A Cost-efficient Approach to Building in Continuous Integration

Xianhao Jin
Department of Computer Science
Virginia Tech
Blacksburg, USA
xianhao8@vt.edu

Francisco Servant
Department of Computer Science
Virginia Tech
Blacksburg, USA
fservant@vt.edu

ABSTRACT
Continuous integration (CI) is a widely used practice in modern software engineering. Unfortunately, it is also an expensive practice — Google and Mozilla estimate their CI systems in millions of dollars. In this paper, we propose a novel approach for reducing the cost of CI. The cost of CI lies in the computing power to run builds and its value mostly lies on letting developers find bugs early when their size is still small. Thus, we target reducing the number of builds that CI executes by still executing as many failing builds as early as possible. To achieve this goal, we propose SmartBuildSkip, a technique which predicts the first builds in a sequence of build failures and the remaining build failures separately. SmartBuildSkip is customizable, allowing developers to select different preferred trade-offs of saving many builds vs. observing build failures early. We evaluate the motivating hypothesis of SmartBuildSkip, its prediction power, and its cost savings in a realistic scenario. In its most conservative configuration, SmartBuildSkip saved a median 30% of builds by only incurring a median delay of 1 build in a median of 15% failing builds.

CCS CONCEPTS
• Software and its engineering → Empirical software validation; Software testing and debugging; Maintaining software; Software maintenance tools.

KEYWORDS
continuous integration, build prediction, maintenance cost

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

1 INTRODUCTION
Continuous integration (CI) is a popular practice in modern software engineering that encourages developers to build and test their software in frequent intervals [15]. For simplicity and consistency with previous studies, we refer as build to the full process of building the software and running all the tests when CI is triggered.

While CI is widely recognized as a valuable practice, it also incurs a very high cost — mostly for the computational resources required to frequently run builds [24–26, 45, 70]. Overall, adopting CI can be very expensive. Google estimates the cost of running its CI system in millions of dollars [26], and Mozilla estimates theirs as $201,000 per month [31]. For smaller-budget companies that have not yet adopted CI, this high cost can pose a strong barrier.

In this paper, we aim to reduce the high cost of CI while keeping as much of its value as possible. The cost of CI is commonly defined by the cost of builds [26, 43], and its value is defined by its ability to reveal problems early [10, 15]. Thus, we aim to reduce the cost of CI by running fewer builds, while running as many failing builds as early as possible. Our goal also responds to the need to run fewer builds that developers frequently express in Q&A websites[61], which they currently may approach by using CI plug-ins [8, 29, 64] to manually skip builds that they deem “safe”, e.g., changes in README files.

Existing research approaches to save cost in CI include the automatic detection of such non-code changes [2] and techniques to make CI builds faster [24, 37]. In contrast, our proposed approach focuses on skipping builds that are predicted to pass in more complicated cases — for any kinds of changes that happened between builds. Our approach complements existing techniques and could potentially be applied in combination with them.

We propose SmartBuildSkip, a novel approach to reduce the cost of CI based on automatic build-outcome prediction — by skipping builds that it predicts will pass, and running builds that it predicts will fail. Our strategy is motivated by two hypotheses: $H_1$: Most builds in CI return a passing result. We expect that software changes will generally be done carefully, making passing builds more common than failing builds. By this hypothesis, skipping passing builds would produce large cost savings. $H_2$: Many failing builds in CI happen consecutively after another build failure. One of the strongest predicting factors in existing build-outcome predictors is the result of the previous build [23, 43, 73]. Also, Rausch et al. observed build failures mostly occurring consecutively in a small number of Java projects [47]. By this hypothesis, most failing builds could be easily predicted — since most follow another build failure.

Thus, SmartBuildSkip differentiates between first failures and subsequent failures, following a two-phase process. First, SmartBuildSkip uses a machine-learning classifier to predict build outcomes to catch first failures. After it observes a first failure, it then determines that all subsequent builds will fail — until it observes a build pass and then changes its operation to predicting again. This strategy aims to address the limitations of existing build-prediction approaches [23, 43, 73], which strongly rely on the outcome of the
The results of our studies support our hypotheses — build passes are numerous (median 87% of all builds), and subsequent failures are also a high proportion of all builds (median 52%). In our experiments, SmartBuildSkip significantly improved the accuracy of the state-of-the-art build predictor — up to median 8% F-measure for first failures, and up to median 52% F-measure for all failures. Finally, SmartBuildSkip’s predictions resulted in high savings of build effort that could be customized for developers with different preferred trade-offs, i.e., faster observation of build failures vs. higher savings in build effort. In its most conservative configuration, SmartBuildSkip saved a median 30% of all builds by only incurring a median delay of 1 build in a median 15% build failures. In a more cost-saving-focused configuration, SmartBuildSkip saved a median 61% of all builds by incurring a 2-build delay for 27% of build failures. This paper provides the following contributions:

- The conceptual separation of build failures into first and subsequent failures, to improve the effectiveness of build prediction models.
- Two studies, of the prevalence of build passes over build failures, and of subsequent failures over first failures.
- A study of factors that predict first failures.
- SmartBuildSkip, a customizable, automatic technique to save cost in CI by predicting build outcomes, that can be applicable with or without training data, and that improves the prediction effectiveness of the state-of-the-art.
- A collection of simple predictors, based on factors that predict first failures, that can be applied as a rule-of-thumb, with no adoption cost.
- An evaluation of the extent to which SmartBuildSkip can save cost in CI while keeping most of its value, with the ability of customizing its cost-value trade-off.

2 MOTIVATING HYPOTHESES

We motivate our hypotheses and our proposed approach with an example. Figure 1 depicts an example timeline of builds, the ideal timeline in which we would save most effort, the timeline produced after applying a state-of-the-art build prediction technique, and the timeline produced after applying our approach SmartBuildSkip. The example timeline shows a numbered sequence of builds in CI. We depict passing builds as circles with a P and failing builds as circles with an F. The ideal timeline shows the outcome that an ideal technique would achieve — skipping every passing build and building all failing builds. We depict skipped builds with a dashed empty circle. This ideal timeline depicts our goal of saving cost in CI by running as few builds as possible while running as many failing builds as possible.

We propose SmartBuildSkip following two main hypotheses: $H_1$: Most builds in CI return a passing result. If this was true, our strategy of predicting build outcomes and skipping those expected to pass would provide substantial cost savings — since passing builds would be a majority and they would be skipped.

$H_2$: Many failing builds in CI happen consecutively after another build failure. If true, if we built an automatic approach that predicted that subsequent builds to a failing build will also fail, we would correctly predict a substantial portion of failing builds.

First failures vs. subsequent failures. Assuming that our hypothesis $H_2$ would be supported, we also propose the distinction between first failures — the first build failure inside a sequence of build failures — and subsequent failures — all the remaining consecutive build failures in the sequence. Figure 1 highlights first failures with gray fill.

Limitations of existing work. Figure 1 also illustrates the limitations of applying existing build predictors (e.g., [23, 43, 73]) to the problem of saving cost in CI by skipping passing builds. The timeline for “existing build predictors” uses a diamond to depict the prediction of the outcome of an upcoming build. If the upcoming build is predicted to pass, the technique skips it and transitions to predict for the next build. We depict this with an arrow leaving the diamond and going into the next diamond, e.g., in build 2. If the upcoming build is predicted to fail, it is executed. We depict this with an arrow leaving the diamond and going into the next build. We posit that existing predictors, by not distinguishing first and subsequent failures, likely provide limited accuracy for both.

Limited prediction of first failures. We posit that existing predictors will rarely correctly predict first failures, because they strongly rely on the status of the previous build for prediction. first failures are preceded by a build pass, by definition. However, we expect that it’s more often build passes that are preceded by a build pass. Thus,
We designed SmartBuildSkip by following the two hypotheses that we described in §2. We also include the timeline produced by SmartBuildSkip for our motivating example in Figure 1.

**SmartBuildSkip’s overall strategy.** SmartBuildSkip follows a two-phase strategy. In its first phase, SmartBuildSkip predicts the outcome of the next build based on a set of predicting features (see §6). If the build is predicted to pass, it is not executed — its cost is saved — and SmartBuildSkip will predict again for the next build. An example is build 5 in Figure 1. If the build is predicted to fail, SmartBuildSkip executes it and checks its outcome. If the actual outcome of the executed build is pass, SmartBuildSkip will predict again for the next build — as in build 8 in Figure 1. If the actual outcome of the executed build is fail, SmartBuildSkip will shift to its second phase — as in build 2 in Figure 1. In its second phase, SmartBuildSkip determines that all subsequent builds will fail and thus executes them until the build passes, after which it returns to the first phase — as in builds 2–4 in Figure 1.

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

The downside of this approach is that, by continuously building after observing a first failure, one false positive is guaranteed for every sequence of failures — as in builds 5 and 8 in Figure 1. However, we believe that this downside is smaller than the benefit that SmartBuildSkip gets from its overall strategy. Besides, existing predictors will also likely incur in these false positives because they strongly rely on the last build status — which in these cases is a bad predictor. Finally, we argue that these first-pass builds are valuable for practitioners, because they inform them of when they have fixed the problem that caused the build to fail.

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

**SmartBuildSkip-Within:** This variant is trained in the past builds within the same software project in which it is applied. It uses the build features that we report in §6.

**SmartBuildSkip-Cross:** This variant is trained in the past builds of different software projects than the one in which it will be applied. It uses the build features as well as the project features that we report in §6. We propose this variant to help with the cold-start problem [71] in software projects for which only a few builds have been executed and they would not be enough to provide high-quality predictions.

## 4 RESEARCH QUESTIONS

We perform two empirical studies to test our hypotheses and inform the design of SmartBuildSkip. Then, we perform two experiments to evaluate it. In our studies and experiments, we answer the following research questions:

**Empirical Study 1: Evaluating our Motivating Hypotheses**

RQ1: Are passing builds more numerous than failing builds?

RQ2: Are subsequent failures numerous?

**Empirical Study 2: Characterizing first failures**

RQ3: What features predict first failures?

**Experiment 1: SmartBuildSkip for Build Prediction**

RQ4: How effective is SmartBuildSkip predicting first failures?

RQ5: How effective is SmartBuildSkip predicting all failures?

**Experiment 2: SmartBuildSkip for Build Effort Reduction**

RQ6: How many resources, i.e. builds, will SmartBuildSkip save?

RQ7: What is the value trade-off for such resource savings?
Study Subjects. We perform our study over the TravisTorrent dataset [3], which includes 1,359 projects (402 Java projects and 898 Ruby projects) with data for 2,640,825 build instances. We remove “toy projects” from the data set by studying those that are more than one year old, and that have at least 200 builds and at least 1,000 lines of source code, which is a criteria applied in multiple other works [27, 43]. After this filtering, we obtained 274,742 builds from 359 projects (53,731 failing builds). We focused our study on builds with passing or failing result, rather than error or canceled — since they can be exceptions or may happen during the initialization and get aborted immediately before the real build starts. Besides, in Travis a single push or pull-request can trigger a build with multiple jobs, and each job corresponds to a configuration of the building step [16, 81]. We did a preliminary investigation of these builds and found that these jobs with the same build ID normally share the same build result and build duration. Thus, as many existing papers have done [16, 28, 48], we considered these jobs as a single build. We applied LOD [60] to remove outliers that have higher or lower than three standard deviations above or below the mean number of the failing ratio.

5  EMPIRICAL STUDY 1: EVALUATING OUR MOTIVATING HYPOTHESES

RQ1: Are passing builds more numerous than failing builds? We first evaluate our motivating hypotheses to understand if our approach to save build effort in CI is promising. Our first hypothesis posits that passing builds will be numerous — and thus skipping them would provide high build-effort savings in CI.

Research Method. We measured the proportion of all builds to all builds in each studied project, and we show the distribution of such ratios in Figure 2.

Result. For most projects, the passing builds represented a very large proportion — with a median 88% (and a mean 84%) of all builds passing. This result supports our hypothesis that skipping passing builds would strongly save build effort in CI, since they generally represented a large portion of the executed builds. Furthermore, this result also shows the upper bound for how many builds could be saved — given a “perfect” technique that would correctly predict every single passing build.

RQ2: Are subsequent failures numerous? Our next hypothesis posits that subsequent failures will be numerous — and thus predicting that subsequent builds to a failing build will also fail would correctly predict a substantial portion of failing builds.

Research Method. We measured the proportion of subsequent failures to all failures for each project (e.g., in a build history P-F-F-F-P, the ratio of subsequent failures to all failures is 2/3). We show the distribution of these proportions for all projects in Figure 3.

Result. Figure 3 supports the hypothesis that subsequent failures are numerous, i.e., there are many of them. A high number of projects had a high (i.e., not low) ratio of subsequent failures: >52% for 50% of projects, and >38% for >75% of projects. Thus, our approach would correctly predict a high proportion of all build failures, since we expect it to correctly predict all subsequent failures. Once it observes a failure, it would correctly predict all the subsequent ones.

6  EMPIRICAL STUDY 2: CHARACTERIZING FIRST FAILURES

RQ3: What features predict first failures? We found that subsequent failures are numerous and easy to predict. Next, we will focus on predicting first failures. To inform our prediction technique, we perform a second empirical study to identify features that characterize them.

Research Method. We study two different kinds of features to characterize first failures: build features and project features. As build features, we selected all the features included in TravisTorrent that previous studies found to be correlated with all build failures, e.g., [27, 47]. Our goal was to study whether such features are also correlated with first failures. Then, to be able to address the cold-start problem [71], we also created four project features that could be used for cross-project predictions. Our intuition is that project features would aid the classifier in “adapting” its trained model across projects of different characteristics — since projects using continuous integration are diverse [18]. To the extent of our knowledge, no previous work studied the correlation between all build failures (or first failures) and these project features (as defined by us, with a single value per project). We list in Table 1 the features that we studied, along with a brief description.

Build features. Build features will be useful to train our approach with past builds from the same software project. To identify build features that have a relationship with first failures, we first removed subsequent failures from our studied dataset (§4).

Then, we measured the correlation between the ratio of first failures to all builds (which now only included first failures and passing builds) and each studied build feature in each studied project. For each value of a build feature in a project, we measured the ratio
Table 1: Features studied for correlation with first failures.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src_churn (SC)</td>
<td>The number of changed source lines since the last build.</td>
</tr>
<tr>
<td>file_churn (FC)</td>
<td>The number of changed source files since the last build.</td>
</tr>
<tr>
<td>test_churn (TC)</td>
<td>The number of changed test lines since the last build.</td>
</tr>
<tr>
<td>num_commit (NC)</td>
<td>The number of commits since the last build.</td>
</tr>
<tr>
<td>Project_performance_short (PS)</td>
<td>The proportion of passing builds in the recent five builds.</td>
</tr>
<tr>
<td>Project_performance_long (PL)</td>
<td>The proportion of passing builds in the whole previous builds.</td>
</tr>
<tr>
<td>Time_frequency (TF)</td>
<td>The time gap (hour) since the last build.</td>
</tr>
<tr>
<td>Failure_Distance (FD)</td>
<td>The number of builds since the last failing build.</td>
</tr>
<tr>
<td>Week_day (WD)</td>
<td>The weekday [0, 6] (0 being Monday) of the build.</td>
</tr>
<tr>
<td>Day_time (DT)</td>
<td>The time of day [0, 23] of the build.</td>
</tr>
<tr>
<td>Test_density (TD)</td>
<td>The median number of test cases per 1000 executable production source lines over the project’s CI usage history.</td>
</tr>
</tbody>
</table>

Project features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Team_size (TS)</td>
<td>The median number of developers over the project’s CI usage history.</td>
</tr>
<tr>
<td>Project_size (PS)</td>
<td>The median number of executable production source lines of code in the repository over the project’s CI usage history.</td>
</tr>
<tr>
<td>Project_age (PA)</td>
<td>The time duration between the first build and the last build for that project.</td>
</tr>
<tr>
<td>Test_density (TD)</td>
<td>The median number of lines of test cases per 1000 executable production source lines over the project’s CI usage history.</td>
</tr>
</tbody>
</table>

of first failures to all builds that have that value for that feature in the project. For continuous features, such as src_churn, we use the Pearson correlation coefficient as effect size and its corresponding p-value for the significance test. For categorical variables, such as week_day, we measure effect size using Cramer’s V and we use Pearson’s $X^2$ for the statistical significance test.

Project features. Project features will be useful to train our approach with past builds from other software projects. When no (or few) past builds are available for a software project, we could use past builds from different software projects to train our predictor. This situation is known in machine learning as the cold-start problem [71]. In such cases, our predictor will use project features to learn how representative past builds from other projects are for the project for which not enough past builds existed.

As we did to study build features, we also removed subsequent failures to study project features. Then, we measured the correlation across projects between the value of each project feature and the project’s ratio of first failures to all builds. Since all features were continuous, we applied Pearson’s correlation coefficient and decided statistical significance for $p < 0.05$.

Results. We report the results of our correlation analysis for build and project features.

Build features. We show in Figure 4 the correlation between different build features and the ratio of first failures. Each box in the box plot represents the distribution of correlation coefficients between a feature (see Table 1) and the ratio of first failures, for all the projects for which that feature’s correlation was statistically significant ($p < 0.05$). We report the percentage of projects for which a feature’s correlation was statistically significant in its label in the X axis.

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

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

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

Another interesting observation is that most of the build features that did not show strong statistical correlation with first failures are those that intuitively would be strongly correlated with subsequent failures instead. That is, subsequent failures happen after a particularly short number (zero) of failing builds (FD), after a particularly low proportion of passing to failing builds (PS, PL), and probably a particularly short time after another build (TF). Intuitively, first failures would not particularly have any of these characteristics.

Project features. We use a bar chart to show each project feature and its corresponding correlation coefficient. The value following the name of each project feature represents its corresponding p-value. We found three project features for which first failures were more prevalent, i.e., for which the project feature increased and its difference is statistically significant (Figure 5): test density (TD), project size (PS), and project age in CI (PA). These are the features that we will use to design our technique to train across projects.

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

7 EXPERIMENT 1: EVALUATING BUILD-FAILURE PREDICTION

RQ4 & RQ5: How effective is SmartBuildSkip predicting first failures and all failures? In our second empirical study, we discovered features that predict first failures (§6). Next, we use them in SMARTBUILD to evaluate it.

We evaluate SMARTBUILD in two experiments that complement each other. First, we evaluate its effectiveness for predicting build failures (§7), and then we evaluate the cost reduction that its predictions provide in practice (§8).

Our first experiment (§7) allows us to compare the effectiveness of SMARTBUILD with that of existing build-prediction techniques (e.g., [23, 43, 73]) in the scenario in which they were originally proposed and evaluated: a scenario in which the information about previous builds is always known — ignoring that it would not be available if a previous build was skipped. Automatic build prediction in such a scenario can be useful to give developers more confidence about their code changes, e.g., [23] — even if they did not skip builds.

Our second experiment (§8) allows us to evaluate how much cost SMARTBUILD would save in CI in our target scenario — a practical scenario in which the outcome of builds that were skipped is unknown.

Research Method. We evaluate the prediction effectiveness of SMARTBUILD in comparison to the state-of-the-art build-prediction technique: HW17 [23]. To better understand the benefit of SMARTBUILD’s two-stage design (see §3), we separately evaluate predictions for first failures and all failures. We evaluate both techniques over our dataset described in §4, and we measure their prediction effectiveness using precision, recall, and F1 score. We tested our results for statistical significance with a two-tailed Wilcoxon test, and decided statistical significance for $p < 0.05$.

State-of-the-art Build-prediction Technique: HW17. To provide a point of reference for this evaluation, we replicated the state-of-the-art build prediction technique: HW17 [23]. We use the acronym HW17 to refer to it — the first letter of the authors’ last names and its publication date — since the authors did not assign it a specific name. To the extent of our knowledge, HW17 is the existing build prediction technique that provided the highest precision and recall.

HW17 predicts build outcomes with a random-forest machine-learning algorithm, informed by a collection of 16 features about the current build, 4 features about the previous build, and 8 features generated from analyzing build logs. In contrast, SMARTBUILD requires only 4 current-build features, no previous-build features, and 3 project features (§6). Only a few features are considered by both HW17 and SMARTBUILD: SC, FC, and TC (see their descriptions in §6). Hassan and Wang found these features to be correlated with all failures [23], and we found them to be correlated with first failures (see §6).

Our proposed approach SMARTBUILD provides two main strengths over HW17 for saving cost in continuous integration: (1) SMARTBUILD runs predictions only for first failures, and determines that all subsequent builds fail until a pass is observed. HW17 does not make such distinction, and runs predictions for all builds. We posit that SMARTBUILD’s strategy will be more effective at predicting both first and subsequent failures (which we evaluate in this experiment). (2) SMARTBUILD predicts based only on features describing the current build and the project, but it does not rely on features about the previous build. HW17, like the other existing build-prediction approaches, does rely on the outcome of the previous build, among other features. We posit that such choice would make HW17 much less effective in real-world usage: whenever a previous build is skipped, its outcome will be unknown, which would negatively impact HW17 but not SMARTBUILD (which we study in §8).

Predicting first failures vs. all failures. We evaluate the prediction of first failures and all failures over two different datasets. For predicting first failures, we removed subsequent failures from our dataset and evaluated our studied techniques over it. For predicting all failures, we evaluated our studied techniques over the dataset originally used to evaluate HW17 [23], which contains both first and subsequent failures, i.e., all failures. The paper’s authors generously shared this dataset with us and we applied to it the same curation that we described in §4. Like our dataset, HW17’s is also obtained from TravisTorrent [4] — HW17’s dataset is in fact a subset of ours. HW17’s dataset includes only Java projects that use the Ant, Maven, or Gradle build systems. In total, their dataset contains 37 projects.

This decision strengthens our experiment in two ways. (1) Performing our evaluation for all failures over HW17’s dataset allows us to make a fair comparison between HW17 and SMARTBUILD. HW17 relies on some pre-computed features about the preceding...
Cross-validation. We perform 8-fold cross validation, also to study the same conditions in which HW17 was evaluated. Thus, we randomly divided our dataset (§4) into 8 subsets of builds, i.e., folds, iteratively using one of them as our test set and the remaining ones as our training set, until we have used every fold as test set.

We evaluated the Within variations of our studied techniques applying cross-validation individually for each software project — randomly dividing the set of builds of the same software project into subsets. We evaluated the Cross variations of our studied techniques applying cross-validation across software projects — randomly dividing the set of projects in our dataset into subsets of projects, and using all the builds within a project for testing or training, accordingly. In both cases, we report the results of our evaluation metrics per software project.

**Independent Variable: Technique.** We evaluate four different approaches: our proposed approaches and HW17 [23], in their Within and Cross variants.

**SMARTBUILD-SKIP-WITHIN:** Our proposed approach described in §3, trained in the same software project, using the predicting build features that we discovered in §6.

**SMARTBUILD-SKIP-CROSS:** Our proposed approach described in §3, trained in other software projects, using the predicting build features and project features that we discovered in §6.

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

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

**Dependent Variables.** We used three metrics to evaluate our studied techniques: precision, recall, and F1 score. We calculated the value of these metrics for each studied software project, first for the set of first failures, and then for the set of all failures.

We measured precision as the number of correctly predicted build failures divided by the number of builds that the technique predicted as build failures. We measured recall as the number of correctly predicted build failures divided by the number of actual build failures. We measured F1 score as the harmonic mean of precision and recall.

**Results.** We plot the results of this experiment in Figure 6 for the prediction of first failures, and in Figure 7 for the prediction of all failures. The boxes in these box plots for each dependent variable represent its distribution of values for all the studied projects. We discuss our observed differences in results in terms of absolute percentage point differences over the median value of each metric across projects.

**Predicting first failures.** SMARTBUILD-SKIP improved HW17’s median precision by 3% for its Within approach and by 9% for its Cross approach. SMARTBUILD also improved HW17’s median recall by 4% for its Within approach and by 7% for its Cross approach. These differences were statistically significant ($p < 0.05$). We posit that SMARTBUILD-SKIP-CROSS provided an even higher improvement because its training set was much larger — encompassing multiple projects — and because build features likely vary little from project to project. These findings validate our hypothesis in §2 that separately predicting first failures is more effective than training a predictor based on features from all failures.

**Predicting all failures.** SMARTBUILD-SKIP improved HW17’s median precision by 16% for its Within approach and was 9% worse for its Cross approach. It also improved HW17’s median recall by 28% for its Within approach and by 68% for its Cross approach. These differences were statistically significant ($p < 0.05$). We posit that
SMARTBUILDSKIP’s precision and recall are now much higher than HW17’s because it is much better at predicting subsequent failures. We also observed that both techniques generally improved both their precision and recall. We believe that this is due to the increase in the number of failing builds in the dataset — after adding subsequent failures), allowing all techniques to learn them better. This is particularly acute for SMARTBUILDSKIP’s CROSS variants, which became much more inclined to predict build failures after being trained with much more data (across projects), which dramatically increased its recall, but reduced its precision. These findings also validate our hypothesis in the section that choosing to always build after a failure is a highly successful strategy to predict subsequent failures.

8 EXPERIMENT 2: EVALUATING CI COST REDUCTION

RQ6 & RQ7: How many resources, i.e., builds, will our approaches save? What is the value trade-off for such resource savings? After finding that SMARTBUILDSKIP improves the precision and recall of the state-of-the-art build predictor, we measure the cost reduction that it would provide in practice.

Research Method. We now simulate the more realistic scenario in which the builds that are skipped are not available for training. We use the same setting as in §7, with one change. Now, when a predictor predicts the upcoming build as a pass, we skip the build, and accumulate the value of the build-level features for the next coming build. We only update the information connected to the last build when the predictor actually decides to build. In this context, we measure four metrics for each evaluated technique: how many builds it saves, how many failing builds are observed immediately (and how many with a delay), the delay length of delayed failing builds, and a new metric to measure the balance between failing build observation delay and build execution saving.

Independent Variable: Technique. We evaluate the same four predictors as in Experiment 1, in addition to a new collection of techniques that we call rule-of-thumb techniques. In the spirit of cost-saving, we propose this additional collection of techniques because of their low adoption cost. These rule-of-thumb techniques are based on the individual build features that we observed in §6. They simply decide to skip builds when the given feature value is below a certain threshold. We propose these techniques as a potentially “good-enough” alternative for software teams that do not have the resources to implement and adopt SMARTBUILDSKIP, or for them to use in the time period while they are implementing it. Finally, we also include a “Perfect” technique that would skip all passing builds and run all failing builds — as a reference for how many builds could be desirably skipped.

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

We evaluated 50 different thresholds (values 1–50), which meant: absolute value for the “rule-of-thumb” techniques, and predicted likelihood (in percentage) of the build to fail for SMARTBUILDSKIP.


Studied dataset. Since this experiment is focused on predicting all builds, we also use the dataset in which HW17 was originally evaluated (§7).

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

Results. We plot the results for our Experiment 2 in Figure 8. This figure shows the median value for each metric across studied projects. For Failing-build Delay, it’s the median across projects of their median Failing-build Delay. The Y axis is the metric for evaluation and each box contains every project’s result. The X axis has different meanings for different techniques: the threshold for rule-of-thumb techniques (e.g., threshold 5 for $#src\_files$ means that <5 files were changed in that build), or the prediction sensitivity (in percentage) for the predictors.

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

HW17 achieved the poorest saving efficiency. As we anticipated in §2, HW17 predicted most builds to pass because it relied too much on the status of the last build. It saved a large amount of builds, but it also executed very few failing builds as a result.

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

Finally, if more conservative or more risky approaches are preferred. Figure 8 shows a wide variety of trade-offs that could be achieved by different techniques and configurations.


9 DISCUSSION

Diverse Cost-saving Needs. Different developers will have different preferences in the trade-off between observing failing builds early and saving build effort. Thus, we propose SmartBuildSkip as a customizable solution, with an adjustable prediction sensitivity. Some developers may value observing failing builds early much more than saving cost (but still want to save some cost), e.g., developers at large companies that have been using CI for some time and are exploring ways to reduce its cost (like Facebook [37], Microsoft [24], or Google [12]). These developers could configure SmartBuildSkip in its most conservative sensitivity (0) and save the cost of 30% of their builds while only introducing a 1-build delay in 15% of their build failures.

In contrast, other developers may be looking for a way to reduce CI’s high-cost barrier [70] to adopt it, even if it means observing build failures less quickly. These developers could configure SmartBuildSkip with a more liberal sensitivity (2) and save the cost of 61% of their builds and still observe 73% failing builds with no delay (and the remaining 27% with a 2-build delay). In this scenario, SmartBuildSkip dramatically lowers the cost of CI for non-adopters, letting them still get a strong value from it — particularly considering that non-adopters currently do not benefit from CI at all. Furthermore, as developers’ budgets increase, they could also adapt the sensitivity of SmartBuildSkip over time to build more and observe failures more quickly.

The Impact of Delayed Failing Builds. Our approach reduces the cost of CI, but it also reduces its value — it delays the observation of some build failures. Some existing techniques target developers who cannot afford a single delayed failing build — by skipping only tests [59] or commits [2] that are guaranteed to pass, i.e., tests for other modules and non-code changes. In exchange for such guarantee, this strategy is limited in how much cost it can save — the number of guaranteed-pass tests and commits.

Our proposed technique targets developers for whom some delay in failure observation is acceptable — as do existing techniques based on test selection. Such techniques, which introduce failure observation delays, are valued and adopted by many large software companies, e.g., Google [11], Microsoft [24], or Facebook [37]. We argue that, for many developers, the cost savings provided by SmartBuildSkip overcome the introduced delay in failure observation — particularly for SmartBuildSkip’s most conservative sensitivities, which produce a delay of one or two builds. For context, Herzig et al.’s approach [24] (deployed at Microsoft) introduced a delay of 1–3 builds. Ultimately, though, we believe that different developers would prefer different cost-saving trade-offs, which is why we made SmartBuildSkip customizable.

Other Purposes of CI. The main reason for developers to use CI is to catch bugs earlier [25], but they also use it to: have a common build environment, make integrations easier, enforce a specific workflow, simplify testing across multiple platforms, be less worried about breaking builds, deploy more often, and have faster iterations, [25, 26]. Most (the first four) of these purposes are achieved as soon as CI is adopted, so we do not expect them to be impacted by introducing a cost-saving technique like SmartBuildSkip. However, the last three purposes (and others like safety-checking pull requests) may
be impacted, since they benefit from observing build passes. This applies to both our and existing techniques that skip tests or builds.

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

Furthermore, SMARTBUILDSKIP provides an additional benefit over existing test-selection-based techniques for purposes that rely on build observations. Test-selection techniques may give a false sense of confidence [45] when a build that should have failed instead passes because some of its failing tests were skipped.

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

10 THREATS TO VALIDITY

Construct Validity. We use metrics as proxies to represent the value — early observation of build failures — and cost — build execution — in CI. However, these are metrics that developers have reported as describing the value and cost of CI, e.g., [10, 15, 26, 43], and are metrics that other existing approaches for saving cost in CI have used, e.g., [2, 37]. Herzeg et al. [24] assign specific dollar amounts to each test case that is saved and each failure observation that is delayed. We avoid using their numbers, since they were calculated at Microsoft and will probably be different at other companies.

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 and we filtered outliers and toy projects out of it. Our results may also be affected by flaky tests causing spurious failing builds. However, CI systems are expected to function even in the presence of flaky tests, since most companies do not consider it economically viable to remove them, e.g., [37, 40].

Another threat could be the risk of over-fitting in our empirical study 2 ($\S6$), since we performed it over our complete data set — since we aimed to increase the generalizability of our observed correlated features. To address the over-fitting risk, we repeated our study on the chronologically earlier half of data for build features and a half of projects for project features through stratified random sampling [9] on number of builds. The features selected with our original criteria remained the same (correlation coefficients — SC: 0.68, FC: 0.91, TC: 0.75, NC: 0.73, TD: 0.16, PS: 0.22, PA: 0.18).

Also, our usage of cross-validation may result in placing future builds in the training sample. An alternative approach would have been to use chronological training and testing, e.g., [5, 56, 63]. However, our goal was to compare SMARTBUILDSKIP with HW17 in the scenario in which it was originally proposed and evaluated, i.e., using cross-validation. Nevertheless, we believe that SMARTBUILD-SKIP would provide similar precision and recall in a chronological experiment, since it uses build features that likely do not vary much over time, i.e., we believe that SC, FC, TC, and NC do not necessarily vary significantly as projects age. Furthermore, SMARTBUILDSKIP’s cross-project variant is not affected by this threat, since it was trained in different projects than it was tested.

Finally, we also increase our internal validity by validating the hypothesis that influence our proposed technique via studies 1 ($\S5$) and 2 ($\S6$).

External Validity. To increase external validity, we selected the popular dataset Travis CI, which has been analyzed by many other research works. The projects we chose were all Java or Ruby projects, because there are no projects with other programming languages in the data set. Although these two programming languages are popular, different CI habits in other languages may provide slightly different results to the ones in this study. Finally, our cost-saving technique may not be suitable for software projects that cannot afford a single delay in observing failing builds. We target projects that can afford some delay in exchange for the cost savings, as do other techniques that skip builds, e.g., [2] or tests, e.g., [37].

11 RELATED WORK

Empirical Studies of CI and its Cost. Multiple researchers focused on understanding the practice of CI, studying both practitioners e.g., [26] and software repositories [68]. Vasilescu et al. studied CI as a tool in social coding [67], and later studied its impact on software quality and productivity [68]. Zhao et al. studied the impact of CI in other development practices, like bug-fixing and testing [79]. Stahl et al. [62] and Hilton et al. [26] studied the benefits and costs of using CI, and the trade-offs between them [25]. Lepannen et al. similarly studied the costs and benefits of continuous delivery [34]. Felidré et al. [13] studied the adherence of projects to the original CI rules [15]. Other recent studies focused on the difficulties [45] and pain points [70] of CI.

The high cost of running builds is highlighted by many empirical studies as an important problem in CI [24–26, 45, 70] — which reaches millions of dollars in large companies, e.g., at Google [26] and Microsoft [24].

Approaches to Reduce the Cost of CI. A popular effort to reduce the cost of CI focuses on understanding what causes long build durations e.g., [19, 66]. Thus, most of the approaches that reduce the cost of CI aim at making builds faster by running fewer test cases on each build. Some approaches use historical test failures to decide which tests to run [12, 24] Others run tests with a small distance with the code changes [39] or skip those testing unchanged modules [59]. Recently, Machalica et al. predicted test case failures using a machine learning classifier [37]. These techniques are based on the broader field of regression test selection (RTS) e.g., [20, 49, 50, 74, 78, 80]. While these techniques focus on making every build cheaper, our work addresses the cost of CI differently: by reducing the total number of builds that get executed. A related recent technique saves cost in CI by not building when builds only include non-code changes [1, 2]. Our technique predicts build outcomes for any kind of changes (code and non-code). Thus, our work complements existing techniques to reduce cost in CI, and could potentially be applied in addition to them.

A related effort for improving CI aims at speeding up its feedback by prioritizing its tasks. The most common approach in this
direction is to apply test case prioritization (TCP) techniques e.g., [11, 12, 36, 38, 42, 51] so that builds fail faster. Another similar approach achieves faster feedback by prioritizing builds instead of tests [35]. In contrast, our work focuses on saving cost in CI by skipping tests instead of prioritizing them. Prioritization-based techniques increase feedback speed but do not focus on saving cost, i.e., all builds still get executed, and all passing tests get executed if no test failure is observed.

Finally, other complementary efforts to reduce build duration have targeted speeding up the compilation process e.g., [7] or the initiation of testing machines e.g., [17].

Characterizing Failing Builds. Multiple studies investigated the reasons why builds fail. Some studies [41, 69] found that the most common build failures were compilation [77], unit test, static analysis [76], and server errors. Paixão et al. [44] studied the interplay between non-functional requirements and failing builds. Other studies found factors that contribute to build failures: architectural dependencies [6, 52] and other more specific factors, such as the stakeholder role, the type of work item and build [32], or the programming language [3]. Other less obvious factors that could cause build failures are build environment changes or flaky tests [47]. Rausch et al. [47] also found that build failures tend to occur consecutively, which Gallaba et al. [16] describe as “persistent build breaks”. These observations inform our hypothesis that subsequent build failures would be numerous and easy to anticipate.

Other studies found change characteristics that correlate with failing builds, such as: number of commits, code churn [27, 47], number of changed files, build tool [27], and statistics on the last build and the history of the committer [43]. In our study, we separate failing builds into first failures and subsequent failures. We found that first failures are predicted by some of the factors that predict all builds (line, file, and test churn, and number of commits), but also by factors that were not found to correlate with all builds (project size, age, and test density).

Finally, other studies investigated the characteristics of build failures outside the CI context [22, 46, 65]

Predicting Failing Builds. Some works aimed at predicting build outcomes in industrial settings where continuous integration was not yet adopted. These techniques mostly approached this problem using machine learning classifiers, e.g., measuring social and technical factors and using decision trees [21]; applying social network analysis and measuring socio-technical factors [33, 72]; and using code metrics on incremental decision trees [14].

In the continuous integration context, Ni and Li [43] predict build outcomes using cascade classifiers measuring statistics about the last build and the committer of the current build. Xie and Li [73] use a semi-supervised method over change metrics and the last build’s outcome. Hassan and Wang [23] use a predictor over the last build’s status and type. Since all these predictors rely on the outcome of the last build to be known, their prediction power may be limited in a cost-saving context, where the last build means the last build that was executed. In contrast to these predictors, SMARTBUILDSKIP is not affected by how stale the last build status is, since it does not rely on it for its prediction.

12 CONCLUSIONS AND FUTURE WORK

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

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

In the future, we will work on extending SMARTBUILDSKIP’s algorithm with static analysis techniques to predict build failures based on characteristics of the contents of their code changes. We will also explore adding prediction features based on the historical properties of the changed modules between builds, such as their code-change history [53–55, 57, 58]. Currently, SMARTBUILDSKIP benefits from statistical properties of builds. This future approach would focus on taking advantage of their structural properties.

13 REPLICATION

We include a replication package for our paper [30].

REFERENCES

Romit Jain, Saket Kumar Singh, and Bharavi Mishra. 2019. A Brief Study on
Wagner Felidré, Leonardo Furtado, Daniel Alencar Da Costa, Bruno Cartaxo,
sebastian elbaum, retเยe g malleshwarkar, and Gregg rothermel. 2002. Test
case prioritization: A family of empirical studies. IEEE transactions on software
eengineering, ACM, 235–245.
Wagner Felidré, Leonardo Furtado, Daniel Alencar Da Costa, Bruno Cartaxo,
13th ACM/IEEE International Symposium on Empirical Software Engineering and
Measurement. 10.
Jacqui Finlay, Russell Pears, and Andy M Connor. 2014. Data stream mining for
predicting software builds outcomes using source code metrics. Information and
Martin Fowler and Matthew Foemmel. 2006. Continuous integration. Thought-
Noise and heterogeneity in historical build data: an empirical study of Travis CI.
In Proceedings of the 33rd ACM/IEEE International Conference on Automated
Alessio Gambi, zabolotnij rostylav, and Schabram Dastur. 2015. Improving cloud-
based continuous integration environments. In Proceedings of the 37th
Aakash Gautam, Sakti Vodvrasid, and Francisco Servant. 2017. An empirical study
of activity, popularity, size, testing, and stability in continuous integration.
In 2017 IEEE/ACM 14th International Conference on Mining Software Repositories
(MSR). IEEE, 495–498.
study of the long duration of continuous integration builds. Empirical Software
Milos Gilgoric, Lamaya Elouadi, and Darko Marinov. 2015. Practical regression test
selection with dynamic file dependencies. In Proceedings of the 2015 Symposium on
Software Testing and Analysis. ACM, 211–222.
Ahmed E Hassan and Ken Zhang. 2006. Using decision trees to predict the
certification result of a build. In Automated Software Engineering. 2006. ASE'06.
Automatic building of java projects in software repositories: A study on feasibility
Engineering and Measurement. IEEE, 38–47.
Kim Herzg, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015.
The art of testing less without sacrificing quality. In Proceedings of the 37th
Michael Hilton, Niyati Nundell, and Timothy tunnell. 2017. Trade-offs in
continuous integration: assurance, security, and flexibility. In Proceedings of the
Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig.
2016. Usage, costs, and benefits of continuous integration in open-source projects.
In Proceedings of the 31st IEEE/ACM International Conference on Automated Soft-
Md Rakibul Islam and Minhaz F Zibran. 2017. Insights into continuous integra-
tion build failures. In Mining Software Repositories (MSR), 2017 IEEE/ACM 14th
International Conference on. IEEE, 467–470.
Ranit Jinu, Sakti Kumar Singh, and Bhavani Mishra. 2019. A Brief Study on
Build Failures in Continuous Integration: Causation and Effect. In Progress in
27-April-2019].
Xianhao Jin and Francisco Servant. 2019. When Should I Build? A Collection of
Cost-efficient Approaches to Running Builds in Continuous Integration. https://
doi.org/10.5281/zenodo.2667577
Nooredine Kerazzi, Foutse Khomh, and Bram Adams. 2014. Why do automated
builds break? an empirical study. In Software Maintenance and Evolution (ICSME),
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, 5 (2011),
324–337.
Marko Leppinen, Simo Mäkinen, Max Pagels, Velči-Pelka Eloranta, Juha Itkonen,
Mika V Mäntylä, and Toni Mannisto. 2015. The highways and country roads to
Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. 2018. Redefining priori-
tization: continuous prioritization for continuous integration. In 2018 IEEE/ACM
44th International Conference on Software Engineering (ICSE). IEEE, 688–698.
Assessing test case prioritization on real faults and mutants. In 2018 IEEE
International Conference on Software Maintenance and Evolution (ICSEME). IEEE,
248–251.
Mateusz Machalica, Alex Samylink, Meredith Porth, and Satish Chandra. 2019.
Predictive test selection. In Proceedings of the 41st International Conference on
Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. 2013. Test case prioritization for
continuous regression testing: An industrial case study. In 2013 IEEE International
Conference on Software Maintenance. IEEE, 540–543.
Atif Memon, Zehao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siem-
borski, and John Mccoo. 2017. Taming google-scale continuous testing. In Pro-
cedings of the 39th International Conference on Software Engineering: Software
Silver Lake Press, 23–34.
Shaikh Mostafa, Xiaoyin Wang, and Tao Xie. 2017. Perfranker: Prioritization of
performance regression tests for collection-intensive software. In Proceedings of the
26th ACM SIGSOFT International Symposium on Software Testing and Analysis.
ACM, 23–34.
Ansong Ni and Ming li. 2017. Cost-effective build outcome prediction using
cascaded classifiers. In Mining Software Repositories (MSR), 2017 IEEE/ACM 14th
Kleirsson VR Paixão, Cícero Z Felício, Ferdihael, Fernando M Delfim, and Marcelo de A Maia.
2017. On the interplay between non-functional requirements and builds on
continuous integration. In Proceedings of the 14th International Conference on
Gustavo Pinto, Marcel Rebouças, and Fernando Castor. 2017. Inadequate testing,
time pressure, and (over) confidence: a tale of continuous integration users.
In Proceedings of the 10th International Workshop on Cooperative and Human Aspects
Noam Rabban, Michael S Harvey, Sadnan Saquif, Keheliya Gallaba, and Shane
Context. In 2018 IEEE/ACM 15th International Conference on Mining Software
An empirical analysis of build failures in the continuous integration workflows
of Java-based open-source software. In Proceedings of the 14th International
Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing regression test selec-
tion techniques. In Proceedings of the 13th ACM/IEEE International Symposium on
Gustavo Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test
selection technique. ACM Transactions on Software Engineering and Methodology
Gregg Rothermel, Roland H. Untch, Chengyun Chiu, and Mary Jean Harrold.
2001. Prioritizing test cases for regression testing. IEEE Transactions on software
Engineering 27, 10 (2001), 929–948.
A Cost-efficient Approach to Building in Continuous Integration


