniques: to provide higher cost savings and higher safety. To provide higher cost savings, HybridCISave combines techniques to predict and skip executions of both full builds that are predicted to pass and partial ones (only the tests in them predicted to pass). To provide higher safety, HybridCISave combines the predictions of multiple techniques to obtain stronger certainty before it decides to skip a build or test. We evaluated HybridCISave by comparing its effectiveness with the existing build selection techniques over 100 projects and found that it provided higher cost savings at the highest safety. We also evaluated each design decision in HybridCISave and found that skipping both full and partial builds increased its cost savings and that combining multiple test selection techniques made it safer.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; **Maintaining software;**

Additional Key Words and Phrases: Software maintenance, Continuous Integration, build selection, test selection

ACM Reference format:
Xianhao Jin and Francisco Servant. 2023. HybridCISave: A Combined Build and Test Selection Approach in Continuous Integration. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 93 (May 2023), 39 pages.
https://doi.org/10.1145/3576038

1 INTRODUCTION

Continuous Integration (CI) is a widely used practice in modern software engineering that encourages developers to check in, build, and test their code in frequent intervals [29], sometimes as frequently as building and testing for every code change. However, while CI is widely recognized as a valuable practice, it also incurs an expensive cost—due to the high computational workloads created by frequently executing software builds [44, 46, 47, 81, 111]. For simplicity and consistency

F. Servant performed some of this work while at Virginia Tech and Universidad Rey Juan Carlos.
This material was based upon work supported by the National Science Foundation under award CCF-2046403, and by Universidad Rey Juan Carlos under an International Distinguished Researcher award C01INVESDIST.
Authors' addresses: X. Jin, Virginia Tech, 2202 Kraft Drive SW, Blacksburg, VA 24060; email: xianhao8@vt.edu; F. Servant, Universidad de Málaga, Málaga, Spain; email: fservant@uma.es.

1049-331X/2023/05-ART93 $15.00
https://doi.org/10.1145/3576038

with previous studies [1, 51], we will use the term *build* to refer to a *full build* that includes all stages of build preparation (e.g., compilation), and testing. We will also use the term *execution* to refer to the execution of either builds or tests.

Some research approaches have been proposed to reduce the computational cost of CI (e.g., [1, 2, 44, 51, 67]). The goal of these techniques is to skip as many as possible of the (build or test) executions that provide lower value for developers while still executing as many as possible of those that are more valuable. Many of the existing techniques thus aim to skip (build or test) executions that pass while still executing as many (build or test) executions that fail—since they provide feedback about an issue that needs to be resolved (e.g., [44, 50, 51, 53, 67]).

Build or test selection techniques are evaluated in terms of their provided *cost savings* (i.e., the ratio of executions that they choose to skip), and in terms of their *safety* (i.e., the ratio of failing executions that they choose to not skip). All build or test selection techniques provide some trade-off between cost savings and safety [53]. For example, if they are more inclined to choose to skip executions, they will save more cost, but they will also more likely skip failing executions—and delay their observation.

In this article, we present a novel approach for reducing the computational cost of CI: HybridCISave. We designed HybridCISave to improve on the limitations of existing techniques: to provide higher *cost savings* and higher *safety*.

To provide high cost savings, we designed HybridCISave to skip builds both fully and partially. Existing cost-saving techniques for CI selectively skip executions at either the *build* [1, 2, 51] or *test* [39, 44, 67, 101] granularities. Build selection techniques skip *full builds*—skipping both the build preparation steps and all tests for the project. Test selection techniques skip *partial builds*—executing the build preparation steps, but skipping some of the tests. To the extent of our knowledge, HybridCISave is the first technique to combine both approaches.

To provide high safety, HybridCISave contains other design decisions within its components HybridBuildSkip and HybridTestSkip, which we describe next.

HybridCISave is a hybrid technique that applies two separate components: HybridBuildSkip to predict which builds to skip fully, and HybridTestSkip to predict which builds to skip partially. We depict HybridCISave in Figure 1. First, for a given software build, HybridCISave applies HybridBuildSkip to predict if the build will pass. If so, HybridCISave skips the build fully (i.e., all its preparation steps and test executions), saving its cost. Second, if HybridBuildSkip instead predicted the build to fail (and thus should not be skipped fully), then HybridCISave will still try to save *partial* cost in that build. In that case, HybridCISave applies HybridTestSkip to each individual test, and predicts whether it will pass (and skip its execution, saving its cost) or it will fail (and execute it).

We designed HybridBuildSkip and HybridTestSkip with the same goals as HybridCISave: to provide high cost savings and high safety.

To provide high cost savings, we designed HybridBuildSkip also as a hybrid technique. HybridBuildSkip applies a collection of build and test selection techniques and then uses their individual outcomes as features for a machine learning predictor (Random Forest). Our intuition behind this design is that HybridBuildSkip's predictor could learn to counterbalance the incorrect predictions of some techniques with the correct predictions of others, to make more correct predictions. HybridTestSkip also helps provide high cost savings by skipping partial builds in addition to the full builds skipped by HybridBuildSkip.

To provide high safety, HybridBuildSkip takes two measures. First, it applies build selection techniques, which were originally designed to provide high safety, by aiming to skip builds that developers would want to skip (CI-Skip Commits [1, 2, 88]) or by skipping builds that they predict will pass with high certainty [51]. Second, HybridBuildSkip applies test selection techniques to

predict build outcomes in a highly safe way: it models that a test selection technique predicts a build to pass if it predicts all its individual tests to pass. HybridTestSkip also contributes to providing high safety by only skipping a test when multiple test selection techniques agree that the test should be skipped.

The novelty of our proposed approach (HybridCISave) lies in the many specific decisions that went into its design (described previously). There are many other ways in which multiple techniques could have been combined into a hybrid one. We designed our hybrid technique in a specific, novel way, in which we made specific decisions following a rationale that aimed to achieve desirable properties: high cost savings and high safety.

We designed our evaluation of HybridCISave to study the extent to which our specific design decisions in HybridCISave actually achieved the desirable properties that we aimed to achieve. Thus, our evaluation contributes empirical evidence of the benefits provided by our specific design decisions in HybridCISave.

First, we studied whether HybridCISave provides higher cost savings and safety than existing techniques (RQ1). We compared HybridCISave with all other build selection techniques. In this part of our evaluation, we found that HybridCISave provided higher cost savings than all previous build selection techniques, at the same (or higher) levels of safety. Additionally, HybridCISave provided higher safety than the safest of our studied techniques (Abdalkareem20). In its safest configuration, HybridCISave could save 14% of build *duration* (i.e., time spent executing builds, including all their steps, e.g., preparing the build environment, compiling, and running tests) while still executing 100% of failing builds (in the median case).

Second, we studied whether the specific design in which we combined techniques in HybridCISave provided higher cost savings and safety than alternative designs (RQ2, RQ3, RQ4, and RQ5). For that, we performed an ablation study in which we separately substituted each one of our design decisions for an alternative one and observed whether performance worsened.

We found in RQ2 that our decision of adding a component to skip partial builds in addition to full builds (HybridTestSkip and HybridBuildSkip) provided higher cost savings than not having the component to skip partial builds (HybridBuildSkip only). We also found in RQ3 that our decision of combining test selection techniques with build selection techniques to selectively skip builds (within HybridBuildSkip) provided higher cost savings than not using them. Then, we found in RQ4 that our decision to use a Random Forest classifier to combine techniques in HybridBuildSkip (as opposed to other machine learning models or to a simple voting scheme) provided higher prediction effectiveness. In RQ5, we found that our design of HybridTestSkip to combine test selection techniques with high safety did provide higher safety than other test selection techniques.

Third, we evaluated the relative importance of HybridBuildSkip's and HybridTestSkip's features (RQ6, RQ7). We found that the most important feature for skipping builds was Subsequent Failures, and that Gligoric15 had the strongest importance when skipping tests. This is because a large portion of failing builds happened after another build failure (40%), and more than 90% of builds with no test to cover the changes (which is the criterion used by Gligoric15 to skip tests) were passing builds. We also observed that the least important feature to predict the outcome of test executions was Herzig15, since it focused on skipping builds with long execution times, and our dataset contained few of these.

Finally, we studied the risk of HybridCISave's execution time taking longer than the process that it aims to skip: executing the build themselves (RQ8, RQ9). HybridCISave requires to obtain and combine multiple predictions, which could potentially take long and thus defeat its purpose. 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, whereas the median build duration in our studied dataset was 441.5 seconds. Thus, HybridCISave's

execution time only reduced its savings by up to 0.016 pp. This article provides the following contributions:

- The first hybrid approach to saving computational cost in CI (HybridCISave), which skips full and partial builds.
- Empirical evidence that the novel design in HybridCISave provides high cost savings and high safety.
- Empirical evidence that the specific design decisions that went into HybridCISave individually contributed to it providing high cost savings and high safety.
- Empirical evidence that the complex, hybrid design of HybridCISave (combining multiple other approaches) requires only negligible execution time when compared to the time that it saves.

2 RELATED WORK

2.1 Empirical Studies of CI and Its Cost

Multiple past works focused on understanding the practice of CI, in dimensions related to both practitioners (e.g., [47]) and software repositories [107].

Ståhl and Bosch [103] and Hilton et al. [47] studied the benefits and costs of CI usage, and the trade-offs between them [46]. Elazhary et al. [21] also studied benefits and challenges of CI practices [21]. Widder et al. [111] identified the pain points in using Travis CI. Zampetti et al. [119] identified the problems and solutions of applying CI to a specific domain: cyber-physical systems, and Soares et al. [102] studied the impact of CI on software engineering more generally, surveying the research literature. Gallaba et al. [32] studied the operational aspects of a specific CI platform (CircleCI), such as how much time each build stage takes, and how many builds are canceled. Leppänen et al. [62] studied the costs and benefits of a related practice: continuous delivery.

Other studies focused on the barriers [81] and challenges [18] for CI adoption. Felidré et al. [27] studied the adherence of projects to the original CI rules [29], and Gallaba and McIntosh [34] and Zampetti et al. [121] characterized complementary collections of bad practices in CI.

Zhao et al. [125] studied the impact of CI in other development practices, like bug-fixing and testing. Vasilescu et al. studied CI as a tool in social coding [106], and later studied its impact on software quality and productivity [107]. Wang et al. [110] also studied the impact of CI in product quality, particularly as test automation matures.

Our work in this article aims to reduce the high computational cost of using CI. This high cost was observed in multiple past studies [44, 46, 47, 81, 111], and it can reach millions of dollars in large companies, such as at Google [47] and Microsoft [44]. Past studies also highlighted long waiting times as the main pain point of CI in those companies [63]—which would also be reduced if fewer builds get executed.

2.2 Characterizing Builds

There are many existing works aiming to characterize builds. Some studies investigated the reasons for build failures. Some studies [73, 108] sort common build failures into compilation [122], unit test, static analysis [120], and server errors. Paixão et al. [76] studied the interplay between non-functional requirements and failing builds. Other studies found factors that contribute to build failures: architectural dependencies [13, 93], and other more specific factors, such as the stakeholder role, the type of work [59], the programming language [12], the interactions among developers [91, 113], or their socio-technical congruence [60]. Other less obvious factors that could cause build failures are build environment changes or flaky tests [82]. Some work [51, 82] also found that build failures tend to occur consecutively, which Gallaba et al. [33] describe as "persistent build breaks."

Other works aimed to characterize builds to predict build outcomes. Hassan and Zhang [41] measured social, technical, coordination, and prior-certification characteristics of builds, and applied decision trees to predict their outcome. Finlay et al. [28] characterized builds according to complexity, Halstead, and basic metrics to also predict their outcome with a Hoeffding tree. Xia et al. [114, 115] and Luo et al. [66] measured characteristics of a build and its project, and used them to evaluate multiple machine learning predictors to predict build outcomes. Xie and Li [116] improved the prediction of previous approaches by applying online AUC optimization. Saidani et al. [87] also measured build and team characteristics, and applied a deep learning classifier. Hassan and Wang [42] found that features related to the previous build are most effective when predicting build outcomes. Ni and Li [75] and Chen et al. [15] also found that some specific historical features (like the last build status) were more useful than other features about the current build (like the number of files changed) for predicting the build's outcome. Other studies found change characteristics that correlate with failing builds, such as code churn [48, 82], short dependency distance from the tests [70], build tool [48], and statistics on the last build, and the history of the committer [75]. Barrak et al. [6] found that features about code smell can be effective for predicting build outcomes. Santolucito et al. [90] predict builds that would fail due to configuration errors, using static analysis. Finally, Ghaleb et al. [38] studied the impact of noisy data on the results of build outcome predictors.

In contrast with these past works that aim to understand why builds fail or to characterize failing builds, our work (HybridCISave) has a different goal: reducing the cost of CI by skipping passing executions. This different goal motivated multiple design decisions in HybridCISave that were not present in these past works. First, HybridCISave has the incentive to be highly safe—that is, it prefers to make mistakes by executing a passing build than by skipping a failing build. For this incentive, HybridCISave combines multiple build selection techniques that were designed to be safe, and it requires multiple test selection techniques to agree before it decides to skip a test. Second, HybridCISave skips the execution of both builds and tests. Since its goal is to reduce the cost of CI, it values skipping both kinds of executions. Third, HybridCISave combines the predictions of past build selection techniques that follow multiple criteria: skipping builds that will likely pass (e.g., Jin20), and skipping builds that developers may decide to skip (e.g., Abdalkareem19). This way, it diversifies its opportunities to save cost in CI.

2.3 Approaches to Reduce the Cost of CI

Past efforts to reduce the cost of CI focused on either executing fewer builds, or on reducing the computational cost of each build. Toward this latter goal, past efforts focused on understanding what causes long build duration (e.g., [37, 105]), and on reducing it, by skipping test executions or other steps within a build.

The techniques that save cost in CI by running fewer builds (i.e., build selection techniques) follow two main strategies: skipping builds that they predict that developers will want to skip (CI-Skip Commits) [1, 2, 88], or skipping builds that they predict to pass [51, 55]. Abdalkareem et al. studied the characteristics of CI-Skip Commits, and created techniques to predict and skip them, based on rules [2] and machine learning [1]. Saidani et al. [88] proposed a technique to predict and skip CI-Skip Commits using an evolutionary algorithm. Jin and Servant focused on skipping builds that they predict to pass, using machine learning with features correlated with build passes [51], and with build passes and CI-Skip Commits [55].

In contrast with the techniques to predict build outcomes that we described in Section 2.2, these techniques that predict and skip builds have the incentive to be highly safe (i.e., to incur a low ratio of mistakes even if it means skipping fewer builds). The techniques that skip CI-Skip Commits (e.g., [1, 2, 88]) aim to provide high safety by focusing on predicting and skipping builds that developers

would want to skip. The techniques that skip builds that pass (e.g., [51, 55]) aim to provide high safety by providing a customizable prediction sensitivity threshold that can be used to make them inclined to skip fewer builds (and thus make fewer mistakes).

When compared with the previous build selection techniques (e.g., [1, 2, 51, 55, 88]), our proposed technique (HybridCISave) aims to provide higher cost savings, since it is a hybrid build and test selection technique. HybridCISave can skip the execution of full builds (as build selection techniques do), and it can also skip some of the tests in the builds that it decides not to skip fully (which build selection techniques do not do). Additionally, HybridCISave's build selection component (HybridBuildSkip) is also different from previous build selection techniques. HybridBuildSkip is different from Abdalkareem19 [2], Abdalkareem20 [1], and Jin20 [51] individually: HybridBuildSkip uses them as some of the features in its predictor (see Section 3.1). HybridBuildSkip is also different from Saidani21 [88]: HybridBuildSkip uses a Random Forest classifier (see Section 3.1), and Saidani21 uses an evolutionary algorithm with different features (see Section 5.1.1). HybridBuildSkip is also different from Jin22 [55]: Jin22 uses as features Abdalkareem19's CI-Skip Rules in addition to 4 CI-Run Rules that they defined (see Section 5.1.1), and HybridBuildSkip uses as features the output of Abdalkareem19's predictor in addition to the output of 5 other predictors (see Section 3.1).

Other existing techniques aim to reduce cost in CI by skipping tests within each build. These techniques focused on skipping, e.g., tests that historically failed less [44], that test unchanged classes [101] and modules [25, 100, 101], that are predicted to pass by a machine learning classifier [67], skipping complete test suites [77], based in a particular industrial context [69], or in a specific programming language [56]. Recent work also proposed a framework to evaluate and compare these techniques [24]. These techniques are based on regression test selection (e.g., [39, 78, 84, 85, 117, 118, 124, 126]), which was a popular field before CI was proposed.

This strategy of skipping tests within builds provides different benefits to the strategy of skipping full builds. Skipping tests within builds enables the possibility of saving computational cost on all builds (both in those that pass and in those that fail). However, it misses the opportunity of skipping other steps in a build (e.g., build-preparation steps), which are saved by the techniques that skip full builds [52, 53]. Our proposed technique (HybridCISave) leverages the benefits of both strategies (skipping full and partial builds) by combining them.

A related effort for improving CI targets prioritizing its tasks to provide early fault observation. The most common approach in this direction is to apply test case prioritization techniques (e.g., [22, 23, 65, 68, 74, 86]) so that builds fail faster. Another similar approach achieves faster feedback by prioritizing builds instead of tests [64] when there is a queue of builds waiting to be executed due to having limited computation resources.

Prioritization-based techniques advance feedback but are not able to save cost (i.e., all builds and tests still get executed). In contrast, our work aims at saving cost in CI by skipping (build or test) executions.

Other existing techniques aim to reduce cost in CI by skipping other steps within a build. Some skip retrieving unnecessary files from within dependencies [14], and others skip unnecessary steps to prepare the build environment [31, 35]. A final strategy aims to reduce cost in CI by batching together multiple builds [7, 8] or multiple targets [109].

In contrast to these approaches, our proposed work HybridCISave aims to reduce cost in CI by skipping full and partial builds. We chose this focus because skipping the execution of (all or some) tests provides the highest opportunity for cost savings—most build execution time is spent executing tests (91.5%) [31]. Still, future work could explore adding further extensions to HybridCISave to also skip other steps within a build, such as unnecessary build preparation or compilation steps.

2.4 Other Automated Support for CI

Techniques have been proposed to support CI in other ways. For example, Ziftci and Reardon [127] proposed a technique to automatically locate the changes that induced a failure in CI. Hassan and Wang [43] proposed a technique to automatically fix build failures caused by errors in build scripts. Zhang et al. [123] proposed a technique to cluster builds that fail for the same reason. Tronge et al. [104] proposed a technique to adapt performance testing of HPC applications to CI.

3 OUR APPROACH: HYBRIDCISAVE

Here, we describe our novel hybrid build-and-test selection technique: HybridCISave. We represent our design of HybridCISave in Figure 1. HybridCISave is a hybrid technique that consists of two separate components: HybridBuildSkip to predict which builds to skip fully (depicted in the top half of Figure 1 and described in Section 3.1), and HybridTestSkip to predict which builds to skip partially (depicted in the bottom half of Figure 1 and described in Section 3.2). First, for a given software build, HybridCISave applies HybridBuildSkip to predict if the build will pass. If so, HybridCISave skips the build fully (i.e., all its preparation steps and test executions), saving its cost. Second, if HybridBuildSkip instead predicted the build to fail (and thus should not be skipped fully), then HybridCISave will still try to save *partial* cost in that build. In that case, HybridCISave applies HybridTestSkip to each individual test in the build, and predicts whether it will pass (and skips its execution, saving its cost) or it will fail (and executes it).

3.1 HybridBuildSkip

Given a build, HybridCISave first executes HybridBuildSkip to predict if the build will pass, and if so, skip it and save its cost. We depict HybridBuildSkip's design in the top half of Figure 1. For a given build, HybridBuildSkip extracts the characteristics of its changes, and of its individual tests. Then, it uses those characteristics as features for six existing techniques: four build selection techniques [1, 2, 51] and two test selection techniques [39, 67]. Each of these techniques uses a different set of features. HybridBuildSkip executes all six techniques, and then uses their six resulting predictions as features for a final machine learning predictor (Random Forest), to obtain a final prediction. 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. If the final prediction is that the build will pass, HybridBuildSkip skips its execution. If the final prediction is that the build will fail, HybridBuildSkip tries to save some of its cost by applying HybridTestSkip (see Section 3.2).

We train HybridBuildSkip as a cross-project predictor—that is, we train it on the past builds of different software projects than the one in which we apply it. This allows it to train on a larger amount of data, and it aids in the cold-start problem [112] (i.e., in software projects where only a few builds have been executed). We also make HybridBuildSkip customizable—that is, we can customize its prediction sensitivity threshold to varying levels of tolerance to skipping failing builds. Most machine learning algorithms return a probability that an instance belongs to a class. The prediction sensitivity threshold can be used to determine how high the predicted probability needs to be for the algorithm to actually classify the instance into the class. For example, setting HybridBuildSkip's prediction sensitivity threshold to 0.1 means that if the technique predicts that a build is at least 10% likely to fail, then HybridBuildSkip will classify the build as a potential failure that should be run (and not skipped). Thus, lower prediction-sensitivity thresholds will make a technique more inclined to predict builds to fail, and higher prediction-sensitivity thresholds will make a technique more inclined to predict builds to pass. Next, we describe the six techniques that HybridBuildSkip uses for its features.

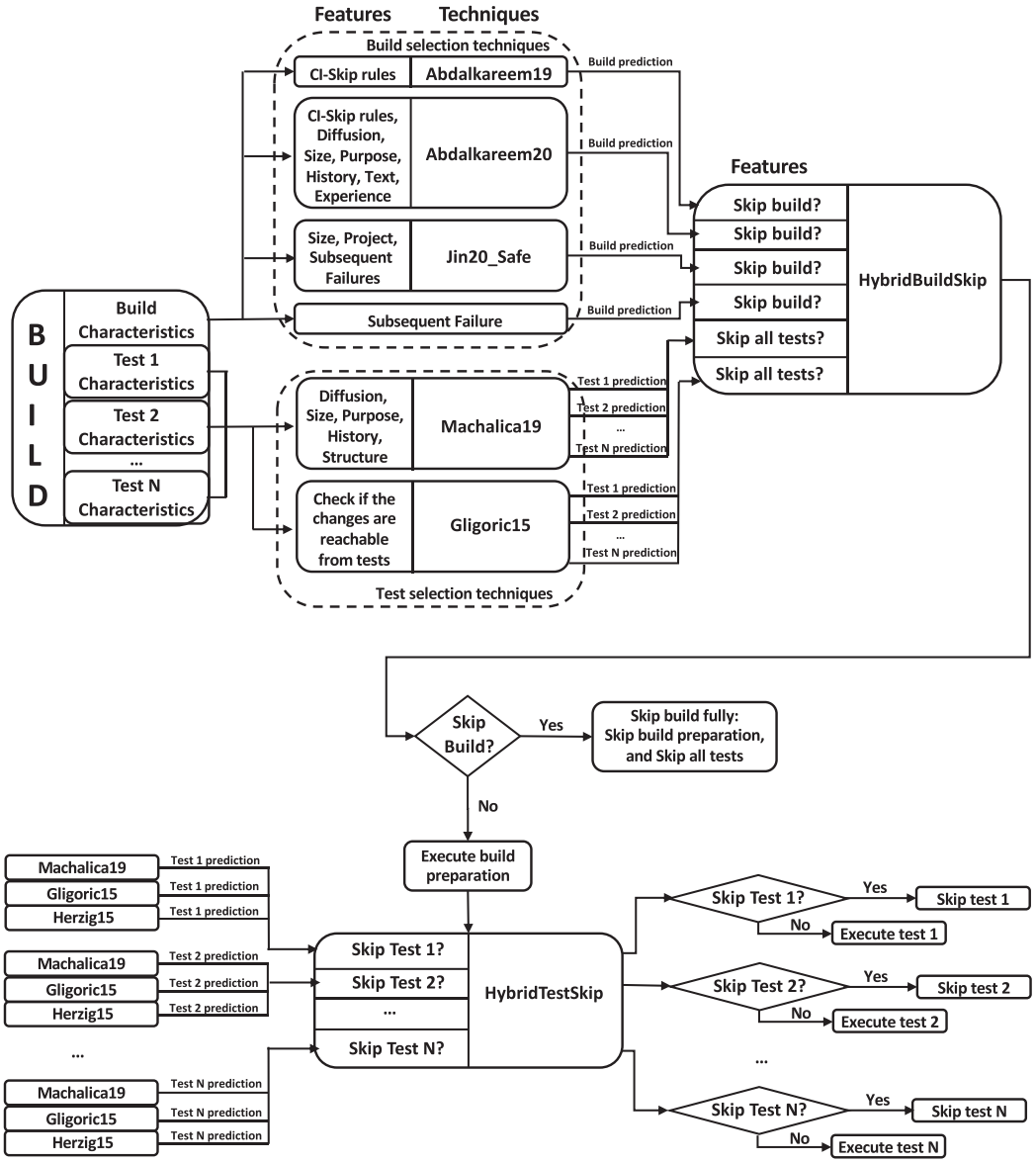

Fig. 1. Flow chart of the design of HybridCISave.

3.1.1 Build-Selection-Based Features. The first four features of HybridBuildSkip contain the prediction produced by four build selection techniques. To provide high safety, we decided to use all existing build selection techniques, which were originally proposed for high safety: two techniques that skip only the builds that they predict that developers will want to skip (CI-Skip Commits): Abdalkareem19 [2] and Abdalkareem20 [1]; and one technique that skips the builds that it predicts to pass with high confidence: Jin20_Safe [51]. We also used one technique that skips builds that are not subsequent to a build failure: Subsequent Failures.

One other build selection technique was recently proposed to skip the builds that developers would want to skip (CI-Skip Commits): Saidani21 [88]. However, we decided to not include it in

our approach because of its high execution time. In our studied dataset, obtaining Saidani21's prediction about a build took longer time than actually running the build itself (defeating the purpose of the prediction), for 52% of our studied projects. We also did not include in HybridBuildSkip's design another build selection technique that skips builds that it predicts to pass, since it was not published at the time of designing our technique [55]. Next, we provide more details about the four build selection techniques that HybridBuildSkip uses as features.

F1: Abdalkareem19 [2] is a build selection approach that uses a collection of rules (CI-Skip Rules) to skip builds that developers often decide to skip (CI-Skip Commits). Abdalkareem19 skips builds that only changed: source comments, formatting, non-source files, meta-files, or version-release files.

F2: Abdalkareem20 [1] is a build selection approach that also skips builds that developers often decide to skip (CI-Skip Commits), but it now uses a machine learning predictor (decision tree). Abdalkareem20 informs its predictor with CI-Skip Rules, but also with features of Diffusion (number of modified subsystems, directories, files, and their distribution), Size (lines of code added, deleted, in a file before the change, and the types of files changed), Purpose (whether the change is a bug fix, a merge commit, or another maintenance activity according to keywords like "refactoring"), History (number of developers that changed the modified files, time between last and current change, and number of unique changes to the modified files), Experience (developer experience, recent developer experience, and developer experience in the modified subsystems), and Text (weighted terms in the commit message using TF-IDF).

F3: Jin20_Safe [51] is a build selection approach that skips builds that it predicts will pass with high confidence. Jin20_Safe follows a two-phase algorithm. First, it applies machine learning (Random Forest) to predict whether a build will pass, and if so, skip it. We used the safest variant (Jin20_Safe, with a prediction sensitivity of 0%) so that it only skips builds when it is 100% confident that they will pass. Jin20_Safe's second phase is triggered when it executes a build and it observes that it fails. In such case, it continues executing the next build, until it observes that it passes, and then goes back to the predicting phase. To make its predictions, Jin20_Safe uses features of Size (the number of changed lines, files, tests, and commits in the build) and Project (the number of executable source lines and tests in the project, and the age of the project).

F4: Subsequent Failures is a feature that skips builds that it predicts to pass. This is a simple feature that predicts all builds to pass, except those that are subsequent to a build failure. This is based on the observation that build failures are often followed by other build failures, and build passes are often followed by other build passes [42, 51, 75]. We decided to include this feature alongside other build selection techniques because we expect it to be strongly predictive. As an additional note, in our experiments, this feature uses realistic information—that is, it can only see the status of the last build that was actually executed (not skipped).

3.1.2 Test-Selection-Based Features. The next two features of HybridBuildSkip contain the prediction produced by two test selection techniques. To provide high cost savings, we also use features from test selection techniques in addition to the features from build selection techniques. HybridBuildSkip uses a test selection technique that was originally proposed for CI (Machalica19 [67]), and a popular test selection technique that was proposed outside the context of CI (Gligoric15 [39]). To provide high safety, we adapted these test selection techniques to predict that a build should be skipped only if it predicts that all its tests should be skipped. This makes these techniques safer, since they only decide to skip a build when they predict that every single one of its tests should be skipped.

Although other test selection techniques were proposed for CI, we decided to not include them in our design for various reasons. The techniques of Shi et al. [100] and Elsner et al. [25] are

adaptations of Gligoric15 (which we already include in our design), to proprietary environments, and to skip tests according to both changed classes and modules [101]. The technique of Kauhanen et al. [56] was designed to work only with Python, and that of Martins et al. [69] was proposed to work in a proprietary environment. The technique of Pan and Pradel [77] does not provide predictions for individual tests, only a single prediction about the whole test suite, which does not allow us to adapt it for higher safety by aggregating individual test predictions, as we did for other test selection techniques. The technique of Herzig et al. [44] is inclined to skip tests with long execution times (see Section 3.2.1), making it unlikely to skip all tests within a build (we believe that rarely every single test in a build takes long to execute). So, we decided to use that of Herzig et al. for HybridTestSkip (see Section 3.2), but not for HybridBuildSkip. Finally, Elsner et al. [24] presented a framework to evaluate test selection techniques, but not a technique itself. Next, we provide more details about the two test selection techniques that HybridBuildSkip uses as features.

F5: Gligoric15 [39] is a test selection technique that skips tests that do not exercise the changed files in the build. The intuition for this technique is that tests that do not exercise changed files cannot detect faults in them. Gligoric15 tracks the files that each test exercises dynamically—that is, it monitors the execution of the tests and the code under test to collect the set of files accessed during execution of each class, computes the checksum for these files, and stores them in a dependency file. The accessed files can include either executable code (e.g., class files in Java) or external resources (e.g., configuration files).

F6: Machalica19 [67] is a test selection technique that skips tests that it predicts to pass. Machalica19 predicts which tests will pass using a gradient boosted decision trees classifier (XGBoost) with features of Change (history of changed files, number of changed files, number of targets using the changed files, changed file extensions, and number of authors), Target (historical failure rate, project name, and number of tests in the target), and Cross (minimal distance between the changed files and the prediction target, and number of common tokens between the path of the modified files and the path of the test).

3.2 HybridTestSkip

When HybridBuildSkip predicts that a build will fail and thus should not be skipped, HybridCISave then executes HybridTestSkip to try to skip some of its tests (those that it predicts to pass). HybridTestSkip allows HybridCISave to increase its cost savings, by saving some cost even from builds that it predicts to fail. We also designed HybridTestSkip to provide high safety: it predicts a test to pass when multiple test selection techniques predict it to pass.

We depict HybridTestSkip's design in the bottom half of Figure 1. For a given build that was predicted to fail, HybridTestSkip iterates over its tests. For each test, it executes three test selection techniques [39, 44, 67] and uses their three resulting predictions to provide a final prediction. HybridTestSkip predicts that a test will pass if all three test selection techniques predict that it will pass. Otherwise, it predicts that the test will fail. If the final prediction is that the test will pass, HybridTestSkip skips its execution. If the final prediction is that the test will fail, HybridTestSkip executes it.

HybridTestSkip does not require training itself, since it is rule based. Only the test selection techniques that it uses need to be trained.

3.2.1 Features in HybridTestSkip. HybridTestSkip informs its prediction with the same test-selection-based features as HybridBuildSkip, using the same rationale (see Section 3.1.2)—with the difference that this time HybridTestSkip provides individual predictions per test instead of per build. Another difference is that HybridTestSkip uses a third test selection technique that

was proposed for CI: Herzig15 [44]. We decided to not use Herzig15 as part of HybridBuildSkip because it would rarely skip all tests in a build (it tends to skip mostly tests with long execution times, as we elaborate in the following), but we decided that it would be useful to use it for HybridTestSkip, to save the execution time of some tests. We already described Gligoric15 [39] and Machalica19 [67] in Section 3.1.2. We describe Herzig15 next.

Herzig15 [44] is a test selection technique that skips tests for which it expects that executing them ($Cost_{exec}$) has higher cost than skipping them ($Cost_{skip}$) [45]. As a result, Herzig15 tends to skip tests that have a long execution time.

Herzig15 estimates the cost of executing a test ($Cost_{exec}$) based on the cost of machine time and the cost of inspecting a potential failure that was not caused by a defect, using the following formula.

$$Cost_{exec} = Cost_{machine} + P_{FP} * Cost_{inspect} \quad (1)$$

$Cost_{machine}$ is a constant representing the per minute infrastructure cost. Herzig et al. set it at 0.03 $/minute in their experiments.

P_{FP} is the probability that the combination of test and execution context will report a false alarm, that the executed test will fail due to any other reason than a defect. Herzig et al. set it at 0.04 in their experiments.

$Cost_{inspect}$ is a constant representing the average cost rate of test failure inspections. Herzig et al. set it at $9.60 in their experiments.

Herzig15 estimates the cost of skipping the test ($Cost_{skip}$) based on the human cost of fixing a failure that has been unobserved for some time delay, as per the following formula.

$$Cost_{skip} = P_{TP} * Cost_{escaped} * Time_{delay} * \#Engineers \quad (2)$$

P_{TP} is the probability that the combination of test and execution context will detect a defect. Herzig et al. set it at 0.07 in their experiments.

$Cost_{escaped}$ is a constant representing the average cost of an escaped defect (an unobserved failure). Herzig et al. set it at $4.20 per developer and hour of delay in their experiments.

$Time_{delay}$ is the average time span required to fix historic defects. In our experiments, we measured this value as the average time window between a build failing and later passing, averaged for all builds in each studied project.

$\#Engineers$ is the number of engineers in the project. In our experiments, we used the "team size" value reported in TravisTorrent for each project.

We replicated Herzig15 using the same values for the constants in the formulas that they used in their experiments.

4 RESEARCH QUESTIONS

We evaluated HybridCISave in multiple ways, over a large dataset of CI builds. First, we compared its effectiveness with that of all existing build selection techniques (RQ1).

Next, we evaluated the extent to which HybridCISave's design contributed to providing high cost savings and high safety. We evaluated how much the decision of skipping partial builds in addition to full builds provided higher cost savings than not doing it (RQ2). We also evaluated the decision of combining test selection techniques with build selection techniques to achieve higher cost savings (RQ3). We also evaluated the decision of using a Random Forest classifier as opposed to other machine learning models (RQ4). Then, we also evaluated the decision of combining multiple test selection techniques to achieve higher safety (RQ5).

Finally, we also studied the relative importance of HybridBuildSkip's (RQ6) and HybridTestSkip's (RQ7) features, and the impact of the time that HybridCISave takes to make its predictions

over the cost savings that it produces (RQ8 and RQ9). 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: Ablation Study: Evaluating HybridCISave's design decisions

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 benefit of using a Random Forest classifier as opposed to other machine learning models in HybridBuildSkip?

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

Experiment 3: Evaluating the importance of HybridCISave's feature techniques

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

RQ7: What is the relative importance of each feature technique in HybridTestSkip?

Experiment 4: Measuring HybridCISave's execution time

RQ8: What is the total execution time of HybridCISave and its individual components?

RQ9: How much cost does HybridCISave save if we account for its execution time?

4.1 Studied Dataset

We performed our study over the Travis Torrent dataset [12] in its latest version, the snapshot for January 25, 2017 [11], which includes 1,359 projects (423 Java projects and 936 Ruby projects). For our experiments, we curated this dataset in multiple ways.

We removed "toy projects" from the dataset by studying those that are more than 1 year old, and that have at least 200 builds and at least 1,000 lines of source code, which are criteria applied in multiple other works [48, 75]. To be able to explore test information, we also filtered out those projects whose build logs do not contain any test information. We focused our study on builds with passing or failing outcome, rather than error or canceled. Besides, in Travis, a single push or pull request can trigger a build with multiple jobs, and each job corresponds to a configuration of the building step. As many existing papers have done [33, 49, 83], 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 (495,322 build instances) from 100 projects (82 Java projects and 18 Ruby projects). We provide multiple statistics about the studied projects in Table 1.

To be able to implement our approach and replicate existing work, we extended the information in TravisTorrent of the 100 projects that resulted from our filtering process in multiple ways. First, we built scripts to download the raw build logs from Travis and parse them to extract all 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 such as changed file names and changed line content, by running scripts to read the content of commits provided by TravisTorrent for each build using GitHub's API. Finally, we built a dependency graph for the source code of each project using a static analysis tool (Scitool Understand [92]) 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

Table 1. Characteristics of the Studied Projects

Project Type	# Projects	Stat	LOC	# Builds	Build Duration (s)	Build Fail Ratio
Complete dataset	100	Min	1,402	201	48	0.4%
		Median	37,872	526	610	7.5%
		Max	1,033,909	14,133	11,347	63.0%
		Cumulative	4,639,716	82,427	280,143,720	16.2%
Java projects	82	Min	1,402	201	48	0.4%
		Median	13,733	398	347	6.0%
		Max	1,033,909	4,815	2,593	63.0%
		Cumulative	308,247	49,318	35,707,512	11.8%
Ruby projects	18	Min	3,154	212	104	1.3%
		Median	24,139	1,111	1,808	14.1%
		Max	533,998	14,133	11,347	58.1%
		Cumulative	1,557,289	33,109	244,436,208	22.8%

graph using Rubrowser [26]. We used a project's static dependency graph to check if there is a path between changed files and test files.

5 EXPERIMENT 1: EVALUATING HYBRIDCISAVE

In the first experiment, we measured the cost reduction and failure observation provided by HybridCISave.

5.1 RQ1: How Effective Is HybridCISave Saving Cost and Observing Failures, Compared to Existing Build Selection Approaches?

5.1.1 Research Method. We applied HybridCISave in a large dataset (Section 4.1), and we compared its effectiveness with that of all other build selection techniques. The goal of this experiment is to understand whether the design decisions that we included in HybridCISave helped it achieve higher effectiveness than the other techniques that follow the same strategy of reducing the cost of CI by skipping builds. Other strategies to reduce the cost of CI exist (e.g., by skipping some build preparation steps, such as some environment setup or compilation steps [14, 31, 35]). These other strategies are complementary to build selection and can be used in combination with any build selection technique. Future work could explore the impact of these combinations.

Studied Techniques. To compare with HybridCISave, we replicated all other build selection techniques: Abdalkareem19, Abdalkareem20, Jin20, Saidani21, and Jin22.

HybridCISave: Our proposed approach (see Section 3). Since its HybridBuildSkip component is customizable (see Section 3.1), we evaluated it for multiple prediction-sensitivity thresholds: 0 to 0.1 in intervals of 0.001 (101 thresholds in total). Higher prediction-sensitivity thresholds make HybridBuildSkip more likely to predict builds to pass.

Abdalkareem19 [2]: A rule-based build selection approach based on CI-Skip Rules: rules that characterize builds that are likely to be skipped by developers (more details in Section 3.1.1).

Abdalkareem20 [1]: A machine learning predictor (Random Forest) using Abdalkareem19's CI-Skip Rules as features (more details in Section 3.1.1). We picked its Random Forest variant since it is reported as the best performance classifier for Abdalkareem20 [1].

Jin20 [51]: A two-phase build selection approach, using a Random Forest classifier with size and project features (more details in Section 3.1.1). Jin20 is also customizable in its prediction sensitivity threshold, so we also study multiple thresholds for it (including its safest variant, Jin20_Safe).

Saidani21 [88]: A build selection approach that predicts and skips builds for CI-Skip Commits based on the adaptation of a Strength-Pareto Evolutionary Algorithm. Saidani21 informs

its predictor with the features of Abdalkareem20, and some additional ones. The complete list is as follows: the Current Commit (number of modified subsystems, directories, files, entropy, lines added, lines deleted, day of the week, commit message, and types of files changed), its Purpose (classification as addition, corrective, merge, perfective, preventative, non-functional, or none, whether it is a fix, documentation, build-files-only, meta-files only, a merge, media, only source code, only formatting, only comments, or maintenance activity), and its link to Last Commits (number of recently skipped commits, number of recently skipped commits by the current committer, result of the previous build by the current committer, number of unique last commits in the modified files, time since last modification of the files, number of developers that last touched the files, size of the changed lines before the commit, number of commits made by the developer before this commit, number of commits made by the developer before this commit in this subsystem, and the number of commits made by the developer before this commit weighted by their age).

Jin22 [55]: A build selection approach that applies a Random Forest classifier to predict and skip builds that it predicts to pass. For its features, Jin22 uses Abdalkareem19's CI-Skip Rules and a collection of additional rules (CI-Run Rules) that make CI-Skip Rules safer. The complete list of features checks whether the changes are only source comments, only formatting, only source comments or formatting, only non-source files, only meta-files, only version-release files, not reachable by tests, on build scripts, on configuration files, increasing the number of platforms tested, or subsequent to a build failure.

Training and Testing. We used the dataset described in Section 4.1, which includes 82,427 builds from 100 projects. We used 10-fold cross validation to evaluate machine learning based techniques: HybridCISave, Machalica19, Jin20, and Jin22. Each fold has 10 distinct projects that are randomly assigned. Each build in the testing fold is tested by a classifier trained on the other 90 projects. Abdalkareem20, however, cannot 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 Section 4.1). For Saidani21, we used the pre-trained model that they provide in their replication package [89]. Rule-based techniques (Abdalkareem19, Gligoric15, and Herzig15) 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. In other words, we only update the information connected to the last build (e.g., for Subsequent Failures), 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 (e.g., number of changed source files) for the next build, as past work did [51].

Metrics. We measured three metrics in this evaluation: *Saved_Duration*, *Observed_Build_Failures*, and *Observed_Test_Failures*, as in previous work [1, 2, 51, 53]. We measured each of these metrics across all builds of a project. Then, we report the median value of each metric across all projects.

Saved_Duration is the proportion of skipped build duration among all build time. We measured skipped build duration as the aggregation of the duration of all build steps (e.g., build preparation steps, or tests) that a technique decides to skip, for all builds in a project. We measured all build duration as the aggregation of the duration of all build steps (e.g., build preparation steps, or tests) included in all builds in a project. 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.

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

5.1.2 Results. Figure 2 shows the results of all our studied techniques in terms of *Saved_Duration*, *Observed_Build_Failures*, and *Observed_Test_Failures*. The *Y* axis represents the value of each metric (they are all measured as percentages). Each data point in Figure 2 represents the median value of one technique's performance on one metric across all 100 projects. The *X* axis represents the prediction sensitivity thresholds that we studied for HybridCISave. For the techniques that are not customizable (Abdalkareem19, Abdalkareem20, and Saidani21), we represent their results as flat lines.

Like HybridCISave, Jin20 and Jin22 are also customizable in their prediction sensitivity. However, they all provide different trade-offs between saved cost and observed failures at different prediction thresholds. So, to be able to compare HybridCISave to Jin20 and Jin22, we fit their curves in the *Observed_Build_Failures* metric. For each studied threshold of HybridCISave, we plot the variant of Jin20 and Jin22 that provides the closest (but lower) *Observed_Build_Failures*. Then, we also plot in the other graphs the corresponding ratio of *Saved_Duration* and *Observed_Test_Failures* for that variant at that threshold. This allows us to observe which technique provides the highest cost savings, given similar levels of safety.

To ease comparing HybridCISave with techniques that are not customizable (Abdalkareem19, Abdalkareem20, and Saidani21), we highlight the threshold for which HybridCISave provided the closest ratio of *Observed_Build_Failures* to them, in all graphs for all metrics. We also highlight the most conservative (safest) variant of Jin20 (Jin20_Safe).

As we described in Section 3.1.1, we observed that Saidani21 often took longer to decide whether to skip a build than actually running the build itself (for 52% of the projects in our dataset), which defeats the purpose of skipping the build. In such cases, we make Saidani21 time out and we execute the build, allowing the observation of its result. For that reason, we report the results of Saidani21 as two variants: its variant tested on all projects in our dataset (Saidani21_ALL) and its variant tested on the 48% of projects in which its execution time is lower than the build time (Saidani21_PART). We could not specifically pinpoint why Saidani21 has high execution time, since their replication package only provides its compiled code (but not its source code) [89]. However, long execution times are typical in genetic programming algorithms (on which Saidani21 is based). This is because genetic programming iteratively evaluates a large space of candidate solutions, searching for one that maximizes a fitness function. Thus, genetic programming typically runs its search for a long time and only stops when it reaches a user-defined stopping criterion (typically after it reaches a given number of evaluated solutions). This creates a trade-off: a larger number of evaluated solutions makes it more likely to find one with higher fit, but it also increases search time (and thus execution time).

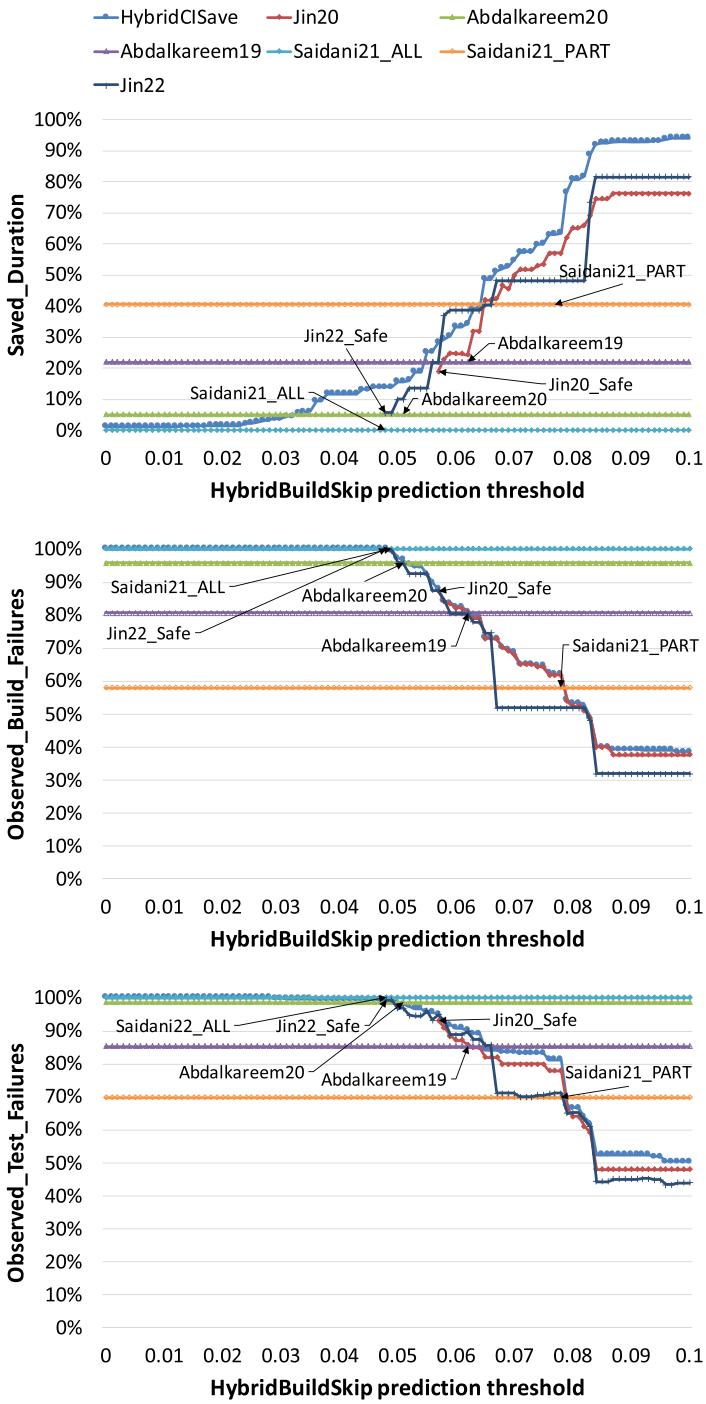

Fig. 2. Cost saved and value kept by HybridCISave and existing build selection techniques.

Our results for RQ1 in Figure 2 show that HybridCISave *provided higher cost savings at the highest safety when compared with previous techniques.*

HybridCISave provided *higher cost savings* than all studied techniques, at similar levels of safety. In the top graph of Figure 2, HybridCISave provided higher values of *Saved_Duration* than any other technique at their highlighted data point (the one at which the compared technique provided similar *Observed_Build_Failures* than HybridCISave). When compared with customizable techniques (Jin20 and Jin22), HybridCISave also provided higher ratios of *Saved_Duration* than them, for all studied thresholds (i.e., ratios of *Observed_Build_Failures*)—with the only exception of a few thresholds for Jin22.

When comparing HybridCISave with the existing technique that observes most build failures (Abdalkareem20), we can observe that HybridCISave achieves higher cost savings (16%) than Abdalkareem20 does (5.1%) while observing the same ratio of failing builds (96%).

The next safest technique is Abdalkareem19. We can observe that HybridCISave is able to save higher cost (34.3%) than Abdalkareem19 (22%) when they observe the same ratio of build failures (81%).

Next, we compare HybridCISave with Saidani21. When executing Saidani21 over all projects in our dataset (Saidani21_ALL), it took longer to decide whether to skip a build than the time to actually execute the build (and thus timed out and we executed the build), for a majority of projects (52%). Therefore, in the median case, Saidani21_ALL had no effect: it produced a median of 0% *Saved_Duration* and of 100% *Observed_Build_Failures*. Even for such high level of safety, HybridCISave provided higher cost savings than Saidani21_ALL: 14% median *Saved_Duration*.

To better understand the effect that Saidani21 could have on projects with low build duration, we also measured its results separately in the remaining 48% of projects (Saidani21_PART). Saidani21_PART obtained a median 40.6% *Saved_Duration* and of 58% *Observed_Build_Failures*. In this case, HybridCISave also provided higher cost savings at similar safety: median 63.5% *Saved_Duration* and of 62% *Observed_Build_Failures*.

We also compared HybridCISave to Jin20, and observed that HybridCISave provided higher *Saved_Duration* given similar ratio of *Observed_Build_Failures*, for all thresholds. We particularly highlight three data points. First, at Jin20's safest threshold (Jin20_Safe), HybridCISave provided higher cost savings: *Saved_Duration* (median 28% vs. median 19%) with the same *Observed_Build_Failures* (median 87%). Second, HybridCISave could achieve even higher safety than Jin20_Safe: it was able to observe median 100% failures (vs. median 87% for Jin20_Safe) and save some cost: 14% median *Saved_Duration*. Third, at Jin20's threshold of lowest safety (median 40% *Observed_Build_Failures*), HybridCISave could achieve much higher *Saved_Duration*: up to median 93%.

When comparing HybridCISave to Jin22, we also observed that HybridCISave provided higher *Saved_Duration* given similar ratio of *Observed_Build_Failures*, for most thresholds (except for 7 thresholds out of 101 studied). Generally, we observed that Jin22 provided higher cost savings than Jin20, given similar ratios of safety, but HybridCISave provided even higher cost savings than Jin22.

Finally, HybridCISave provided *as much safety (100%) as the safest existing technique* while also providing higher cost savings. The top and middle graphs of Figure 2 show that HybridCISave reached 100% *Observed_Build_Failures* (as much as the safest existing techniques Saidani21_ALL and Jin22) while at the same time providing higher cost savings than them (14% *Saved_Duration* for HybridCISave at threshold 0.048, compared to 0% for Saidani21_ALL and 5% for Jin22).

Summary for RQ1: HybridCISave provided higher cost savings at the highest safety when compared with previous techniques.

6 EXPERIMENT 2: ABLATION STUDY: EVALUATING HYBRIDCISAVE'S DESIGN DECISIONS

In the second experiment, our goal is to understand the extent to which each design decision in HybridCISave contributed to providing high cost savings and high safety. We first studied the design decisions skipping partial builds (HybridTestSkip) in addition to full builds (HybridBuildSkip) in RQ2, and the decision of combining test selection techniques with build selection techniques in HybridBuildSkip in RQ3. Then, we studied the decision of using a Random Forest machine learning model in HybridBuildSkip in RQ4. And finally, we studied the decision of combining multiple test selection techniques to achieve higher safety in RQ5.

6.1 RQ2: What Is the Benefit of Having a Test Selection Component in Addition to a Build Selection Component?

6.1.1 Research Method. To answer this research question, we followed the same research method that we described for RQ1 (Section 5.1.1) but with different techniques and similar metrics. We used the same 10-fold cross validation across projects (10 randomly selected projects in each fold) for machine learning based techniques, and we applied rule-based techniques directly (they do not require training).

Studied Techniques. We compared the effectiveness of skipping both full and partial builds (HybridCISave) with the effectiveness of skipping full builds only (HybridBuildSkip).

Metrics. We made evaluations in two dimensions: cost savings 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 HybridCISave in RQ1. Measuring cost savings 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.

6.1.2 Results. We plot our evaluation results in Figure 3. This figure shows the median value for each metric across our studied projects, for multiple prediction thresholds.

In Figure 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-pp higher cost. This shows that the strategy of skipping both full and partial builds is able to increase cost savings 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 savings (1.4%) by skipping partial builds. This saving of cost is produced by its HybridTestSkip component.

Figure 3 also shows that HybridCISave and HybridBuildSkip observed similar ratios of test failures—that is, 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 occurred at thresholds 0 to 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 as HybridBuildSkip.

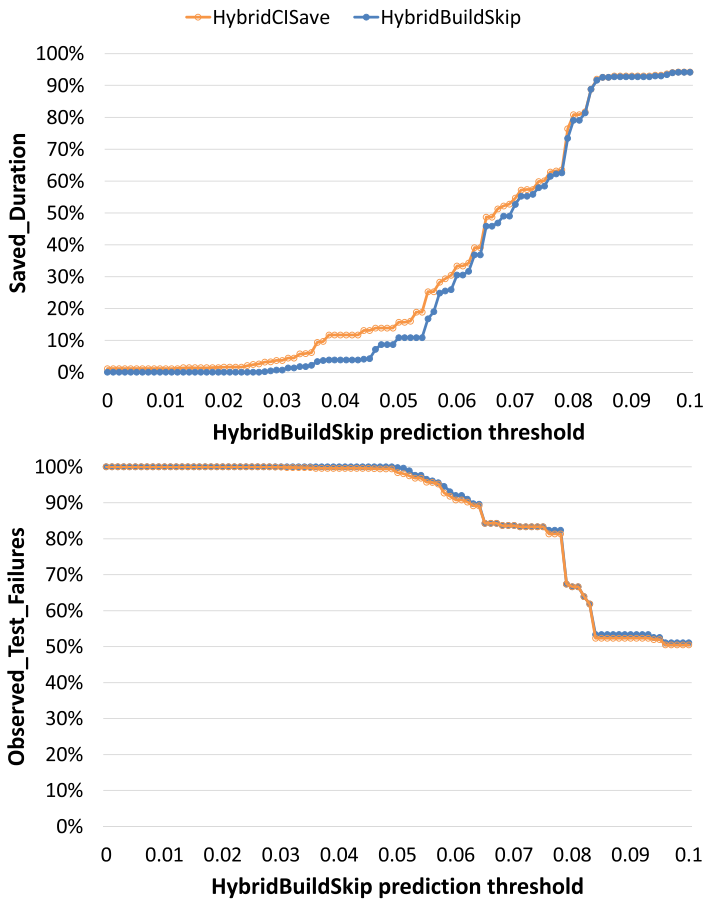

Fig. 3. Cost saved and test failures observed by HybridCISave and HybridBuildSkip.

Summary for RQ2: Having a test selection component in addition to a build selection component increased the cost savings provided by HybridCISave.

6.2 RQ3: What Is the Benefit of Having Test Selection Approaches to Predict Build Outcomes?

6.2.1 Research Method. To answer this research question, we followed the same research method that we described for RQ2 (Section 6.1.1), but with different techniques.

Studied Techniques. We compared the effectiveness of combining both build-selection and test-selection techniques to predict and skip builds (HybridBuildSkip) with the effectiveness of only combining build selection techniques for the same purpose (HybridBuildSkip-Base). HybridBuildSkip-Base is a variant of HybridBuildSkip that only takes as features Abdalkareem19, Abdalkareem20, Jin20_Safe, and Subsequent Failures. To better compare these two techniques, we adjusted HybridBuildSkip-Base to have closest but smaller failure observations so that we can compare their saved duration as we did for Jin20 and Jin22 in RQ1.

6.2.2 Results. We plot in Figure 4 the median value for each metric across our studied projects, for multiple prediction thresholds. Figure 4 shows that by adding test selection approaches to

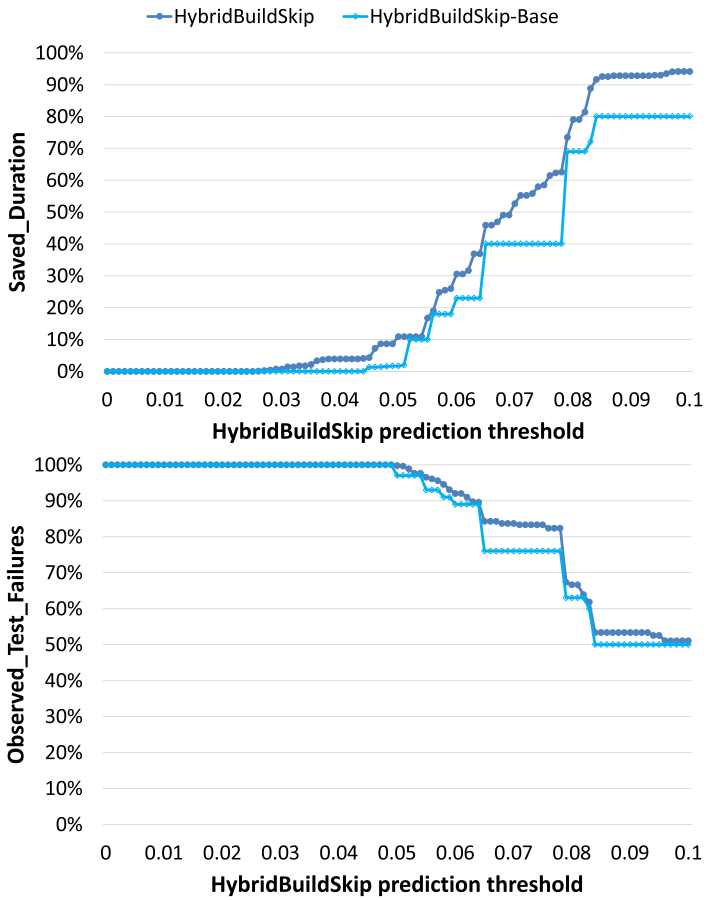

Fig. 4. Cost saved and test failures observed by HybridBuildSkip and HybridBuildSkip-Base.

predict build outcomes, HybridBuildSkip consistently saved more cost and observed more test failures than HybridBuildSkip-Base.

This shows that test selection approaches can be effective for predicting build outcomes. Finally, we want to highlight that although 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).

Summary for RQ3: Having test selection approaches to predict build outcomes in HybridBuildSkip increased both its cost-saving and failure observation ability.

6.3 RQ4: What Is the Benefit of Using a Random Forest Classifier as Opposed to Other Machine Learning Models in HybridBuildSkip?

6.3.1 Research Method. To answer this research question, we followed the same research method that we described for RQ1 (Section 5.1.1), but with different techniques and different metrics. We used the same 10-fold cross validation across projects (10 randomly selected projects in

each fold) for machine learning based techniques, and we applied rule-based techniques directly (since they do not require training).

Studied Techniques. We modified HybridBuildSkip to use different machine learning models, to understand how they would impact its performance. HybridBuildSkip is the only component of HybridCISave that uses a machine learning algorithm. We studied four machine learning algorithms: Random Forest (HybridBuildSkip_RF), AdaBoost (HybridBuildSkip_ABT), Multiple Layer Perceptron (HybridBuildSkip_MLP), and Logistic Regression (HybridBuildSkip_LR). We used the default configuration in scikit-learn for all our studied machine learning algorithms [80]. We also included one baseline technique (HybridBuildSkip_VOTE) that makes its predictions based on the majority vote of all feature techniques in HybridBuildSkip.

Metrics. For other research questions, we measured metrics like *Saved_Duration* and *Observed_Build_Failures*, to understand the impact on cost savings and safety that our studied techniques would produce for developers. For this research question, however, we are interested in studying the traditional metrics of *precision*, *recall*, and *F1-score*, to compare the performance of different machine learning models. We again measured each of these metrics across all builds of a project, and reported the median value of each metric across all projects, for each studied threshold.

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.

6.3.2 Results. Figure 5 shows the median precision, recall, and F1 that each variant of HybridBuildSkip obtained across projects, for each studied prediction threshold.

We make multiple observations in Figure 5. First, VOTE obtained a low F1-score (median 12%) and could correctly predict the majority of build failures (median 90% recall), but with a very low precision (median 8%). In other words, it would skip few build failures but also save very little cost.

Machine learning algorithms achieved higher scores than the HybridBuildSkip_VOTE baseline technique on all metrics, particularly as their prediction thresholds increased. Higher prediction thresholds made techniques more likely to predict builds to pass, which decreased their recall and increased their precision. F1-scores also increased with higher prediction thresholds, since prediction and recall scores became more similar to each other.

Generally, all algorithms followed this trend and performed quite similarly to each other. HybridBuildSkip_MLP and HybridBuildSkip_LR provided quite similar precision, recall, and F1-score to each other, for all thresholds. HybridBuildSkip_RF and HybridBuildSkip_ABT also performed similarly but with a small difference: for a few thresholds (approximately 0.083–0.1), HybridBuildSkip_ABT provided approximately 10-pp higher recall (approximately 50% vs. 40%), and HybridBuildSkip_RF provided approximately 20-pp higher precision (approximately 40% vs. 20%). This means that for those thresholds, HybridBuildSkip_ABT was approximately 10-pp safer but saved approximately 20-pp less cost than HybridBuildSkip_RF. Some practitioners may prefer the trade-off offered by HybridBuildSkip_ABT, and others may prefer the one offered by HybridBuildSkip_RF, for these specific thresholds.

Finally, to be able to compare techniques with a single number, we also calculated the mean F1-score for each variant across thresholds, and we show it in Table 2. This table also shows that RF and ABT performed quite similarly to each other, that MLP and LR performed quite similarly to each other, and that all machine learning models improved over VOTE. We could have selected either of RF or ABT as the best-performing model for the design of our proposed technique

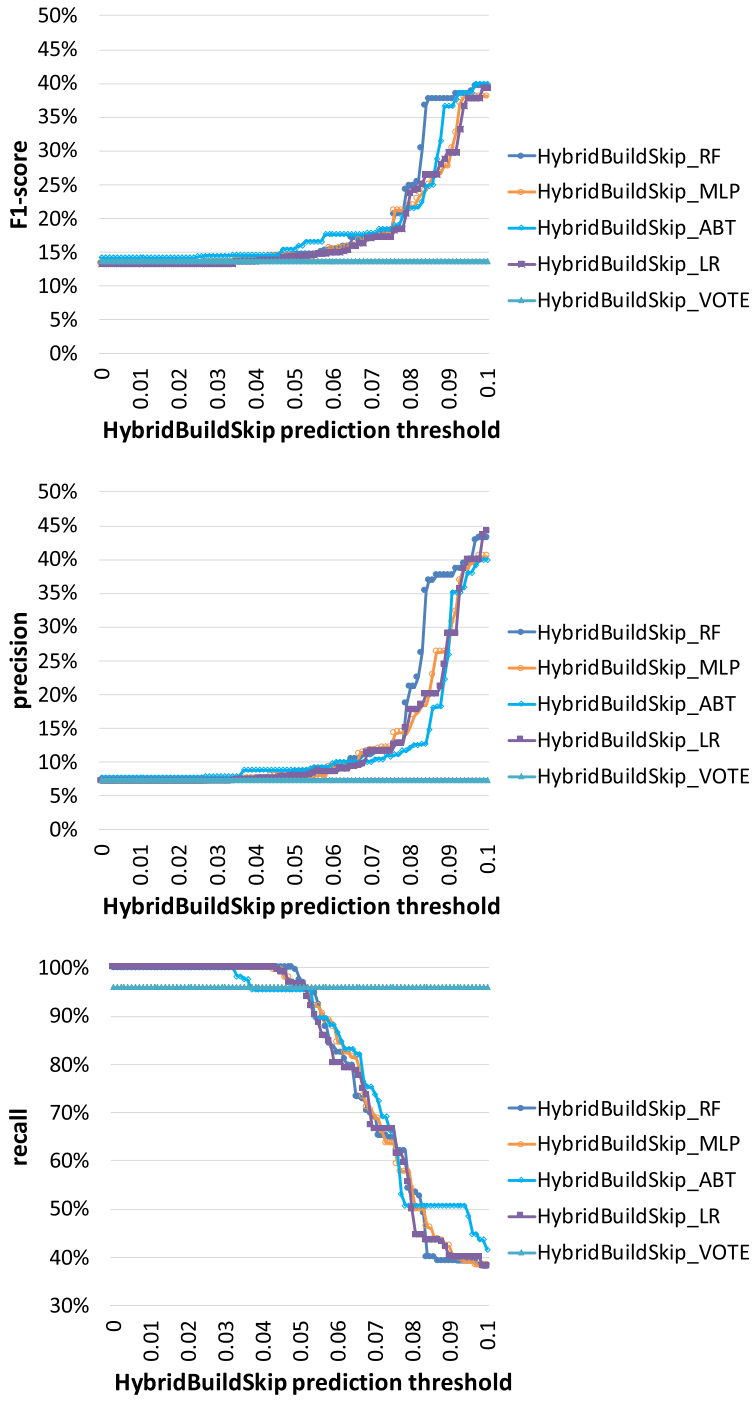

Fig. 5. Cost saved and value kept by HybridBuildSkip under different machine learning algorithms.

Table 2. Mean F1-Score Across Thresholds for Different Variants of HybridBuildSkip Using Different Machine Learning Models

HybridBuildSkip Variant	Mean F1-score
HybridBuildSkip_RF (Random Forest)	0.1918
HybridBuildSkip_ABT (AdaBoost)	0.1915
HybridBuildSkip_MLP (Multiple Layer Perceptron)	0.1794
HybridBuildSkip_LR (Logistic Regression)	0.1788
HybridBuildSkip_VOTE (Majority Vote)	0.1370

HybridBuildSkip, but we ended up selecting RF because it provided slightly higher mean F1-score. Additionally, RF was faster to execute in our experiments.

Summary for RQ4: Random Forest provided the best prediction performance—slightly better than AdaBoost, and much better than a baseline technique based on the majority vote.

6.4 RQ5: How Much Cost-Saving and Failure-Observation Can HybridTestSkip Achieve?

6.4.1 Research Method. To answer this research question, we followed the same research method that we described for RQ1 (Section 5.1.1) but with different techniques and metrics. We used the same 10-fold cross validation across projects (10 randomly selected projects in each fold) for machine learning based techniques, and we applied rule-based techniques directly (they do not require training).

Studied Techniques. We compared HybridTestSkip with its individual components, to study whether it provided higher safety. *HybridTestSkip* is the test-selection component of our proposed approach (see Section 3). Its goal is to predict and skip some tests within the builds that were not selected to be skipped (more details in Section 3.2). *Gligoric15* [39] is a test selection technique that skips tests that do not exercise the changed files in the build (more details in Section 3.1.2). *Machalica19* [1] is a test selection technique that skips tests that it predicts to pass, using an XGBoost classifier (more details in Section 3.1.2). *Herzig15* [51] is a test selection technique that skips tests for which it expects that executing it has higher cost than skipping it (more details in Section 3.2.1).

Metrics. We used 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 the 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. Since HybridTestSkip is a rule-based technique, we do not apply different thresholds for it.

6.4.2 Results. In Figure 6, 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 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

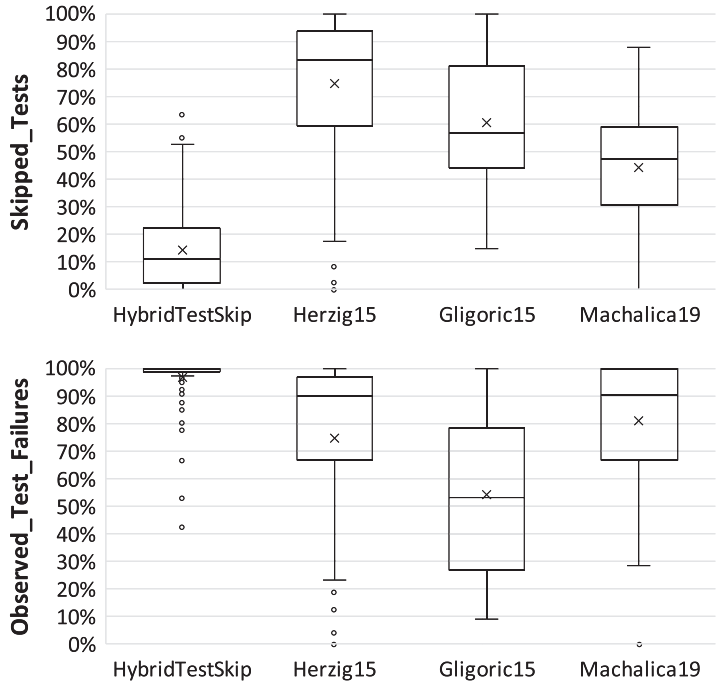

Fig. 6. Cost saved and value kept by HybridTestSkip.

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 savings.

Summary for RQ5: Combining multiple test selection techniques in HybridTestSkip enabled it to achieve very high safety (median 100% Observed_Test_Failures) while also saving some cost (median 11.3% Skipped_Tests).

7 EXPERIMENT 3: EVALUATING THE IMPORTANCE OF HYBRIDCISAVE'S FEATURE TECHNIQUES

In this experiment, we aim to understand the relative importance of each of the feature techniques used in HybridBuildSkip and HybridTestSkip.

7.1 RQ6: What Is the Relative Importance of Each Feature in HybridBuildSkip?

7.1.1 Research Method. We applied the information gain attribute evaluation [5] on all features of HybridBuildSkip for all projects in our dataset. Table 3 shows the median value for each feature and its corresponding information gain across the studied projects, ranked from higher to lower values. We applied the same method to measure the information gain of features in HybridBuildSkip-Base and found that it produced the same ranking.

7.1.2 Results. In Table 3, we observe that Subsequent Failures was the feature with the highest information gain value (0.159). This would signal that failing builds often continued to fail for a

Table 3. Importance of HybridBuildSkip's Features

Feature Name	Information Gain (median value)
F6: Subsequent Failures	0.1590
F4: Gligoric15 [39]	0.0060
F3: Jin20_Safe [51]	0.0040
F2: Abdalkareem20 [1]	0.0038
F1: Abdalkareem19 [2]	0.0034
F5: Machalica19 [67]	0.0028

few more builds, until developers fixed the problem. In other words, it seems that it often took developers a few attempts to fix the bug and turn the build to pass. We investigated our dataset deeper and observed that 40% of build failures happened after another build failure (e.g., commit `SonarSource/sonarqube: 458dcff`), which would support that conjecture. Past work also observed similar high ratios of subsequent failing builds (52%) in other datasets [51], and that the status of the last build is an effective feature to predict build outcomes [15, 42, 75]. Other studies observed that developers using IDEs can also take multiple attempts to fix bugs (e.g., [10, 40]). This observation also shows the benefit for HybridBuildSkip to have included Subsequent Failures as an individual feature, since it obtained much higher information gain than the other features.

All other studied features had a smaller impact on the build outcome prediction. Among them, Gligoric15 had the highest information gain value. We also investigated deeper to understand its effect. We observed that in our dataset 40% of builds had no test to cover the changes (e.g., commit `jOOQ/jOOQ: e66bac6`). Additionally, more than 90% of builds with no test to cover the changes are passing builds. This means that when Gligoric15 predicted a build to pass because it did not find tests to cover the changes, it was an effective signal that the build would pass. This also 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. Gligoric15 was the second most strongly predictive feature in HybridBuildSkip, and it was based on a test selection technique.

Jin20_Safe, Abdalkareem19, and Abdalkareem20 had similar information gain values to each other, signaling that they were useful predictors, but not as strong as Subsequent Failures or Gligoric15.

Finally, Machalica19 had the lowest information gain value. We investigated deeper and observed that it rarely predicted all tests in a build to pass (less than 15% of the time), so it rarely provided a discriminatory signal for HybridBuildSkip.

Summary for RQ6: Subsequent Failures was the most effective feature of HybridBuildSkip. After that, Gligoric15 [39] had the highest importance.

7.2 RQ7: What Is the Relative Importance of Each Feature in HybridTestSkip?

7.2.1 Research Method. We applied the FOIL information gain attribute evaluation on each rule of HybridTestSkip for all projects in our dataset. FOIL information gain [30] 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 4 shows the median value for each feature and its corresponding FOIL information gain across projects.

7.2.2 Results. In Table 4, we observe that Gligoric15 was the feature with the highest FOIL information gain: 2.08. In fact, Gligoric15 was also the feature with second highest information gain for HybridBuildSkip (after Subsequent Failures). This confirms that the strategy chosen by

Table 4. Importance of HybridTestSkip's Features

Feature Name	Information Gain (median value)
Herzig15 [44]	0.02
Gligoric15 [39]	2.08
Machalica19 [51]	1.63

Gligoric15 (skipping tests that cannot reach the changed files) was a strong predictor of whether tests will pass or fail. We observed in RQ6 that Gligoric15 was an important feature to predict whether all tests in a build will pass. Therefore, it is normal that it would also be a strong predictor of whether individual tests will pass.

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 was also a valuable predictor for test outcome. This indicates that tests that failed often in the past are more likely to continue failing. For example, when the test named `org.semanticweb.owlapi.api.test.anonymous.AnonymousTurtleAssertionTestCase` for project `owlcs/owlapi` failed in build #252, it had historically failed in six other previous builds: #240 (`d69e419c327fbde640b772a3ef7b385113863bef`), #243, #246, #247, #248, and #249.

Finally, Herzig15 had a much lower information gain than the other two features: 0.02. We investigated deeper, and we observed that Herzig15 has a tendency to skip tests that have very long executions. Its formula estimates that tests are worth skipping when the cost of skipping them, probably not observing their failure, and finding and fixing the problem later is lower than the cost of executing them. Therefore, the cost of executing a test has to be very high for Herzig15 to decide that it is worth skipping. As a result of this property, Herzig15 very rarely decides that a test should be skipped. Since Herzig15 most often provides the same prediction, it is rarely a good predictor.

Summary for RQ7: Among the feature techniques of HybridTestSkip, Gligoric15 [39] was the most effective, followed by Machalica19 [67], and Herzig15 [44] was least important.

8 EXPERIMENT 4: MEASURING HYBRIDCISAVE'S EXECUTION 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 of RAM, Ubuntu 16.04.3 LTS, and Python 3.5.2.

8.1 RQ8: What Is the Total Execution Time of HybridCISave and Its Individual Components?

8.1.1 Research Method. In this experiment, we measured the execution time of HybridCISave and all of he techniques it uses, including Gligoric15, Jin20_Safe, Abdalkareem20, Abdalkareem19, Machalica19, and Herzig15, in all builds of our dataset. We computed their runtime in seconds across studied projects. For those techniques with a machine learning classifier, we only included its prediction time—that is, we did not 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.

8.1.2 Result. We report in Table 5 the results of this experiment as the median value of median and 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

Table 5. 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
Abdalkareem19	0.000001	0.000001
Abdalkareem20	0.001600	0.031600
Jin20_Safe	0.001500	0.002200
Build Duration	441.500000	2,151.000000

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. Among them, Abdalkareem20 takes highest time (median 0.0016 seconds per build) because its predictor is triggered for every commit in one build, which means that 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 runtimes of our studied techniques are negligible compared to the build times.

Summary for RQ8: The time required to execute HybridCISave was negligible when compared to the time required to run a build.

8.2 RQ9: How Much Cost Does HybridCISave Save If We Account for Its Execution Time?

8.2.1 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*

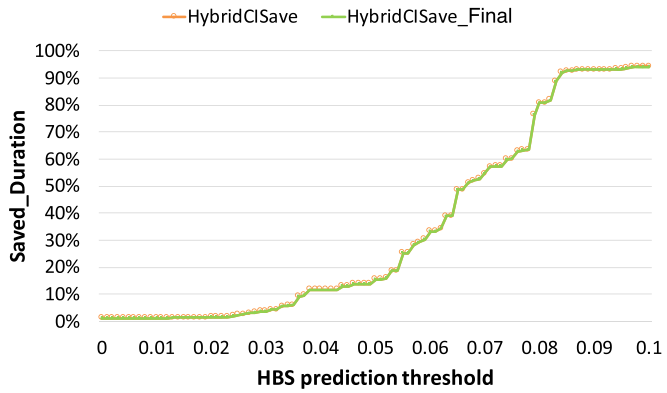

Fig. 7. Cost saved by HybridCISave with or without considering its execution time.

metric, before (HybridCISave) and after (HybridCISave_Real) we deduct HybridCISave's execution time from it.

8.2.2 Result. Figure 7 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 that the execution time of HybridCISave has a negligible impact on its achieved cost reduction. All of the differences are smaller than 0.02 pp. The biggest difference is 0.016 pp: 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 pp: 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 savings while also achieving a high ratio of observed failures (threshold 0.047), the difference between HybridCISave_Real and HybridCISave is only 0.01 pp. As a result, we conclude that the execution time of HybridCISave has a negligible impact on its ability to save cost.

Summary for RQ9: The execution time of HybridCISave had a negligible impact on its ability to save cost.

9 DISCUSSION

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

Combining Approaches in Different Ways. Prior work combined multiple approaches in many different ways for better performance. For example, Zhang [124] 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 HybridCISave. We also found that Subsequent Failures is the most effective feature, which also confirms the findings from previous studies [42, 51, 75]. 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 savings and failure observation. We selected the Random Forest classifier for our approach because it was the fastest when making its predictions.

Predicting Build Outcomes Through Test Selection Approaches. Prior work [124] 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.

Predicting Build Failures in Build Preparation Steps. Test selection techniques predict whether a test case (e.g., [67]) or the whole test suite (e.g., [77]) will fail (and execute it) or pass (and skip it). Build selection techniques predict whether a build will fail (and execute it) or pass (and skip it). Builds can fail when one or more of their tests fail, but also when their build preparation steps fail (e.g., when setting up the execution environment, even before any test was executed [43, 55]). This kind of build failures can only be predicted by build selection techniques, since test selection techniques only predict the outcome of tests.

However, few build selection techniques explicitly use features that try to capture failures caused by reasons other than faults in the source code. For example, Abdalkareem19 [2] in fact predicts that builds that only contain changes to non-source files will pass. To the extent of our knowledge, only Jin22 [55] considers features that explicitly try to capture failures in build preparation steps: by considering changes in build scripts (e.g., "`pom.xml`") and configuration files (e.g., "`travis.yml`") as possibly causing failures. Since our approach HybridCISave was designed before Jin22 was published, it does not use it in its predictor. Thus, HybridCISave also does not explicitly use features to predict failures in build preparation steps (even if it does predict some of them correctly, thanks to its other features). Future improvements to HybridCISave could consider adding more features to predict failures in build preparation steps to improve its safety even further.

Build Failures That Are Hard to Be Detected by Existing Techniques. We further explored the failure reason of failing builds that can only be detected by one feature of HybridCISave (all 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 file 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 a Travis configuration file ("`travis.yml`"). This further validates our previous discussion point: that the configuration file of Travis CI can cause failing builds, and developers may struggle to write the configurations correctly [111].

Build Selection Approach 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 Savings 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 article are designed as customizable, to cater to

different preferences. For aggressive 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, then also increase the ratio of observed build failures.

Different Dimensions to Evaluate Build Selection Approaches. Both previous work and this work [1, 51, 53] evaluate build selection approaches in two dimensions: cost savings 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 savings 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.

Different Performance of HybridCISave on Different Projects. Our dataset contains 100 software projects in two programming languages: 82 projects in Java and 18 projects in Ruby. It is possible that our proposed technique would provide different results for different programming languages. To study this possibility, we ran an additional experiment. We separately evaluated HybridCISave's performance on *Saved_Duration* and *Observed_Build_Failures* on Java and Ruby projects at the threshold of 0.048 (the highest threshold for which HybridCISave provides 100% median *Observed_Build_Failures*). We plot the results in Figure 8.

Figure 8 shows that HybridCISave provided very similar median values of *Saved_Duration* and *Observed_Build_Failures* for both Ruby and Java projects. The main difference was that the results for Ruby projects had higher variance (we can see that the interquartile range was wider and the whiskers were farther apart for Ruby projects).

However, this observed wider variance in Ruby could be the result of many different factors, so we cannot necessarily attribute it to the different programming language. For example, it could be because our studied Ruby projects are more diverse than our studied Java projects, or simply because we studied a limited number of Ruby projects (18). A future study could mine software projects, clustering those with similar characteristics (e.g., programming languages, team sizes, or age), to study the different behavior of build and test selection techniques across clusters.

Finally, we find it very encouraging that HybridCISave provided quite similar median scores for our metrics for both Ruby and Java. That makes us believe that our results would generalize to other datasets and programming languages.

10 THREATS TO VALIDITY

10.1 Construct Validity

We use metrics as proxies to represent the *value* (observation of failures) and *cost* (build duration) in CI. These are metrics that developers have reported as describing the value and cost of CI (e.g., [19, 29, 47, 75]) and are metrics that other existing approaches for saving cost in CI have used (e.g., [2, 67]).

Another threat to construct validity concerns whether developers would want to use HybridCISave in their environments. Multiple factors make us believe that developers would find HybridCISave quite useful in their software development activities.

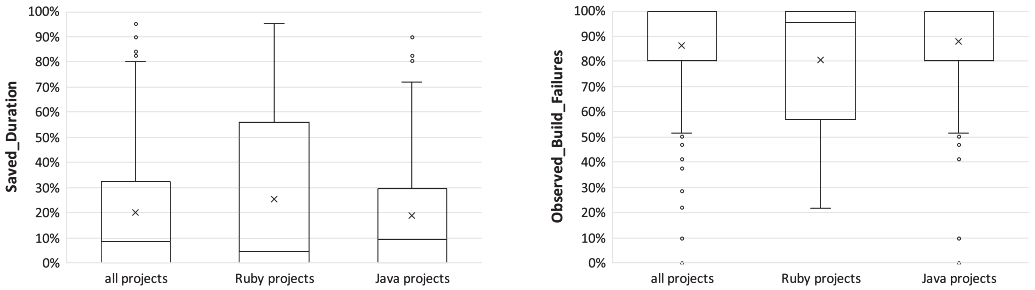

Fig. 8. Cost saved and value kept by HybridCISave for different programming languages.

First, similar techniques to skip the execution of tests are already developed and being used at large software development companies, such as at Microsoft (Herzig15 [44]) and Facebook (Machalica19 [67]).

Second, previous studies directly asked developers about their impressions regarding build selection. Abdalkareem et al. [2] surveyed 40 professional developers and found that 75% of them considered the ability to automatically determine if a build should be skipped to be nice, important, or very important. They also observed that developers already skip commits manually, according to their personal criteria, "to save time" and "to not waste computing resources."

Third, past studies also studied developers actually using a build selection technique. Saidani et al. [88] observed 14 professional developers using an automated build selection technique and found that developers accepted around 90% of the recommendations of the tool. Their studied developers found the technique's recommendations valuable (e.g., "It's bad for this commit to wastefully use server time"). Finally, like Abdalkareem19, they also observed that their studied developers already had their own custom mechanisms in place to sometimes skip builds (e.g., "I found the bot very useful to me, basically what I used to do is to skip the entire build pipeline if certain condition is met based on my 'safe list'").

10.2 Internal Validity

To guard internal validity, we carefully tested our evaluation tools on subsets of our dataset while developing them.

Our analysis could also be influenced by incorrect information in our analyzed dataset. For this, we selected a popular dataset that has been analyzed in other studies. Existing work [33] reports noise data points on this dataset; however, many of the techniques to save cost in CI [2, 42, 51, 64] were originally evaluated on TravisTorrent projects. Additionally, we extensively curated TravisTorrent, removing toy projects following standard practice [48, 75], unusable projects for test-granularity techniques, and canceled builds as in past work [33, 49, 83].

We also took into consideration the advice in the studies by Gallaba et al. [33] and by Ghaleb et al. [38] to account for potential noise in the TravisTorrent dataset. We did so in the following ways. First, sometimes developers manually flag failures to be ignored when they cannot officially support them and thus should not represent the status of the build [33]. We considered passing builds with flagged ignored failures as passing. Second, sometimes builds have both passing and failing jobs [33]. We considered failing builds with passing jobs as failing builds. If at least one job fails, it signals a problem, informing developers. Third, sometimes developers exclude jobs from a build after they fail in a previous build [38]. We follow the same logic in this case: we consider the build as failing when at least one of its jobs fails, and passing when the failing jobs are excluded, and therefore all jobs pass. Fourth, sometimes builds fail after another build failure [33, 38]. We

considered builds that fail after another failure as correctly labeled, because they flag an unsolved problem, being informative for developers. Fifth, sometimes builds fail because of environmental issues, such as internal errors of the CI platform, or running out of space in memory or disk [38]. We consider such failing builds as failing, since they inform developers of a problem that needs to be resolved.

Some of our selected projects may not be resource constrained; however, we believe that the evaluation results should not be influenced, and our technique still benefits those projects that have less resource problems.

Our results may also be affected by flaky tests causing spurious failing builds. Flaky tests are tests that produce non-deterministic behavior (i.e., that fail inconsistently) without changes to the code under test [9, 20, 61, 79]. Common causes for flaky tests are concurrency issues, network connectivity problems, or test order dependencies [79]. Flaky tests are common in software projects, and CI is normally applied in the presence of flaky tests, since companies do not consider it economically viable to remove them (e.g., [67, 71]). Furthermore, previous build selection techniques [1, 2, 51, 55, 88] kept flaky tests in their datasets in their evaluations. Some previous work removed flaky tests from their datasets in their evaluation of a test selection technique (e.g., Machalica et al. [67] and Pan et al. [77]) by removing those that produced a different outcome when re-running them without changes to the code. After this process, Pan and Pradel [77] found that flaky tests only affected a minority of their dataset (0.067% of their test suite runs), and other studies found that flaky tests were a minority of the wider population of tests [44]. For these reasons, we believe that flaky tests had a limited impact in our experiments.

Another possible threat is the fact that our experiments did not keep the chronological order of builds across projects. We trained HybridCISave as a cross-project predictor, using builds from other projects that were not in the test set, including some builds that chronologically happened after the tested builds (i.e., future builds). Violating the chronological order of builds can lead to overfitting when it is done within a project. If a predictor's training set contains future builds of the same project, it could artificially discover trends that it could not have discovered in a realistic scenario (i.e., in which the future builds are not yet available for training). However, we believe that including future builds in HybridCISave's training from other projects that were not in the test set in our experiments has little risk of overfitting, since any future trends discovered in the training projects may or may not be predictive of future trends that the tested projects will follow.

HybridTestSkip is a rule-based technique that does not require any training data. We also increase our internal validity by following the existing techniques' instructions to replicate their techniques, including using the same machine learning algorithm. We only consider prediction time as the execution time, and we do not include training time, but we believe that in the real-world scenario, the training process is usually completed during a less loaded period.

We did not include the training time of HybridCISave since we believe that the training can happen offline and may not add to the cost of computation. However, we calculated the training time (4.584 seconds) for all components in one single training of HybridCISave as a reference for developers to decide their own preferable frequency for training. Our decision of replicating Gligoric15 to consider only static dependencies may have slightly limited HybridCISave's results. Applying tools other than Understand to generate the dependency graph may make HybridCISave save even more cost. Using only static dependencies allows us to provide a rule that is simple to put into practice, and that could be more easily applicable to more programming languages (since the mechanisms to create dynamic dependencies are more diverse across programming languages). Although including dynamic dependencies may have made it safer, it would also make it harder to implement and to potentially require a much higher computational cost to apply it.

The accuracy of the tools that we used to run static analysis (SciTools Understand [92] and Rubrowser [26]) could also have impacted the ability of HybridCISave. We used these tools to capture static dependencies between source code classes. SciTools Understand captured the following dependencies among classes: "calls", "implements", "includes/imports", "inherits", "inits", "modifies", "overrides", "sets", "throws", and "uses". Rubrowser captured all modules and classes definitions, and all constants that are listed inside a module/class and linked them together. Then, we used the resulting graph of classes and dependencies among them from these tools to determine if the changed files were reachable by the project's tests. A possible threat to validity is that bugs in the source code of these tools may make them miss some dependencies (false negatives), or capture some dependencies that do not really exist (false positives), which could affect the accuracy of Gligoric15 as a feature of HybridCISave. Additionally, since both SciTools Understand [92] and Rubrowser [26] capture static dependencies, they both can have false positives (capturing dependencies that will not be executed in runtime). Such false positives would cause HybridCISave to save less cost than it could if we had used dynamic analysis tools.

10.3 External Validity

To increase external validity, we selected the popular dataset TravisTorrent based on Travis CI, which has been analyzed by many other research works. The projects we chose were all Java or Ruby projects (18% of projects are Ruby projects), because there are no projects with other programming languages in the dataset. Although these two programming languages are popular, different CI habits in other languages may provide slightly different results than the ones in this study. Finally, our cost-saving technique may not be perfectly suitable for software projects that cannot afford a single delay in observing failing builds. However, we believe that this happens rarely according to the existing study [2].

Another threat to external validity questions whether the thresholds will generalize to other projects. We do not present these thresholds as absolute values to be reused across software projects. These values may not generalize to other projects. We simply highlight them to show the trade-off between cost savings and failure observation in response to developers' preferences. Our advice for developers using HybridCISave in their software project is to empirically customize its prediction thresholds to their preference, for their software project.

11 CONCLUSION AND FUTURE WORK

In this article, we studied the benefit that could be obtained by combining multiple build and test selection techniques, and whether such combination introduces high prediction times that could threaten the achieved cost savings. For this goal, we first designed HybridBuildSkip, the first build selection technique that leverages multiple existing approaches as features. We also designed another novel technique to achieve cost savings in CI by skipping both full and partial builds (HybridCISave), which outperformed HybridBuildSkip in our experiments in terms of cost savings while only negligibly reducing its ratio of observed failures. Finally, we studied the execution time of HybridCISave to better understand whether its execution time reduces its total saved duration, finding that it had a negligible impact. In the future, we will explore extending HybridCISave's algorithm with other machine learning algorithms (or combinations of them) and additional features to anticipate bugs (e.g., analyzing code-change history [94–96, 98, 99]), alternative testing activity (e.g., [36, 57, 58]), decision-making metadata (e.g., [3, 4, 72]), developer expertise (e.g., [16, 97]), or cross-language issues (e.g., [17]), or by combining it with other strategies (e.g., also skipping unnecessary dependencies [14] or build preparation steps [31, 35]), to try to achieve even better trade-offs of cost savings and failure observation.

12 REPLICATION

We include a replication package for our article [54].

REFERENCES

[1] Rabe Abdalkareem, Suhaib Mujahid, and Emad Shihab. 2021. A machine learning approach to improve the detection of CI skip commits. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2740–2754.

[2] Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Juergen Rilling. 2021. Which commits can be CI skipped? *IEEE Transactions on Software Engineering* 47, 3 (2021), 448–463.

[3] Khadijah Al Safwan, Mohammed Elarnaoty, and Francisco Servant. 2022. Developers' need for the rationale of code commits: An in-breadth and in-depth study. *Journal of Systems and Software* 189 (2022), 111320.

[4] Khadijah Al Safwan and Francisco Servant. 2019. Decomposing the rationale of code commits: The software developer's perspective. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[5] B. Azhagusundari and Antony Selvadoss Thanamani. 2013. Feature selection based on information gain. *International Journal of Innovative Technology and Exploring Engineering* 2, 2 (2013), 1–4.

[6] Amine Barrak, Ellis E. Eghan, Bram Adams, and Foutse Khomh. 2021. Why do builds fail? A conceptual replication study. *Journal of Systems and Software* 177 (2021), 110939.

[7] Amir Hossein Bavand and Peter C. Rigby. 2021. Mining historical test failures to dynamically batch tests to save CI resources. In *Proceedings of the International Conference on Software Maintenance and Evolution*.

[8] Mohammad Javad Beheshtian, Amir Bavand, and Peter Rigby. 2022. Software batch testing to save build test resources and to reduce feedback time. *IEEE Transactions on Software Engineering* 48, 8 (2022), 2784–2801.

[9] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *Proceedings of the International Conference on Software Engineering*.

[10] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. 2019. Developer testing in the IDE: Patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering* 45, 3 (2019), 261–284.

[11] 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 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR'17)*. https://doi.org/10.6084/m9.figshare.19314170.v1

[12] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. In *Proceedings of the International Conference on Mining Software Repositories*.

[13] Marcelo Cataldo and James D. Herbsleb. 2011. Factors leading to integration failures in global feature-oriented development: An empirical analysis. In *Proceedings of the International Conference on Software Engineering*.

[14] Ahmet Celik, Alex Knaust, Aleksandar Milicevic, and Milos Gligoric. 2016. Build system with lazy retrieval for Java projects. In *Proceedings of the International Symposium on Foundations of Software Engineering*.

[15] 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 *Proceedings of the International Conference on Automated Software Engineering*.

[16] Lykes Claytor and Francisco Servant. 2018. Understanding and leveraging developer inexpertise. In *Proceedings of the International Conference on Software Engineering: Companion Proceedings*.

[17] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2019. Why aren't regular expressions a Lingua Franca? An empirical study on the re-use and portability of regular expressions. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[18] Adam Debbiche, Mikael Dienér, and Richard Berntsson Svensson. 2014. Challenges when adopting continuous integration: A case study. In *Proceedings of the International Conference on Product-Focused Software Process Improvement*.

[19] Paul M. Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education.

[20] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding flaky tests: The developer's perspective. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[21] Omar Elazhary, Colin Werner, Ze Shi Li, Derek Lowlind, Neil A. Ernst, and Margaret-Anne Storey. 2022. Uncovering the benefits and challenges of continuous integration practices. *IEEE Transactions on Software Engineering* 48, 7 (2022), 2570–2583.

[22] 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.

[23] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the International Symposium on Foundations of Software Engineering*.

[24] 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 International Symposium on Software Testing and Analysis*.

[25] Daniel Elsner, Roland Wuersching, Markus Schnappinger, Alexander Pretschner, Maria Graber, René Dammer, and Silke Reimer. 2022. Build system aware multi-language regression test selection in continuous integration. In *Proceedings of the International Conference in Software Engineering: Software Engineering in Practice Track*.

[26] Emad Elsaid. 2019. Rubrowser (Ruby Browser). Retrieved January 21, 2022 from https://github.com/emad-elsaid/rubrowser.

[27] Wagner Felidré, Leonardo Furtado, Daniel Alencar Da Costa, Bruno Cartaxo, and Gustavo Pinto. 2019. Continuous integration theater. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*.

[28] 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.

[29] Martin Fowler and Matthew Foemmel. 2006. Continuous integration. *Thought-Works*. Retrieved December 16, 2022 from http://www.thoughtworks.com/ContinuousIntegration.pdf\/.

[30] Johannes Fürnkranz and Peter A. Flach. 2003. An analysis of rule evaluation metrics. In *Proceedings of the International Conference on Machine Learning*.

[31] Keheliya Gallaba, Yves Junqueira, John Ewart, and Shane Mcintosh. 2022. Accelerating continuous integration by caching environments and inferring dependencies. *IEEE Transactions on Software Engineering* 48, 6 (2022) 2040–2052.

[32] Keheliya Gallaba, Maxime Lamothe, and Shane McIntosh. 2022. Lessons from eight years of operational data from a continuous integration service. In *Proceedings of the International Conference on Software Engineering*.

[33] 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 International Conference on Automated Software Engineering*.

[34] Keheliya Gallaba and Shane McIntosh. 2020. Use and misuse of continuous integration features: An empirical study of projects that (mis)use Travis CI. *IEEE Transactions on Software Engineering* 46, 1 (2020), 33–50.

[35] Alessio Gambi, Zabolotnyi Rostyslav, and Schahram Dustdar. 2015. Improving cloud-based continuous integration environments. In *Proceedings of the International Conference on Software Engineering*.

[36] Aakash Gautam, Saket Vishwasrao, and Francisco Servant. 2017. An empirical study of activity, popularity, size, testing, and stability in continuous integration. In *Proceedings of the International Conference on Mining Software Repositories*.

[37] 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* 24 (2019), 2102–2139.

[38] Taher Ahmed Ghaleb, Daniel Alencar da Costa, Ying Zou, and Ahmed E. Hassan. 2021. Studying the impact of noises in build breakage data. *IEEE Transactions on Software Engineering* 47, 9 (2021), 1998–2011.

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

[40] Milos Gligoric, Stas Negara, Owolabi Legunsen, and Darko Marinov. 2014. An empirical evaluation and comparison of manual and automated test selection. In *Proceedings of the International Conference on Automated Software Engineering*.

[41] Ahmed E. Hassan and Ken Zhang. 2006. Using decision trees to predict the certification result of a build. In *Proceedings of the International Conference on Automated Software Engineering*.

[42] Foyzul Hassan and Xiaoyin Wang. 2017. Change-aware build prediction model for stall avoidance in continuous integration. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*.

[43] Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: An automatic approach to history-driven repair of build scripts. In *Proceedings of the International Conference on Software Engineering*.

[44] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The art of testing less without sacrificing quality. In *Proceedings of the International Conference on Software Engineering*.

[45] Kim Herzig and Nachiappan Nagappan. 2015. Empirically detecting false test alarms using association rules. In *Proceedings of the International Conference on Software Engineering*.

[46] 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 Joint Meeting on Foundations of Software Engineering*.

[47] 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 International Conference on Automated Software Engineering*.

[48] Md. Rakibul Islam and Minhaz F. Zibran. 2017. Insights into continuous integration build failures. In *Proceedings of the International Conference on Mining Software Repositories*.

[49] 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*. Advances in Intelligent Systems and Computing, Vol. 714. Springer, 17–27.

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

[51] Xianhao Jin and Francisco Servant. 2020. A cost-efficient approach to building in continuous integration. In *Proceedings of the International Conference on Software Engineering*.

[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 *Proceedings of the International Conference on Software Engineering: Companion Proceedings*.

[53] Xianhao Jin and Francisco Servant. 2021. What helped, and what did not? An evaluation of the strategies to improve continuous integration. In *Proceedings of the International Conference on Software Engineering*.

[54] Xianhao Jin and Francisco Servant. 2022. Hybrid Build and Test Selection Strategies in Continuous Integration. Retrieved December 16, 2022 from https://zenodo.org/record/4716877.

[55] 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* 188 (2022), 111292.

[56] Eero Kauhanen, Jukka K. Nurminen, Tommi Mikkonen, and Matvei Pashkovskiy. 2021. Regression test selection tool for python in continuous integration process. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*.

[57] Ayaan M. Kazerouni, James C. Davis, Arinjoy Basak, Clifford A. Shaffer, Francisco Servant, and Stephen H. Edwards. 2021. Fast and accurate incremental feedback for students' software tests using selective mutation analysis. *Journal of Systems and Software* 175 (2021), 110905.

[58] Ayaan M. Kazerouni, Clifford A. Shaffer, Stephen H. Edwards, and Francisco Servant. 2019. Assessing incremental testing practices and their impact on project outcomes. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE'19)*.

[59] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. 2014. Why do automated builds break? An empirical study. In *Proceedings of the International Conference on Software Maintenance and Evolution*.

[60] 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.

[61] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A study on the lifecycle of flaky tests. In *Proceedings of the International Conference on Software Engineering*.

[62] 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.

[63] Jingjing Liang. 2018. *Cost-Effective Techniques for Continuous Integration Testing*. Master's thesis. University of Nebraska.

[64] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. 2018. Redefining prioritization: Continuous prioritization for continuous integration. In *Proceedings of the International Conference on Software Engineering*.

[65] Qi Luo, Kevin Moran, Denys Poshyvanyk, and Massimiliano Di Penta. 2018. Assessing test case prioritization on real faults and mutants. In *Proceedings of the International Conference on Software Maintenance and Evolution*.

[66] Yang Luo, Yangyang Zhao, Wanwangying Ma, and Lin Chen. 2017. What are the factors impacting build breakage? In *Proceedings of the 2017 14th Web Information Systems and Applications Conference (WISA'17)*. IEEE, Los Alamitos, CA, 139–142.

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

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

[69] Ricardo Martins, Rui Abreu, Manuel Lopes, and João Nadkarni. 2021. Supervised learning for test suit selection in continuous integration. In *Proceedings of the International Conference on Software Testing, Verification, and Validation Workshops*.

[70] 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 International Conference on Software Engineering: Software Engineering in Practice Track*.
[71] John Micco. 2017. The State of Continuous Integration Testing @Google. Retrieved December 16, 2022 from https://research.google/pubs/pub45880/.
[72] Louis G. Michael, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. 2019. Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In *Proceedings of the International Conference on Automated Software Engineering*.
[73] Ade Miller. 2008. A hundred days of continuous integration. In *Proceedings of the Agile Conference*.
[74] Shaikh Mostafa, Xiaoyin Wang, and Tao Xie. 2017. PerfRanker: Prioritization of performance regression tests for collection-intensive software. In *Proceedings of the International Symposium on Software Testing and Analysis*.
[75] Ansong Ni and Ming Li. 2017. Cost-effective build outcome prediction using cascaded classifiers. In *Proceedings of the International Conference on Mining Software Repositories*.
[76] Klérisson V. R. 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 International Conference on Mining Software Repositories*.
[77] Cong Pan and Michael Pradel. 2021. Continuous test suite failure prediction. In *Proceedings of the International Symposium on Software Testing and Analysis*.
[78] Rongqi Pan, Mojtaba Bagherzadeh, Taher A. Ghaleb, and Lionel Briand. 2022. Test case selection and prioritization using machine learning: A systematic literature review. *Empirical Software Engineering* 27 (2022), 29.
[79] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2022. A survey of flaky tests. *Transactions on Software Engineering Methodology* 31, 1 (2022), Article 17, 74 pages.
[80] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12, 85 (2011), 2825–2830.
[81] 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 International Workshop on Cooperative and Human Aspects of Software Engineering*.
[82] 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 International Conference on Mining Software Repositories*.
[83] 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 International Conference on Mining Software Repositories*.
[84] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering* 22, 8 (1996), 529–551.
[85] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology* 6, 2 (1997), 173–210.
[86] 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.
[87] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. 2022. Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engineering* 29 (2022), 21.
[88] Islem Saidani, Ali Ouni, and Wiem Mkaouer. 2022. Detecting skipped commits in continuous integration using multi-objective evolutionary search. *IEEE Transactions on Software Engineering* 48, 12 (2022), 4873–4891.
[89] Islem Saidani, Ali Ouni, and Wiem Mkaouer. 2021. Detecting Skipped Commits in Continuous Integration Using Multi-Objective Evolutionary Search. Replication Package. *GitHub*. Retrieved December 16, 2022 from https://github.com/stilab-ets/SkipCI.
[90] Mark Santolucito, Jialu Zhang, Ennan Zhai, Jürgen Cito, and Ruzica Piskac. 2022. Learning CI configuration correctness for early build feedback. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*.
[91] Adrian Schroter. 2010. Predicting build outcome with developer interaction in Jazz. In *Proceedings of the International Conference on Software Engineering*.
[92] SciTools Understand. 2020. Understand: A Powerful Static Code Analysis Tool. Retrieved March 2, 2020 from https://scitools.com/.
[93] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' build errors: A case study (at Google). In *Proceedings of the International Conference on Software Engineering*.

[94] Francisco Servant. 2013. Supporting bug investigation using history analysis. In *Proceedings of the International Conference on Automated Software Engineering*.
[95] Francisco Servant and James A. Jones. 2011. History slicing. In *Proceedings of the International Conference on Automated Software Engineering*. IEEE, Los Alamitos, CA.
[96] Francisco Servant and James A. Jones. 2012. History slicing: Assisting code-evolution tasks. In *Proceedings of the International Symposium on the Foundations of Software Engineering*.
[97] Francisco Servant and James A. Jones. 2012. WhoseFault: Automatic developer-to-fault assignment through fault localization. In *Proceedings of the International Conference on Software Engineering*.
[98] Francisco Servant and James A. Jones. 2013. Chronos: Visualizing slices of source-code history. In *Proceedings of the Working Conference on Software Visualization*.
[99] Francisco Servant and James A. Jones. 2017. Fuzzy fine-grained code-history analysis. In *Proceedings of the International Conference on Software Engineering*.
[100] 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 International Conference on Software Engineering*.
[101] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and improving regression test selection in continuous integration. In *Proceedings of the International Symposium on Software Reliability Engineering*.
[102] Eliezio Soares, Gustavo Sizilio, Jadson Santos, Daniel Alencar da Costa, and Uirá Kulesza. 2022. The effects of continuous integration on software development: A systematic literature review. *Empirical Software Engineering* 37 (2022), 78.
[103] Daniel Ståhl and Jan Bosch. 2013. Experienced benefits of continuous integration in industry software product development: A case study. In *Proceedings of the 2013 12th IASTED International Conference on Software Engineering*. 736–743.
[104] Jake Tronge, Jieyang Chen, Patricia Grubel, Tim Randles, Rusty Davis, Quincy Wofford, Steven Anaya, and Qiang Guan. 2021. BeeSwarm: Enabling parallel scaling performance measurement in continuous integration for HPC applications. In *Proceedings of the International Conference on Automated Software Engineering*.
[105] Michele Tufano, Hitesh Sajnani, and Kim Herzig. 2019. Towards predicting the impact of software changes on building activities. In *Proceedings of the International Conference on Software Engineering: New Ideas and Emerging Results*.
[106] Bogdan Vasilescu, Stef Van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark G. J. van den Brand. 2014. Continuous integration in a social-coding world: Empirical evidence from GitHub. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, Los Alamitos, CA, 401–405.
[107] 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 Joint Meeting on Foundations of Software Engineering*.
[108] 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 *Proceedings of the International Conference on Software Maintenance and Evolution*.
[109] Kaiyuan Wang, Daniel Rall, Greg Tener, Vijay Gullapalli, Xin Huang, and Ahmed Gad. 2021. Smart build targets batching service at Google. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*.
[110] Yuqing Wang, Mika V. Mäntylä, Zihao Liu, and Jouni Markkula. 2022. Test automation maturity improves product quality—Quantitative study of open source projects using continuous integration. *Journal of Systems and Software* 188 (2022), 111259.
[111] 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 Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.
[112] Wikipedia. 2019. Cold Start (Recommender Systems). Retrieved February 21, 2019 from https://en.wikipedia.org/w/index.php?title=Cold_start_(computing)&oldid=883021431.
[113] Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. 2009. Predicting build failures using social network analysis on developer communication. In *Proceedings of the International Conference on Software Engineering*.
[114] Jing Xia and Yanhui Li. 2017. Could we predict the result of a continuous integration build? An empirical study. In *Proceedings of the International Conference on Software Quality, Reliability, and Security Companion*.
[115] Jing Xia, Yanhui Li, and Chuanqi Wang. 2017. An empirical study on the cross-project predictability of continuous integration outcomes. In *Proceedings of the Web Information Systems and Applications Conference*.
[116] Zheng Xie and Ming Li. 2018. Cutting the software building efforts in continuous integration by semi-supervised online AUC optimization. In *Proceedings of the International Joint Conferences on Artificial Intelligence*.

[117] Shin Yoo and Mark Harman. 2007. Pareto efficient multi-objective test case selection. In *Proceedings of the International Symposium on Software Testing and Analysis*.

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

[119] Fiorella Zampetti, Vittoria Nardone, and Massimiliano Di Penta. 2022. Problems and solutions in applying continuous integration and delivery to 20 open-source cyber-physical systems. In *Proceedings of the 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR'22)*.

[120] 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 *Proceedings of the International Conference on Mining Software Repositories*.

[121] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. 2020. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering* 25 (2020), 1095–1135.

[122] Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. 2019. A large-scale empirical study of compiler errors in continuous integration. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[123] Chen Zhang, Bihuan Chen, Xin Peng, and Wenyun Zhao. 2022. BuildSheriff: Change-aware test failure triage for continuous integration builds. In *Proceedings of the International Conference on Software Engineering*.

[124] Lingming Zhang. 2018. Hybrid regression test selection. In *Proceedings of the International Conference on Software Engineering*.

[125] 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 International Conference on Automated Software Engineering*.

[126] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A framework for checking regression test selection tools. In *Proceedings of the International Conference on Software Engineering*.

[127] Celal Ziftci and Jim Reardon. 2017. Who broke the build? Automatically identifying changes that induce test failures in continuous integration at Google scale. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice Track*.

Received 6 December 2021; revised 2 October 2022; accepted 3 November 2022