Empirical Software Engineering manuscript No.
(will be inserted by the editor)

On Combining Commit Grouping and Build Skip Prediction to Reduce Redundant Continuous Integration Activity

Divya M. Kamath · Eduardo Fernandes ·
Bram Adams · Ahmed E. Hassan

Received: date / Accepted: date

Abstract *Context:* Continuous Integration (CI) is a resource intensive, widely used industry practice. The two most commonly used heuristics to reduce the number of builds are either by grouping multiple builds together or by skipping builds predicted to be safe. Yet, both techniques have their disadvantages in terms of missing build failures and respectively higher build turn-around time (delays). *Objective:* We aim to bring together these two lines of research, empirically comparing their advantages and disadvantages over time, and proposing and evaluating two ways in which these build avoidance heuristics can be combined more effectively, i.e., the ML-CI model based on machine learning and the Timeout Rule. *Method:* We empirically study the trade-off between reduction in the number of builds required and the speed of recognition of failing builds on a dataset of 79,482 builds from 20 open-source projects. *Result:* We find that both of our hybrid heuristics can provide a significant improvement in terms of less missed build failures and lower delays than the baseline heuristics. They substantially reduce the turn-around-time of commits by 96% in comparison to skipping heuristics, the Timeout Rule also enables a median of 26.10% less builds to be scheduled than grouping heuristics. *Conclusions:* Our hybrid approaches offer build engineers a better flexibility in terms of scheduling builds during CI without compromising the quality of the resulting software.

D.M. Kamath
School of Computing, Queen's University, Kingston, ON, Canada
E-mail: divya.kamath@queensu.ca

E. Fernandes
School of Computing, Queen's University, Kingston, ON, Canada
E-mail: eduardo.fernandes@queensu.ca

B. Adams
School of Computing, Queen's University, Kingston, ON, Canada
E-mail: bram.adams@queensu.ca

A.E. Hassan
School of Computing, Queen's University, Kingston, ON, Canada
E-mail: hassan@queensu.ca

Keywords Software build · Build avoidance heuristic · Machine learning · Software analysis · Empirical study

1 Introduction

Continuous Integration (CI) is the practice of periodically integrating code changes from multiple contributors into a single code base [38], in order to identify conflicts as early as possible and hence reduce the overall merge effort. As such, it is one of the most important quality assurance practices in current software development and has been largely adopted by both open source and closed source projects [15]. One of the pillars of CI is automated building and testing of incoming pull requests or commits, a practice confusingly called "CI" as well. Such automated CI builds typically compile a commit on all supported platforms, on each of which the build results are tested using suites of unit, integration, scale and regression testing [12].

While CI testing is essential to identify and fix issues that could lead to build failures [2], to perform such CI in real-world industry settings is far from being a trivial task [20]. On the one hand, a large number of commits are produced daily [34]. Each such commit has to undergo the compilation and test activities, the latter can involve hundreds to thousands of tests to be run. Since all this CI activity has to be performed on all supported platforms, and a failure will eventually lead to a new commit for which, in turn, all compilation and test activity has to be repeated, the total build cost in terms of time, energy and resources for a given commit across all platforms and test suites has become challenging, even for large companies [13, 16, 49].

On the other hand, previous studies [24, 26, 27] suggest that some commits are more likely to lead to build failures compared to other commits. Commits that include simple and recurring types of code changes, such as adding comments, renaming files, and updating `README` files [2], tend not to need a thorough round of testing. Hence, such change-sets can mostly be exempted from CI testing (just undergoing less frequent nightly builds, for instance).

For other kinds of changes, various heuristics have been proposed in order to reduce the time and resources during software compilation and testing. Heuristics for skipping [26] and combining [35] builds focus on reducing the number of builds (i.e., compilation and testing) that will be scheduled, while test case prioritization and reduction [13] focus on speeding up the test activity within scheduled builds. In this study, we are particularly interested in the former heuristics for their build order-preserving characteristics. These heuristics can be grouped in two major categories: (1) *grouping*, which includes heuristics designed for grouping multiple builds together before forwarding the resulting batch to CI testing [6], instead of running each build separately; (2) *skipping heuristics*, which only schedules essential CI builds, i.e., builds that are more likely to fail.

Despite ongoing research on these two categories of heuristics, both types of heuristics still face important challenges. The frailties of the grouping heuristics include a larger turn-around-time for commit building and inefficiencies in identification of the root cause of failing builds. On the other hand, skipping heuristics [2, 24], instead, run the risk of skipping failing builds, which could leave a developer unaware of faults in their code. Since these two heuristics come from

two separate lines of research, they have not been empirically compared with each other or combined into more advanced heuristics combining the strengths of both.

In this paper, we hypothesize that the combination of grouping and skipping heuristics, i.e., incorporating build outcome prediction in commit grouping techniques, can reduce the number of commits for which actual build activity needs to be scheduled. We present a large-scale quantitative study on 79,482 builds from 20 open-source projects aimed to empirically assess our hybrid heuristics for reducing the number of builds made during CI compared to three state-of-the-art grouping heuristics [6] – i.e., BatchBisect, Batch4, and BatchStop4 – and one skipping heuristic, i.e., SmartBuildSkip [24]. In particular, we address the following three RQs:

- *RQ1: At the commit level, how do the baseline batching and skipping heuristics perform compared to each other?* - We compare SmartBuildSkip against the existing state-of-the-art commit grouping techniques and study the trade-off between the number of builds required and the median delays seen between the two methods. Based on the identified strengths and weaknesses of these heuristics, we propose two *hybrid heuristics* for reducing the number of builds during CI testing.
- *RQ2: How does the performance of ML-CI compare to the baseline heuristics?* - The first hybrid heuristic uses a machine learning (ML) model to predict build outcomes of incoming commits. This heuristic can skip unnecessary builds and apply commit grouping heuristics in cases where builds are expected to be made.
- *RQ3: How does the performance of the Timeout Rule compare to the baseline heuristics?* - This hybrid heuristic reduces the delays of the SmartBuildSkip algorithm by both necessitating a build at regular intervals and by using commit grouping heuristics to further reduce the builds made in CI.

We summarize below our major study contributions:

- Grouping heuristics can increase the number of builds required in the CI system by a median of 52.94%, in comparison to skipping heuristics.
- Skipping heuristics can induce a median delay of 11.5 builds in the identification of failing builds. After evaluating our two hybrid heuristics, we observed that they can significantly reduce the median delays introduced by skipping-based heuristics by 96%.
- ML-CI can reduce the median delays by 9.5 builds in comparison to skipping-based heuristics and maintains the low turn-around-time of grouping-based heuristics, but it also suffers from the latter heuristics' high percentage of builds required.
- The Timeout Rule also maintains a low turn-around-time of 1-2 commits along with reducing the number of required builds by a median of 26.10% in comparison to grouping-based heuristics.
- Our study's replication package[1] is available online.

We organized the remainder of this paper as follows. Section 2 discusses background information and related work. Section 3 and 6 detail the main heuristics

[1] `https://github.com/SAILResearch/replication-21-divya_kamath-build_avoiding_heuristics-code`

discussed in the paper. Sections 4 and 7 introduce our study methodology, including the study goal, research questions, and study steps. Sections 5, 8, and 9 present our study results per research question. Section 11 discusses threats to the study validity. Finally, Section 12 concludes this paper and suggests future work.

2 Background and Related Work

2.1 Study of build failures

The study of builds and build outcome prediction began in the early 2000s. Hassan et al. [18] used historical build results and project attributes to predict the certification result of a build. Further studies examined the factors affecting build failures and proposed algorithms to predict build outcomes. Kwan et al. [31] identified that higher social co-ordination in a project correlates with more build failures for certain build types. Cataldo et al. [8] studied the impact of project attributes and cross-project feature interactions on software integration failures. Similarly, Islam et al. [21] identified the relation between factors such as complexity, contribution models, and other project attributes and the build result. Those authors empirically analyzed data from TravisCI, Git and Github, providing evidence that build complexity – usually measured by the number of build commits – and source code churn in terms of changed lines of code or files [29] are negatively correlated with build failures.

Researchers also used an alternate approach of monitoring and sending preemptive alerts to developers when their changes can impact future build activities such as build duration and future build failures [42]. In parallel, Zhang et al. and Seo et al. [39, 47] analyzed the frequency of compiler errors that cause build failures and the corresponding fix times and fix patterns. Kerzazi et al. [29] also reviewed the circumstances that lead to build breakages and the effect of such failures on developers. The Culprit Finder algorithm presented by Zifti et al. [48] uses a combination of heuristics to rank commits according to their suspiciousness to find the root cause of a build failure. As root cause of a failure, Ghaleb et al. [17] studied the characteristics of CI builds that led to long duration of builds, like rerunning failed commands or not caching frequently used variables.

Xia et al. [45] analyzed 4 major build systems to study and categorize their bugs and fixes. Vassallo et al. [43] compared CI processes with the occurrences of build failures and derived a taxonomy of build failures. The impact of various factors on build failures were also studied by Jain et al. [22] who inspected the effects of a large team size on build failures. Rausch et al. [37], analyzed build failures from 14 OS Java projects and identified error categories and noisy build data, while Zolfagharinia et al. [49] studied the impact of OS changes and environments on build failures. Finally, Wu et al. [44] presented insights on failure frequencies and fix times in docker build data and studied the evolution of docker build failures over time.

While these studies reveal the characteristics of builds and the features that impact the outcome of a build, the main contribution of our study is to propose build avoidance heuristics that use decisions made by predictive models and other approaches to reduce the number of builds scheduled by CI. We evaluated our proposed heuristics on data collected from TravisCI and compare our findings to 4 state-of-the-art build reduction heuristics.

2.2 Build Outcome Prediction

In recent years, the study of build outcome prediction has gained momentum. Finley et al. [14] identified useful metrics for build outcome prediction and used the Hoeffding Tree classification method with adaptive sliding window to classify builds. Alternately, Hassan et al. [19] use abstract syntax tree-level code change modification data to predict the pass/fail outcome of a build. Other diverse solutions to build outcome prediction were introduced by Ni et al. [36] and Barrak et al. [3] who used cascaded classifiers and multivariate models, respectively, to predict build outcomes. Beller et al. [7] aimed to understand the importance of software testing in the CI process and identify the influence of the integration environment and project language on testing. This exploratory study suggests that the majority of failed tests occur due to failures in tests run during the builds.

In summary, the studies discussed above suggest that software process, build environment, and development team characteristics are major factors that influence the outcome of builds in CI system. Our study investigates other factors such as the time elapsed since previous scheduled builds and commit factors, which could be used to help engineers make the decision of whether or not to build a commit. Our proposed hybrid algorithms work on these factors to provide effort savings for CI builds.

2.3 Speeding up CI

In previous studies, researchers have reduced the effort required for CI builds according to two broad approaches: (1) by reducing the number of builds scheduled during CI and (2) by speeding up test suite execution within scheduled CI builds. Speeding up test suites covers a wide range of test case prioritization and test case selection techniques. In test case prioritisation techniques, the tests that have higher potential to fail are tested first, while in regression test case selection a minimal set of test cases that ensures a maximum code coverage is tested. The effectiveness of these techniques are explored extensively in literature [13,23,33,46].

Overall, there are three broad techniques introduced to reduce the volume of builds made during CI: (1) Static methods, which use a set of predefined rules to determine which commits are safe to be skipped, (2) Batching, which can build multiple commits at a time and identify culprit commits, and (3) Prediction models, which predict whether a build is likely to succeed or fail, and skip unnecessary builds.

Abdalkareem et al. [1] manually investigated 1,813 explicitly skipped CI commits and presented a rule-based technique to skip further commits. They also presented CI Skip and CI Run rules to automatically identify the commits that

are safe to skip [1]. They extracted 23 features from the historical data of 10 software repositories and identified that the number of developers and commit messages can help in deciding which commits are safe to skip along with the CI Skip rules.

Large companies like Google and Facebook that have a large number of incoming commits, typically attempt to reduce the number of builds made in their CI by grouping ("batching") commits together and scheduling builds one group at a time [6]. Such a batch grouping approach considers a build to be necessary only once for a group of sequential commits, thereby reducing the total number of builds. Nafaji et al., Beheshtian et al., and Bavand et al., [5, 6, 35] presented various batch grouping approaches that can group together multiple commits to reduce the number of builds required during CI. If such a batched build fails, each grouping approach uses a unique methodology to determine the culprit failing commits present in the batch.

Chen et al., [9] identified that different outcomes of previous builds have different impacts in development activities of the current build. They introduced BuildFast, which is a hybrid, adaptive algorithm that switches between 2 prediction models based on the outcome of the previous build. It uses a history-aware approach with specific features to predict build outcomes. Jin et al. [24, 25, 26, 27] presented several studies on skipping builds based on the predicted build outcome. They presented (1) SmartBuildSkip, which uses a Random Forest classifier on build and project features to predict build failures, (2) PreciseBuildSkip, which uses CI-Skip and CI-Run [2] rules to predict build failures, and (3) HybridBuildSkip, which combines both SmartBuildSkip and PreciseBuildSkip in a hybrid way. In a separate work [26], they evaluated 14 variants of 10 techniques of selection and prioritization strategies on build and test granularities. They presented a dataset and a collection of techniques to improve CI using build/test selection and prioritization.

In our study, we focus on reducing the number of scheduled builds by combining commit grouping techniques with the predicted outputs of models performing build outcome prediction. With the introduction of predictive models, our heuristics do not group all commits in a project to be built in group as the novel batch grouping techniques do. Instead, our prediction models choose which commits need to be built, then proceeds to build them in a batch. Sections 3 and 6 detail all the algorithms used in our study.

3 Baseline Heuristics

This section discusses in detail the two families of baseline heuristics that are studied in this paper: Grouping Heuristics and Skipping Heuristics. We present the studied algorithms of these two families in this section and compare their performance in Section 5.

3.1 Grouping Heuristics

In this paper, to limit our scope, we study commit grouping approaches that use batch bisection methods for identifying failing builds. Consequently, we exclude

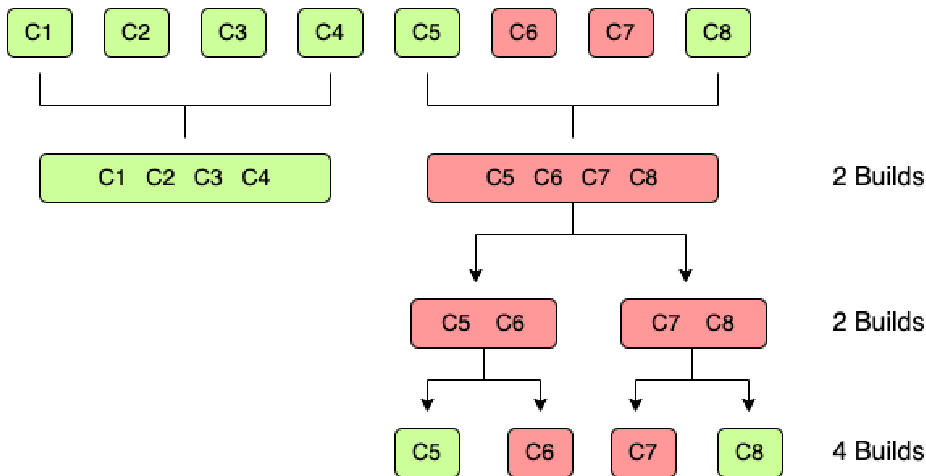


Fig. 1: This figure illustrates the number of builds required by BatchBisect commit grouping algorithm [6]. In the figure, red boxes signify failing builds and green boxes signify successful builds.

other approaches based on pool testing like Dorfman Medical Pool Testing and Double Pool Testing since they do not include bisection to identify culprit commits. Following the work of Bavand et al. [5], who state the Batch4 algorithm to be the best static batching algorithm, we choose the three state-of-the-art approaches discussed by Beheshtian et al. [6], i.e., Batch4, BatchStop4 and BatchBisect.

In each of these algorithms, a sequence of commits is grouped into batches based on the rules of that algorithm, before the entire batch is built once. If the batch build passes, the build has saved substantial amount of time and computing resources compared to running a separate build for each commit. However, if a batch build fails, subsequent builds are necessary to determine the commit(s) causing the failure. Each algorithm defines its own rules to identify those culprit commits.

Figure 1 illustrates running the BatchBisect algorithm (Algorithm 1) on a dataset of 8 commits, with a batch size of 4 commits. It basically groups each sequence of 4 commits into a batch, then runs one build for the entire batch. If we assume that the boxes in green represent commits with successful build outcomes[2] (passing builds) and the boxes in red represent commits with unsuccessful build outcomes (failing build), we can see why the first batch builds successfully, while the second batch fails.

Although commit grouping algorithms can reduce the number of builds made, if a grouped build fails (especially for projects with a failure rate less than 25% [35]), it can get more expensive to identify the culprit commit. For example, upon a failure, a technique like BatchBisect uses a bisection approach to systematically, and with as few extra builds as possible, detect the build failure culprits. In Figure 1, the initial, failing batch C5-C8 is divided into two smaller batches, both of which

[2] Of course, the algorithm does not know this information, only an approach that builds every single commit would.

Algorithm 1 BatchBisect

```
 1: procedure BATCHBISECT(batch, fails)
 2:     if len(batch) == 1 then
 3:         build_outcome = Build(batch)
 4:         if build_outcome == False then
 5:             fails.extend(batch)
 6:         end if
 7:         return
 8:     end if
 9:     outcome ← Build(batch)
10:     if outcome == False then
11:         half = len(batch)/2
12:         BatchBisect(batch[: half])
13:         BatchBisect(batch[half :])
14:     end if
15: end procedure
16: procedure FILLBATCH(batchSize, incomingCommit)
17:     i = 0
18:     batch = []
19:     fails = []
20:     while i < batchSize do
21:         batch.append(incomingCommit)
22:         i ← i + 1
23:     end while
24:     if i == batchSize then
25:         outcome ← BatchBisect(batch, fails)
26:     end if
27: end procedure
```

Algorithm 2 Batch4

```
 1: procedure BATCH4(batch, fails)
 2:     outcome ← Build(batch)
 3:
 4:     if outcome == False then
 5:         i ← 0
 6:         while i < batch_size do
 7:             build_outcome ← Build(batch[i])
 8:             if build_outcome == False then
 9:                 fails.extend(batch[i])
10:             end if
11:         end while
12:     end if
13: end procedure
14: procedure FILLBATCH(batchSize, incomingCommit)
15:     i = 0
16:     batch = []
17:     fails = []
18:     while i < batchSize do
19:         batch.append(incomingCommit)
20:         i ← i + 1
21:     end while
22:     if i == batchSize then
23:         outcome ← Batch4(batch, fails)
24:     end if
25: end procedure
```

Algorithm 3 BatchStop4

```
1: procedure BATCHSTOP4(batch,fails)
2:     if len(batch) <= 4 then
3:         i ← 0
4:         while i < batch_size do
5:             build_outcome = Build(batch[i])
6:             if build_outcome == False then
7:                 fails.extend(batch)
8:             end if
9:         end while
10:    end if
11:    outcome ← Build(batch)
12:    if outcome == False then
13:        half = len(batch)/2
14:        BatchStop4(batch[: half])
15:        BatchStop4(batch[half :])
16:    end if
17: end procedure
18: procedure FILLBATCH(batchSize, incomingCommit)
19:    i = 0
20:    batch = []
21:    fails = []
22:    while i < batchSize do
23:        batch.append(incomingCommit)
24:        i ← i + 1
25:    end while
26:    if i == batchSize then
27:        outcome ← BatchStop4(batch, fails)
28:    end if
29: end procedure
```

fail again. Another step of bisection results in batches of 1 commit, and learns that commits C6 and C7 both have issues. In this case, BatchBisect had to schedule 7 builds compared to the 4 builds of a naive approach that builds every individual commit.

The Batch4 algorithm (Algorithm 2) operates with a single batch size of 4 commits, and does not use bisection upon failure. Instead, if a batch fails, the Batch4 algorithm directly builds all individual commits in the batch to identify the culprit commit(s). The final batching algorithm, BatchStop4 (Algorithm 3) is a combination of BatchBisect and Batch4 –it bisects all commits until a batch size of 4 is reached, then performs Batch4. For instance, for a batch of 16 commits, the whole batch is built first. If there is an error, it splits into 2 batches of 8. If those fail too, it splits into four batches of 4. For each failing batch of 4, it will build each of its 4 commits individually.

In this paper, we make an effort to reduce the turn-around-time of a commit. This turn-around-time refers to the total duration from the time a developer pushes their commit to the CI server to the time the commit is integrated into the code base.

In comparison to traditional CI where each incoming commit is scheduled to build individually, the commit grouping techniques increase the turn-around-time for a commit. The turn-around-time of grouping heuristics includes the grouping-based delay, the build duration of the top-level batch and the bisection duration in case the top-level batch fails. From the time it is pushed to build, the commit

has to wait for the build queue to fill up with new incoming commits. The larger the batch size is, the larger the delay of each individual commit. For example, in a batch size of 4 commits, a median delay of 1.5 builds is incurred, where the first commit waits for 3 builds, second commit for 2 builds, and third commit for 1 build, until the final commit completes the batch and a build can be made. In this paper, we call this delay the “grouping-based delay”.

Apart from order-preserving batching heuristics, Behestian et al. [6] also discuss the RiskTopN and RiskBatch algorithms, which group commits into batches based on the risk associated with each commit. Due to this, the ordering of the commits may be affected, which may lead to inconsistent results due to merge conflicts. Since there is no significant difference between the performance of the order-preserving and order-obstructing algorithms, we have chosen to work with order-preserving commit grouping algorithm in this study, i.e., BatchBisect, Batch4 and BatchStop4.

3.2 Skipping Heuristics

In this paper, we limit ourselves to study one skipping-based algorithm to narrow down our scope. We explore the SmartBuildSkip algorithm introduced by Jin et al. [24], which uses a Random Forest Classifier based on build and project features such as the number of source lines of code, team size, number of files, number of test lines of code, etc., to predict the outcome of a build. We choose to study this algorithm as it is the first technique that does not use the previous build outcome or historical model predictions to predict the outcome of future builds, yielding a more realistic workflow. Our work (presented in Section 6) explores the effects of integrating ‘Timeout Values’ in SmartBuildSkip-based build prediction algorithms.

The algorithm (see Algorithm 4) works in two phases by distinguishing between the first and subsequent failures: a) prediction phase and b) deterministic phase. In the prediction phase, the algorithm predicts a commit’s CI outcome and uses the prediction to determine if a build should be run (failure prediction) or not (pass prediction). If a build is predicted to fail, and the build indeed does fail, the algorithm moves on to the next phase by assuming that all the subsequent builds are going to fail and hence forcing builds to be run (without using the prediction model) until a build pass is observed. Once a build pass is obtained, SmartBuildSkip goes back to the prediction phase.

Figure 2 illustrates the SmartBuildSkip algorithm on the same data set as Figure 1. Note that the colours again indicate the actual build outcome (if a given build would have been scheduled), while in this case the model’s prediction and actions are labelled accordingly.

Beginning with commits C1 and C2, the SmartBuildSkip algorithm predicts the builds to pass, hence skipping the build entirely. When it comes to commit C3, the algorithm predicts the build to fail, and hence schedules the build to identify the error. Yet, the build for commit C3 is successful, which prompts SmartBuild-Skip to continue predicting build outcomes by staying in its first execution phase. Similarly, commits C4 and C5 are predicted to pass and the build is skipped. When the algorithm predicts commit C6 to fail, a build is run, which reveals an unsuccessful build. This prompts the algorithm to switch to the deterministic phase

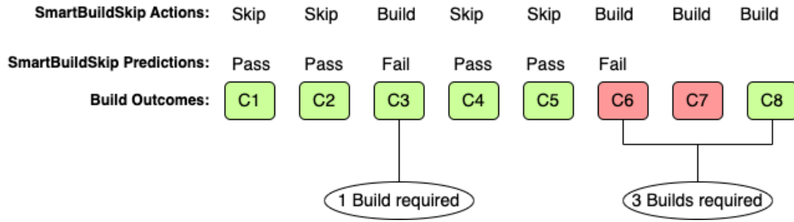


Fig. 2: This figure illustrates the number of builds required in the SmartBuildSkip algorithm. In the figure, red boxes signify failing builds and green boxes signify successful builds.

and it also builds commit C7. When this build fails, the algorithm continues to build commit C8, which runs successfully. Once this passing build is seen, SmartBuildSkip switches back to the prediction phase for the remaining commits. While BatchBisect required 1 build for C1-C4 and 7 builds for C5-C8, SmartBuildSkip required only 4 builds (C3 and C6-C8) for the whole set of commits.

Algorithm 4 SmartBuildSkip

```
 1: predictor ← SmartBuildSkip
 2: build_flag ← 0
 3: while True do
 4:     commit ← incoming_commit
 5:     if build_flag is True then
 6:         result ← Build(commit)
 7:         if result is True then
 8:             build_flag ← 0
 9:         end if
10:     else
11:         outcome ← predict(commit)
12:         if outcome is True then
13:             continue
14:         else
15:             result ← Build(commit)
16:             if result is False then
17:                 build_flag ← 1
18:             end if
19:         end if
20:     end if
21: end while
```

However, when running the SmartBuildSkip algorithm, a delay can be incurred in the identification of a build failure. This occurs because the model used to predict the build outcome could incorrectly predict a failing build to pass and potentially could do so for a long time. Since, due to the nature of the SmartBuildSkip algorithm, builds that are likely to pass are skipped, the true build failure is not caught until a future build failure is predicted and hence built for real.

Figure 3 further illustrates these delays, showing the total delay in a set of 14 commits from C1-C14. When a failing build is skipped, (for example, for C2, C7 and C9) the build failure and its root cause are unknown to the build engineer.

The algorithm continues its prediction phase until a future commit is predicted to fail (for example, for C4 and C10). When these commits are built, the previous build failures are also caught. In this paper, we refer to these delays as "skipping-based delays". Such delays are also discussed by Jin et al. [24], who try to address these by finding the optimal threshold for the prediction model's predicted failure probability, and also propose several rule-of-thumb techniques that only rely on basic rules comparing a single build feature to a threshold (as opposed to using a complex prediction model).

For skipping-based heuristics, the turn-around-time varies according to the actual build outcome of a given commit, as shown in Table 1. In case a pass is predicted, but the build actually fails, one incurs a skipping-based delay, while in case of failure predictions or during the deterministic phase, the turn-around-time equals the actual build duration. Only when a pass prediction was correct, no delay is incurred.

Table 1: Turn-around-time for skipping-based heuristics.

Prediction	**Actual Build Outcome**	
	Pass	**Fail**
Pass	0	Skipping-based delay
Fail	Build duration	Build duration
Deterministic Phase	Build duration	Build duration

While the large turn-around-time (especially the skipping-based delays) of SmartBuildSkip can be concerning, it also cannot accommodate cases where developers split up their work into multiple commits [30], causing some of the intermediate commits to fail. While SmartBuildSkip may flag each failure immediately and lead to unnecessary build executions, grouping heuristics are more likely to stitch the multiple intermediate commits together into one full build (batch) that can build successfully.

4 RQ1 Study Methodology

4.1 Data Set

The dataset for this paper is extracted from the replication package of the SmartBuildSkip algorithm [24]. In their study, Jin et al. [24], selected 359 projects out of the 1,359 projects having CI data available on Travis Torrent. They only selected projects that are more than one year old, with at least 200 builds and at least 1,000 lines of source code. Beheshtian et al. [6] evaluated the top 9 largest projects out of 1,200 available projects on Travis Torrent, selecting only active projects that have a failure rate of less than 25%.

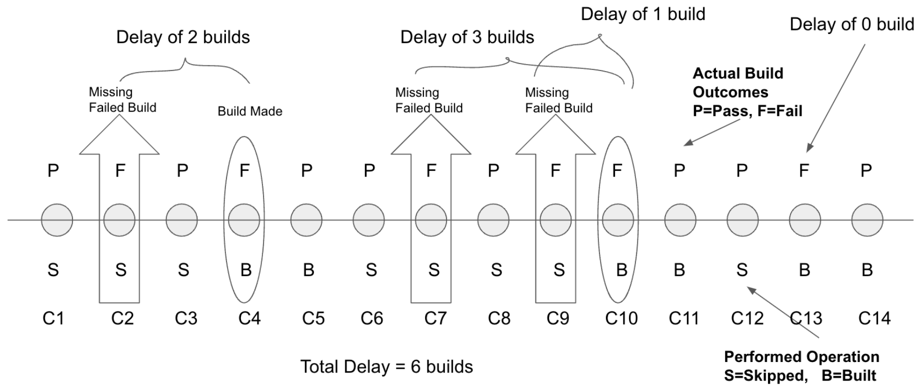


Fig. 3: This figure illustrates the delays incurred during the SmartBuildSkip algorithm.

In our study, from the 1,277 projects in the replication package of SmartBuildSkip [24], we select those projects that consist of at least 2,000 commits, in order to analyse the trends of the four models. We do not limit our selection based on the failure rate of the project in order to identify the ramifications of our approaches on all kinds of projects. Eventually, our study resulted in a dataset of 20 projects.

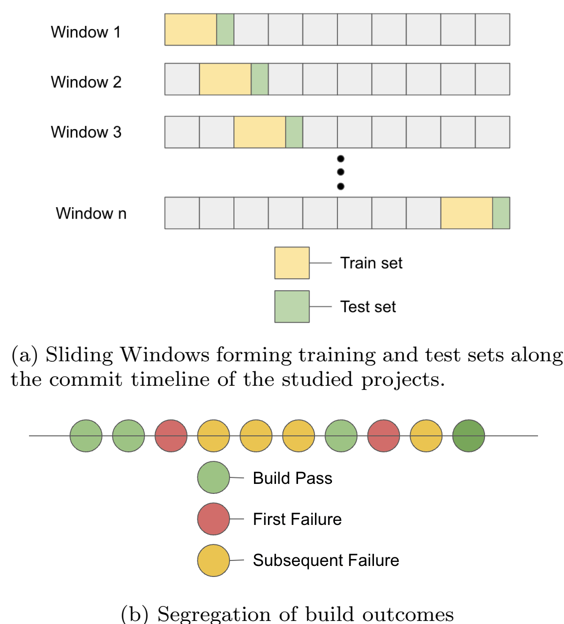
Table 2: Studied Software Projects (ordered by #commits)

Projects	**Size of Dataset** (number of commits)	**Distribution** (% of failed builds)
Rails	14,133	27.94
Jruby	8,275	55.79
Metasploit-framework	7,602	3.39
Cloudify	4,815	17.79
Vagrant	4,049	6.74
Rubinius	4,005	28.43
Open-build-service	3,718	21.70
Gradle	3,569	6.16
Sonarqube	3,163	14.73
Loomio	2,924	23.73
Fog	2,785	27.89
Opal	2,595	6.82
Cloud-controller-ng	2,399	18.71
Puppet	2,347	4.13
Concerto	2,301	8.30
Sufia	2,259	10.88
Geoserver	2,183	47.36
Orbeon-forms	2,166	46.49
Graylog2-server	2,110	2.93
Heroku	2,084	14.87

4.2 Evaluation Methodology

Our choice for at least 2,000 commits allows our study to chronologically segregate our data into training and test data sets of adequate sizes. Using projects of this minimum size will also allow to incorporate different real-world behaviours into the dataset like long continuous streaks of build failures, scattered build failures, etc. We perform a sliding window based strategy making a series of training and testing windows throughout the timeline of a project. We need to find an optimal window size to ensure that the size of the training data is large enough to incorporate both passing and failing commits in the window. By trial and error, we found the optimal window size for our evaluation approach to be obtained by dividing each project's dataset by 5. The dataset segregation procedure is depicted in Figure 4a.

For each window, we use 70% of the sliding window for training and use the remaining window data for testing. After completing the analysis on this window, we slide the window by half its length to get a new window. We repeat this process for the entire dataset incorporating the maximum data points possible. We exclude from our analysis any remaining data points that fall outside all windows.



(a) Sliding Windows forming training and test sets along the commit timeline of the studied projects.

(b) Segregation of build outcomes

Fig. 4: Training process and data segregation.

For each heuristic, we developed a simulator to apply the rules of the algorithms discussed in Section 3 chronologically on the data in a given test set. As a given heuristic is applied on the incoming commits, we record the number of builds made, builds skipped and unidentified build fails. We also record the number of consecutive predicted build passes for the timeout feature. At the end, for each heuristic, for each project, we compute the percentage of builds required along with the median delays induced by each heuristic.

4.3 RQ1 Analysis Procedure

This question compares the existing prediction-based algorithm SmartBuildSkip with the commit grouping algorithms proposed by Beheshtian et al. [6]. We evaluate each heuristic in terms of two performance metrics.

The first performance metric is the Percentage of Builds Required. The lower this value, the better, as it indicates that more builds were skipped. We compare this metric between both baseline heuristics as well as TestAll, i.e., a simple baseline building each commit individually [6]. We measure the percentage of builds required for all the heuristics by calculating the total number of builds made and skipped for each project.

The second performance measure is Delay. For SmartBuildSkip, we measure the skipping based delays, whereas we measure grouping based delays for the commit grouping algorithms. We measure the median delay of a given heuristic across all commits of a given project, and, for a given heuristic, the median of these median delays across all projects.

Beheshtian et al. [6] identified the best batch sizes to be used with the BatchBisect algorithm as being between 4 and 8 builds, whereas batch sizes of 2, 4 and 8 builds gave major savings with BatchStop4. To that extent, we evaluate all the heuristics in this paper with batch sizes 2, 4 and 8. To further evaluate the effect of larger batch sizes on all heuristics, we also evaluate the batch size 16.

We study the percentage of builds and median delays required by all 4 baseline heuristics, namely SmartBuildSkip, Batch4, BatchBisect and BatchStop4. In addition, we also analyze the number of bad skips made by SmartBuildSkip to study the robustness of the latter approach.

5 Performance Results of Baseline Heuristics (RQ1)

Table 3 shows the percentage of builds required by the baseline heuristics. For the heuristics BatchBisect and BatchStop4, the batch size that provides the best performance is listed along with the results achieved with it.

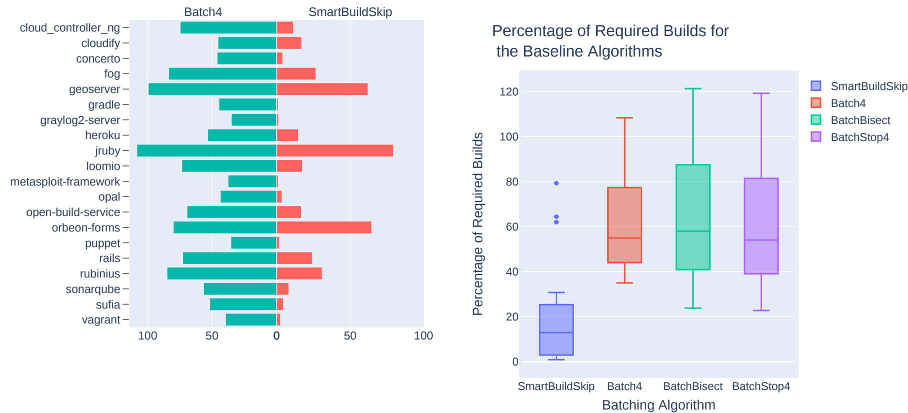
Compared to the best batching approach (i.e., Batch4), SmartBuildSkip requires only a median of 12.92% builds to be made compared to 54.97%, saving a median of 41.06% builds. Figure 5a plots the percentage of builds required for the 20 projects by the Batch4 and SmartBuildSkip algorithms, while Figure 5b summarizes the results for the four baseline approaches. For BatchBisect and BatchStop4, the plots show the results for the batch sizes with the lowest percentage of builds required.

Project	Algorithm					
	SmartBuildSkip	Batch4	BatchStop4		BatchBisect	
	% of Builds Required	% of Builds Required	% of Builds Required	Batch Size	% of Builds Required	Batch Size
cloud-controller-ng	11.265	74.691	80.401	4	74.691	4
cloudify	16.955	45.252	42.868	16	38.639	16
concerto	3.865	45.974	44.122	8	42.19	8
fog	26.587	83.73	93.915	2	83.73	4
geoserver	62.002	99.491	108.227	2	99.491	4
gradle	0.883	44.548	40.966	8	39.512	8
graylog2-server	1.225	34.996	23.797	16	22.747	16
heroku	14.578	53.333	55.644	8	51.378	8
jruby	79.32	108.406	121.373	2	108.406	4
loomio	17.298	73.485	82.828	4	73.485	4
metasploit-framework	1.023	37.476	28.972	16	28.436	16
opal	3.49	43.519	40.883	8	39.459	8
open-build-service	16.542	69.357	76.333	4	69.357	4
orbeon-forms	64.444	80.085	96.838	2	80.085	4
puppet	1.734	35.303	25.138	16	24.507	16
rails	24.109	72.799	82.966	4	72.799	4
rubinius	30.751	84.878	92.07	2	84.878	4
sonarqube	8.187	56.608	60.234	4	56.608	4
sufia	4.412	51.797	53.513	8	49.755	8
vagrant	2.378	39.552	33.836	16	30.818	16
Median	12.922	54.971	57.939	6.0	53.993	6.0

Table 3: Percentage of Builds Required by Individual Heuristic

The low percentage for SmartBuildSkip in all 20 projects confirms that it can substantially reduce the resources consumed during CI. At least 25% of the projects require only 41% builds using the BatchBisect algorithm, while the Batch(Stop)4 algorithms require up to 80% of builds for at least 75% of the projects. Except for three outliers (i.e., Geoserver, JRuby, Orbeon-forms), SmartBuildSkip requires less than 31% builds to be made for all projects.

The optimal batch size is 8 builds for 75% of the projects using BatchStop4 and for 20% of the projects using BatchBisect. In some projects, better performance can be achieved by simply increasing or decreasing the batch size to 2, 4 or 16. By increasing the batch size, we reduce the number of builds required to be made for a given test dataset. However, if more builds in the test dataset are failing, the batching algorithms make more builds in order to find the culprit commit(s). Hence, batching algorithms can improve or reduce performance depending on the failure rate, i.e., the percentage of build failures in a project: if there are more failing builds than passing builds in a project, a smaller batch size is more effective, like for BatchBisect in Fog, Rubinius etc. If there are more passing than failing builds, larger batch sizes are more effective, like for BatchStop4 and BatchBisect using a batch size of 16 in Vagrant, Graylog2-server,



(a) Percentage of builds required for the 20 projects by SmartBuildSkip (SBS) and Batch4.

(b) Boxplot of the Percentage of Builds Required for each Baseline Algorithm.

Fig. 5: Builds Required by Baseline Algorithms.

Puppet etc. For similar reasons, Nafaji et al. [35] and Beheshtian et al. [6] recommend using batching algorithms with projects having less than 25% failure rate.

The median delay of SmartBuildSkip varies from 2 to 80 commits. As shown in Figure 6, the minimum median delay is made by project Orbeon-Forms (2 commits) while the maximum median delay (80 commits) is made by project Metasploit-Framework, for a median value of 11.5 builds across all projects. This means that if SmartBuildSkip misses to catch a build failure in Metasploit-Framework, it can take a median of 80 commits before the build failure is caught. This number varies for every project.

The drastic shift in performance from Orbeon-Forms to Metasploit-Framework may be due to the sensitivity of the SmartBuildSkip model. Looking at Table 2, Orbeon-Forms has a 46.49% failure rate, whereas Metasploit-Framework has a low failure rate of 3.29%. This means that the model trained on Metasploit-Framework has not seen many examples of build failures, which can lead to inadequate training of the SmartBuildSkip model. Subsequently, SmartBuildSkip is not able to identify build failures effectively.

Alternately, builds are scheduled less frequently due to the predictive nature of the SmartBuildSkip algorithm. The more builds are skipped by the algorithm, the more out of touch it becomes with the actual outcomes, had the builds been scheduled. Since it has no knowledge of previous build outcomes, SmartBuildSkip will not be able to recognise an actual build failure that it incorrectly predicted to pass. This build failure is only revealed when a future build is scheduled due to a predicted failure. If builds were scheduled more frequently, build failures would have been identified earlier. We proposed our Timeout Rule to combat this shortcoming of SmartBuildSkip by issuing frequent builds, governed by a timeout mechanism.

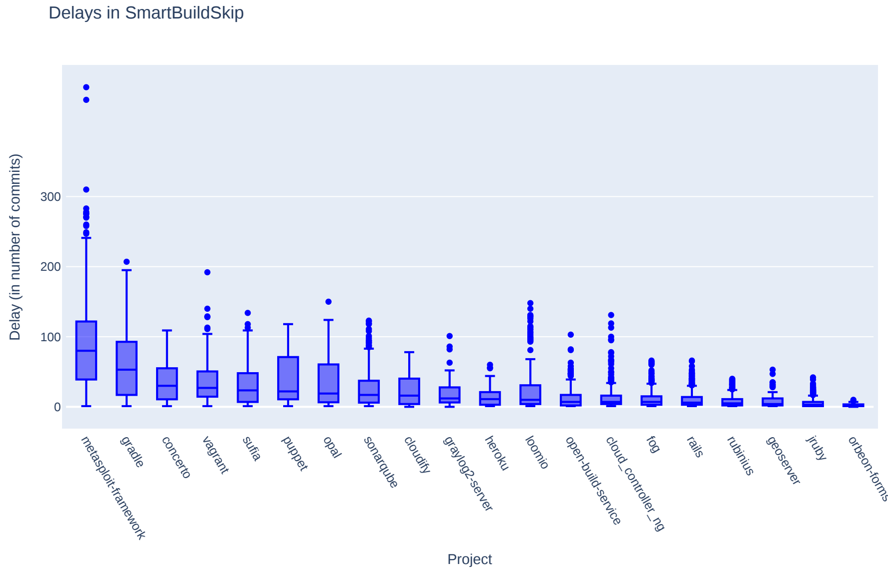


Fig. 6: Boxplots of delays made per project by SmartBuildSkip algorithm.

Summary of RQ1: *While SmartBuildSkip can save a median of 41.06% builds more than Commit Grouping, it introduces median delays in the identification of failed builds varying from 2 to 80 commits due to the low frequency of scheduled builds.*

6 Hybrid Heuristics

Given the trade-offs in benefits and drawbacks of commit grouping and skipping heuristics, we propose two hybrid heuristics (combining both families of heuristics) that try to maximize the benefits of both heuristics, while minimizing the median delays. The first hybrid heuristic, ML-CI, uses prediction models, while the second one, Timeout Rule, is rule-based (inspired by the basic rules of Jin et al. [24]).

6.1 ML-CI Model

To overcome the (1) potential sensitivity of SmartBuildSkip to failing intermediate commits and potential skipping-based delays in observing failures, and the (2) larger number of builds required by commit grouping algorithms upon a failing batch build, we combine the two heuristics together into a hybrid heuristic that aims to reduce the number of builds scheduled in the second phase of SmartBuildSkip, as well as the median delays.

Instead of immediately scheduling a build to be run upon a predicted build failure, the second phase of SmartBuildSkip would wait until a batch of commits

Table 4: Commit-level Metrics

Metric	Description	Usage
patch_size	Size of the commit in terms of number of source lines of code	The larger the size of the commit, the more likely it is to contain bugs
test_churn	Number of lines modified in test files	Test files are less likely to contain bugs than source code
freq_file_churn	Number of times a source code file is modified	If a file is changed often, its possibility of being error-prone is higher
files_churn (added)	Number of files added in the pull request(PR)	If more files are interacted by the PR, it is better to be rigorously tested
files_churn (deleted)	The number of files deleted in the pull request(PR)	If more files are interacted by the PR, it is better to be rigorously tested
timeout	Number of commits skipped since the last build	If last build was scheduled too long ago, it is better to schedule build to make sure everything is working well
num_commit	Number of commits in the PR	The more commits in a PR, the more need for testing

has arrived before building the batch. By doing a batch build, the likelihood of a build failing due to a commit being an intermediate commit would drop, as the other commits in the batch might fix the incomplete behaviour. Hence, the likelihood of a passing build would increase, reducing the number of builds that actually would be run. The main risk of this hybrid approach would be that a correctly predicted build failure would risk having to wait for the group build to start before finding the culprit.

In order to deal with the second problem of SmartBuildSkip, i.e., the issue that build failures could take a longer time to be spotted if the prediction model does not predict any failure (an issue potentially made worse due to the use of batching), we expanded SmartBuildSkip's prediction model with a timeout feature that simply keeps track of how many consecutive build passes have been predicted by the model before the current commit. The higher this timeout, the more out-of-sync our knowledge about the build status is with reality, and the higher the likelihood that a build failure has been missed.

While SmartBuildSkip's model explicitly was limited to features that do not rely on knowledge of previous build runs (because those builds might not have been run at all), information about previous predictions of the model itself would be readily available, assuming we take care of the case when model predictions are not used, i.e., the second phase of SmartBuildSkip.

Algorithm 5 shows ML-CI's integration of batch building within SmartBuildSkip. While the overall operation of SmartBuildSkip is maintained (modulo the metrics and timeout feature), ML-CI will group commits and build a batch, instead of individual commit builds, when a commit is predicted to fail. For example, consider C to be the set of commits $\{c_1, c_2, c_3, \ldots, c_n\}$ where n is the number of commits in the CI system. Also consider that we are using the BatchStop4 commit grouping algorithm with a BatchSize of B. As in the case of SmartBuildSkip, if the commit c_i is predicted to pass, the model does not spend time building the commit. However, if c_i is predicted to fail, then all commits from c_i to c_{i+B-1} are grouped before being built together. If the group build passes, we go back to

the prediction phase for commit c_{i+B}, however, if the group build fails, the rules of the commit grouping algorithm are applied. This process is repeated until a group build passes. We count a finished batch build as a timeout of "1" on line 13, because we have successfully seen one passing build.

Algorithm 5 ML-CI Model

```
1: predictor ← RandomForestModel
2: build_queue ← []
3: num_passes ← 0
4: Grouping ← False
5: while True do
6:     commit ← incoming_commit
7:     if Grouping is True then
8:         build_queue.append(commit)
9:         if len(build_queue) is batch_size then
10:            BatchStop4(build_queue)                    ▷ See Algorithm 3
11:            Clear(build_queue)
12:            Grouping ← False
13:            num_passes ← 1
14:        end if
15:    else
16:        outcome ← predict(commit ∪ num_passes)
17:        if outcome is True then
18:            num_passes ← num_passes + 1
19:            continue
20:        else
21:            num_passes ← 0
22:            build_queue.append(commit)
23:            Grouping ← True
24:        end if
25:    end if
26: end while
```

By introducing build prediction in a commit grouping algorithm, we can reduce the number of groups required to be built. Similarly, for sensitive prediction algorithms that perform many unnecessary builds, we reduce the number of builds actually scheduled by grouping builds together. However, as a drawback, the ML-CI model suffers from both grouping-based and skipping-based delays due to the combination of grouping and skipping heuristics. With larger batch sizes, the degree of grouping-based delay becomes higher, whereas smaller batch sizes (and the use of the timeout feature in the prediction model) might find an optimum between avoiding spurious builds while not incurring too high skipping delays.

6.2 Timeout Rule

In the same vein as Jin et al. [24], we also propose a simpler rule-based approach, in this case focusing explicitly on skipping-based delays of SmartBuildSkip, i.e., the risk that a failure is predicted a long time after an actual failing commit occurred. Our hybrid heuristic again exploits a timeout metric, but this time as a measure of confidence in the SmartBuildSkip model's predictions. The timeout metric is used external to the SmartBuildSkip model (i.e., not as a feature of the model)

to count the number of consecutive builds skipped by it. In essence, this heuristic not only schedules a build when SmartBuildSkip's model predicts a failure, but also when the model has predicted too many build passes consecutively, i.e., when the timeout metric exceeds a timeout threshold.

In other words, while the model would not consider a build necessary, the timeout mechanism adds a safety check able to avoid long stretches of predicted build passes without scheduled build. When the timeout threshold is reached, we again use batching to possibly reduce the number of builds to actually run. Similar to ML-CI, we incorporate commit grouping in the Timeout Rule to deal with transient build failures that might be fixed by a quick follow-up commit. By waiting for a batch of builds instead of immediately scheduling a build, the likelihood of such transient failures requiring additional builds can more likely be reduced.

Build Time-out = 3
Batch Size = 4

Actual Build Outcomes	SmartBuildSkip Predictions		Success Streak Rule Action	Delay Incurred
Pass	Pass	C1	Skip, timeout = 1	---
Pass	Pass	C2	Skip, timeout = 2	---
Fail	Pass	C3	Skip, timeout = 3	1 (Skipping-based)
Pass	Pass	C4	C4-C7 grouped together to build	3 (Batching-based)
Pass	Pass	C5		2 (Batching-based)
Pass	Pass	C6		1 (Batching-based)
Pass	Pass	C7		0 (Batching-based)
Pass	Pass	C8	Skip, timeout = 1	---

Fig. 7: Illustration of the Timeout Rule meta-heuristic in a worst case scenario for SmartBuildSkip (no failure predictions). In the figure, red boxes signify failing builds, green boxes signify successful builds and yellow boxes signify the grouped builds.

Algorithm 6 shows the Timeout Rule, and is illustrated in Figure 7, which shows a worst-case scenario when no build failure is predicted by SmartBuildSkip, hence SmartBuildSkip itself would not schedule any build to be run for a long time. When the build time-out threshold is reached at commit c_i (i.e., c_4), the prediction of SmartBuildSkip is ignored and instead the commits from c_i to c_{i+B-1} are grouped and built together (with the value of the build time-out reset to 1). If the batch truly fails, the subsequent steps of the batching algorithm are applied to find the culprit commits. Note that on lines 11 and 19 *Grouping* is set to *False* if the batch build is successful, otherwise to *True*. Grouping is a Flag that shows when to start grouping commits together. If true, we know we have to start grouping; if false, we have to continue with processing individual commits.

Algorithm 6 Timeout Rule

```
 1: predictor ← SmartBuildSkip
 2: build_queue ← []
 3: timeout ← 0
 4: Grouping ← False
 5: while True do
 6:     commit ← incoming_commit
 7:     if timeout > threshold then
 8:         build_queue.append(commit)
 9:         if len(build_queue) is batch_size then
10:             (batch_outcome) ← Build(build_queue)
11:             Grouping ← (batch_outcome)?False : True
12:             timeout ← 1
13:         end if
14:     else
15:         if Grouping is True then
16:             build_queue.append(commit)
17:             if len(build_queue) is batch_size then
18:                 (batch_outcome) ← Build(build_queue)
19:                 Grouping ← (batch_outcome)?False : True
20:                 timeout ← 1
21:             end if
22:         else
23:             outcome ← predict(commit)
24:             if outcome is True then
25:                 timeout ← timeout + 1
26:                 continue
27:             else
28:                 Clear(build_queue)
29:                 build_queue.append(commit)
30:                 Grouping ← True
31:             end if
32:         end if
33:     end if
34: end while
```

By introducing the Timeout Rule, we aim to reap the benefits of SmartBuild-Skip while reducing the skipping-based delay. Although the total number of builds is reduced, a build is made periodically to ensure that large delays are not induced. The value of the confidence threshold can be set based on the requirements of an organization. If the project has historically had a larger failure rate, setting a smaller confidence threshold will ensure that the builds are enforced more frequently. On the other hand, if a project has a lower failure rate, larger confidence thresholds can be used to perform period builds and ensure that there are no undetected build failures.

Of course, if the timeout threshold is too long, skipping-based delays can still be incurred. Those skipping-based delays, however, will only reach the maximum threshold of $max(timeout, batchSize)$. Additionally, the rule also introduces grouping-based delays. When a confidence threshold is reached or a build is predicted to fail, grouping-based delays are incurred as the commits wait for the build queue to fill.

7 Study Design

7.1 Study Goal and Research Questions

In order to validate the performance of the two hybrid heuristics compared to the two baseline heuristics, we perform an empirical study. We used the Goal/Question/Metric template [4] to describe our study goal as follows: **analyze** the performance of the hybrid heuristics designed for reducing the number of builds during Continuous Integration (CI) testing; **for the purpose of** 1) identifying advantages and drawbacks of the existing grouping and skipping heuristics and 2) evaluating the performance of hybrid techniques that combine both categories of heuristics; **with respect to** the percentage of builds required in comparison to building all commits and to the delay of observing build failures; **from the point of view of** software engineering researchers with expertise in mining software repositories and CI testing; and **in the context of** state-of-the-art heuristics and open source projects collected from a Travis CI data set shared by recent work [24].

In particular, we evaluate the two hybrid heuristics introduced in Section 6, i.e., ML-CI and Timeout Rule, and compare them to the Grouping and Skipping Heuristics in terms of the following research questions:

RQ2: *How does the performance of ML-CI compare to the baseline heuristics?* – ML-CI performs commit grouping in the second phase of SmartBuildSkip, while adding a timeout feature in the first phase model. We aim to identify the practicality of such a hybrid heuristic and identify its strengths and weaknesses in comparison to the baselines. We also evaluate the performance of ML-CI with varying hyper-parameters such as batch size and the choice of commit grouping algorithm.

RQ3: *How does the performance of the Timeout Rule compare to the baseline heuristics?* – The Timeout rule forces SmartBuildSkip to schedule builds once a timeout threshold is exceeded, then uses commit grouping in the second phase of SmartBuildSkip. Along with comparison of this hybrid heuristic to the baselines, this research question also aims to evaluate its performance with varying hyper-parameters such as batch size and commit grouping algorithm.

The SmartBuildSkip algorithm recognises that many build failures occur consecutively after another build failure. To distinguish the characteristics of the first build failure from the subsequent build failures, Jin et al. [24] used the data of build passes and first build failures to train their model. Similarly, we segregate the dataset into build passes, first failures and subsequent failures, then discard the latter when training our models in both of our meta-heuristics, i.e., the Timeout Rule and ML-CI. This segregation is depicted in Figure 4b.

7.2 RQ2 Analysis Procedure

As discussed in Section 6.1, ML-CI is a machine learning Random Forest Model trained on the commit metrics defined in Table 4, including a new time-out metric, to perform a prediction of the pass/fail outcome of the incoming commit under analysis. If a failure is predicted, we use commit grouping algorithms.

By varying the batching algorithm used with ML-CI between Batch4, BatchBisect and BatchStop4, we come up with three variants of ML-CI. We evaluated

the BatchBisect ML-CI variant with a batch size of 2, 4, 8, 16, and evaluated the BatchStop4 variant with the batch sizes 8 and 16. We omit the use of batch size 4 for the BatchStop4 variant as it is identical to the Batch4 variant. After comparing the three variants, we will compare the best one to SmartBuildSkip and its corresponding baseline batching algorithm.

As ML-CI integrates commit grouping approaches with SmartBuildSkip, the ML-CI hybrid algorithm inherits both skipping and grouping delays. Some failing commits that are not recognized by the prediction model cause skipping-based delays whereas the failing commits that are recognised and built can cause grouping-based delays. In other cases, the passing commits that are skipped by ML-CI cause zero delays, whereas passing commits that are incorrectly predicted to fail will cause grouping-based delays. To measure the Delay of ML-CI, we calculate the combined delays, i.e., for a given commit we select either the skipping-based delay, grouping-based delay or zero, whichever applies (note that only one applies to each commit).

We further apply statistical tests to determine whether the performance of ML-CI variants and baseline heuristics is similar or different. For the 4 heuristics (three ML-CI variants and one baseline heuristic), we evaluate the statistical difference separately for percentage of builds required and for median delays. To do so, we use the Kruskal-Wallis test supported by the Pairwise Wilcoxon test as a post-hoc measure at a confidence interval of 95% (alpha=0.05). If two heuristics are found to be statistically different, we further evaluate the effect size to gauge the magnitude of their difference.

7.3 RQ3 Analysis Procedure

The Timeout rule discussed in Section 6.2 essentially uses the SmartBuildSkip model to predict build outcomes, yet (1) forces a build once a set number of builds has been skipped ("timeout") and (2) uses build grouping instead of direct execution of a build.

To evaluate this approach, we use timeout values in the range [2, 20]. As with ML-CI, we also vary the batching algorithm used with the Timeout Rule to get its three variants then use the best one to compare with baseline heuristics. Similarly, we evaluated the BatchBisect Timeout Rule variant with a batch size of 2, 4, 8, 16, and evaluated the BatchStop4 variant with the batch sizes 8, 16. To simplify our observations, we only report the results of the timeout value that gives the best results (i.e., minimum builds required and minimum median delay). We do not perform further analysis of variations of timeout values as we use the best values for each project and heuristic. For example, the best timeout value to be used with project Geoserver while using the TR-Batch4 variant is 2, on the other hand, the best timeout value with to be used with project Rails for the same Timeout Rule variant is 20.

While all batching delays are grouping-based, and all SmartBuildSkip delays are skipping-based, our Timeout rule could cause either skipping or grouping delays. Similar to ML-CI, we measure the combined delay of Skipping and Grouping-based delays together along with the statistical difference between the 3 variants of the Timeout Rule and the baseline heuristics.

Table 5: Variants of ML-CI

Grouping Heuristic	Batch Size Used	Obtained Variant
Batch4	4	*MLCI-Batch4*
BatchBisect	1	*MLCI-BatchBisect-1*
	2	*MLCI-BatchBisect-2*
	4	*MLCI-BatchBisect-4*
	8	*MLCI-BatchBisect-8*
	16	*MLCI-BatchBisect-16*
BatchStop4	8	*MLCI-BatchStop4-8*
	16	*MLCI-BatchBisect-16*

8 Performance Results for ML-CI (RQ2)

In the following sections, we discuss the performance of ML-CI with respect to the baseline algorithms, i.e., SmartBuildSkip and the Batching Algorithms. Section 8.1 evaluates the 3 variants of the ML-CI model. Sections 8.2.2 and 8.2.3 compare the results of ML-CI with SmartBuildSkip and Batch4, respectively.

8.1 Comparing the variants of ML-CI

8.1.1 Approach

We derive the 3 main variants of ML-CI that are listed in Table 5, obtained by varying the batching algorithm used between Batch4, BatchBisect and BatchStop4. We further obtain 8 sub-variants by varying the batch size used for each batching algorithm. Table 6 lists the results of the percentage of builds required and the median delay observed for the 20 studied projects, for each variant of the ML-CI model. For the Batch4 column in the table, each entry is obtained by using a batch size of 4 builds. For the BatchBisect column, the batch size can vary between [1, 2, 4, 8, 16] whereas for the BatchStop4 column, the batch size can be 8 or 16 builds. Figure 8 depicts the box plots of Percentage of Builds required, median combined delays and best batch sizes used by the ML-CI variants across 20 projects.

Using two Kruskal-Wallis tests, we statistically compare the performance (percentage of builds required and median delay) of the three main variants of ML-CI obtained with the best performing batch size as listed in Table 6. We further investigated any statistical differences reported by Kruskal-Wallis using the Pairwise Wilcoxon post-hoc test. We measure the effect size using the eta-squared statistic [41] and interpret it accordingly: $0.01-0.06$ (small effect), $0.06-0.14$ (moderate effect) and $>= 0.14$ (large effect).

8.1.2 Results

For ML-CI, the median percentage of builds required is 45.84% for MLCI-Batch4, 50.13% for MLCI-BatchStop4, and 43.15% for MLCI-BatchBisect, although this performance is not consistent across projects. The Kruskal-Wallis test performed at a confidence level $\alpha = 0.05$, revealed that

Table 6: Results of ML-CI

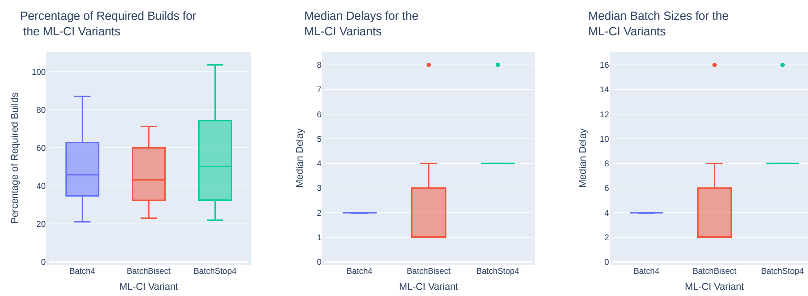
Project	ML-CI								
	MLCI-Batch4			MLCI-BatchBisect			MLCI-BatchStop4		
	% Builds Required	Median Delay	Batch Size	% Builds Required	Median Delay	Batch Size	% Builds Required	Median Delay	Batch Size
cloud-controller-ng	62.27	2	4	55.56	1	2	74.61	4	8
cloudify	21.11	2	4	23.95	2	4	28.49	4	8
concerto	39.45	2	4	40.42	4	8	38.49	4	8
fog	69.78	2	4	64.09	1	2	86.31	4	8
geoserver	87.02	2	4	71.25	1	2	103.39	4	8
gradle	37.59	2	4	38.63	2	4	37.85	4	8
graylog2-server	30.71	2	4	23.01	8	16	21.96	8	16
heroku	36.27	2	4	38.13	1	2	42.31	4	8
jruby	73.06	2	4	62.62	1	2	103.6	4	8
loomio	54.1	2	4	57.26	1	2	73.86	4	8
metasploit-framework	34.04	2	4	28.53	8	16	28.0	8	16
opal	35.4	2	4	36.68	2	4	36.54	4	8
open-build-service	51.97	2	4	45.89	1	2	60.49	4	8
orbeon-forms	63.33	2	4	67.26	1	2	73.93	4	8
puppet	31.21	2	4	23.48	4	8	23.01	4	8
rails	44.47	2	4	36.29	1	2	64.09	4	8
rubinius	74.0	2	4	63.12	1	2	88.15	4	8
sonarqube	48.36	2	4	51.17	1	2	52.4	4	8
sufia	47.22	2	4	49.84	2	4	47.88	4	8
vagrant	26.43	2	4	28.44	4	8	25.51	4	8
Median	45.846	2.0		43.154	1.0		50.137	4.0	

Table 7: Comparing Median Delays of ML-CI variants with each other

	Test Used	p-values		
		MLCI-Batch4	MLCI-BatchBisect	MLCI-BatchStop4
	Kruskal-Wallis	**1.523e-07**		
MLCI-Batch4	Pairwise Wilcoxon	–	0.0654	**3.3e-09**
MLCI-BatchBisect	Pairwise Wilcoxon	–	–	**0.0001**

there is no statistical difference ($p - value = 0.5624$) in the median percentage of builds required amongst the 3 variants. The three variants of ML-CI consistently demonstrate their best performance, i.e., the least percentage of builds required, in projects like Graylog2-server, Cloudify, Puppet and Vagrant, while they demonstrate their least effective performances in projects like Geoserver, JRuby, Rubinius and Fog.

We formulate two hypotheses explaining these findings of ML-CI. Based on the metadata of the dataset projects, we can hypothesize that dataset distribution



(a) Boxplot of Required builds in ML-CI variants using the 3 Batching algorithms.

(b) Boxplot of Median Delays in ML-CI variants using the 3 Batching algorithms.

(c) Boxplot of Batch Sizes used in ML-CI variants using the 3 Batching algorithms.

Fig. 8: Boxplots demonstrating the range of Percentage Build Required, Median Delays and the best performing batch sizes in ML-CI.

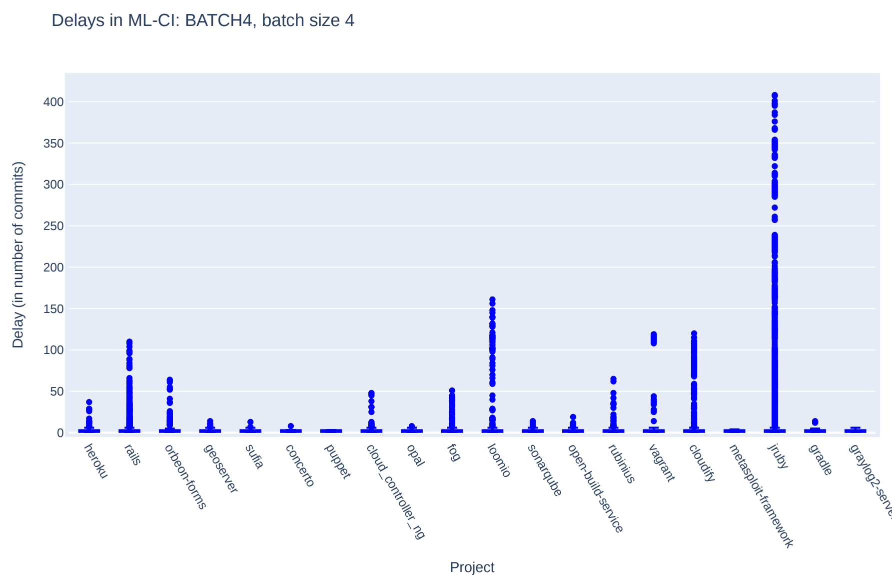


Fig. 9: Boxplots of delays made per project by the ML-CI Batch4 variant.

can be a factor directly related to the performance of the ML-CI model. More balanced datasets seem to require more builds for ML-CI compared to datasets that have a lower distribution of failing builds. For example, based on Table 2, we see that many projects such as Graylog2-server, Sufia, Puppet etc., record more passing builds than failing builds. Therefore, the Random Forest Model of ML-CI that trains on these projects' data points, is more likely to predict their builds to

Table 8: Comparing Percentage of Builds Required by ML-CI variants amongst each other

	Test Used	p-values		
		MLCI-Batch4	MLCI-BatchBisect	MLCI-BatchStop4
	Kruskal-Wallis	0.5624		
MLCI-Batch4	Pairwise Wilcoxon	Not needed, since Kruskal-Wallis test shows no significant difference		
MLCI-BatchBisect	Pairwise Wilcoxon			

pass. However, projects like Geoserver and JRuby contain a similar distribution of passing and failing builds. Hence, the models trained on these data points are less biased and do not as frequently predict build passes. This pattern can also be seen in the data of low-failure ratio datasets like Metasploit-Frameworks, Opal, Gradle and Puppet, and high-failure ratio datasets like Orbeon-forms, Loomio, Fog and Rubinius.

Another explanation could be the distribution of the failed builds, i.e., whether build failures are seen to occur consecutively or are singularly distributed throughout the dataset. If build failures occur close to one another, they are likely to be grouped together in a single batch, with the batching algorithms requiring more builds to identify these multiple build fails in the batch (due to bisection). However, if a single build failure is to be detected in a batch, bisection can reveal the culprit commit and the batching algorithms can detect them with less builds comparatively. This hypothesis can explain the behaviour behind moderate failure datasets such as Rails and Cloud-Controller-Ng requiring more builds to be made.

ML-CI exhibits median delays of 1 build for the BatchBisect variant, 2 builds for the Batch4 variant and 4 builds for the BatchStop4 variant. From Table 6, we see that the batch size that provides the best performance with BatchBisect is 2, whereas batch size 8 provides the best results with BatchStop4. Every time a batch of size 8 is filled, a median delay of 3.5 builds is incurred, while it only causes a delay of 0.5 build when a batch of size 2 is made. Similarly, a batch of size 4 incurs a median delay of 1.5 builds. Hence, the median delay incurred by BatchBisect is less than the median delay incurred by Batch4 and BatchStop4. Although large batch sizes may reduce the number of builds made, they also increase the waiting time for builds in the queue, which can reduce the efficiency of the developers in a project. Essentially, ML-CI creates a trade-off between build wait times and build resources. To moderate the trade-off, a carefully considered batch size should be used that can optimise the priorities of the project that uses ML-CI.

We also see from Figure 9 that the median skipping based delays in ML-CI is 2 builds for all projects. This means that if a failing build is skipped by ML-CI, it is identified after a median of 2 builds. At a confidence interval $\alpha = 0.05$, a Kruskal-Wallis test indicated a significant difference in the median delays of the three ML-CI variants with a $p - value = 1.523e - 07$.

To further investigate this statistical difference, we used the Pairwise Wilcoxon Test at a confidence interval $\alpha = 0.05$ as a post-hoc testing measure. We measured

$p - value = 3.3e - 09$, and $p - value = 0.0001$, showing a strong statistical difference between the median delays of MLCI-BatchStop4 against MLCI-Batch4 and MLCI-BatchBisect respectively, with a large effect size of $r = 0.76$ and $r = 0.50$. From Table 6, we see that MLCI-BatchStop4 requires higher median delays than the other two variants. We can hypothesize that this difference arises due to the difference in BatchSize used by the three algorithms. The Batch4 variant uses a batch size of 4, while the BatchBisect variant is also seen to perform best with a median batch size of 4 builds. However, the BatchStop4 variant performs best with a median batch size of 8 builds. Table 6 shows the builds required for all variants of ML-CI grouped by the batch size. We see that the variants require similar percentages of builds when they are scheduled to build with the same batch size.

8.2 Comparison to Baselines

8.2.1 Approach

We choose one variant of ML-CI to compare to the baseline heuristics in this work. According to Bavand et al. [5], as the baseline Batch4 algorithm has the best performance among static batching techniques, we choose the MLCI-Batch4 variant to be compared to the baselines. Since the delays of SmartBuildSkip and the delays of the Batching algorithms are two different kinds of delays (skipping and grouping) and hence cannot be compared head-to-head, the following subsections separately compare ML-CI to SmartBuildSkip and to the Batch4 Batching algorithm. The scatterplots in Figure 11 and 13 plot the Percentage of Builds Required, whereas, Figure 10 and 12 plot the median delay, each of which compares ML-CI with the SmartBuildSkip and Batch4 baselines, respectively. These results are also noted in Table 9.

To determine whether ML-CI numerically outperforms the Baseline models, we look at the proximity of the data points to the X and Y axes. If the data point is closer to the Y axis compared to the X axis, then we can ascertain that ML-CI is a more effective build saving heuristic for the given project compared to the baseline algorithm. For simplicity, we plot the $y = x$ reference line on the graphs.

We used statistical tests to measure any differences in performance between the three ML-CI variants and the baseline algorithms. The results of these tests are reported in Tables 10 and 11.

8.2.2 Comparing to SmartBuildSkip

ML-CI incurs a median skipping delay of 2 builds across projects, significantly lower than that of SmartBuildSkip's 2 to 80 builds delay. From Figure 10, we see that all the data points of ML-CI lie below those of SmartBuildSkip, indicating that the median delay incurred by ML-CI is lower compared to that of SmartBuildSkip. Considering the magnitude of the delays, SmartBuildSkip appears to cause a median skipping delay greater than 10 builds for 11 projects out of the studied 20 projects. On the other hand, the median delay caused by ML-CI reaches 2 builds for all data points using the Batch4 ML-CI variant. Similarly, the median delays incurred by ML-CI ranges from 1 to 8 builds, with the maximum

Table 9: Comparing MLCI-Batch4 with Baseline algorithms

Project	Builds Required			Median Delay		
	MLCI-Batch4	Baseline Batch4	Smart-Build-Skip	MLCI-Batch4	Baseline Batch4	Smart-Build-Skip
cloud-controller-ng	62.269	74.691	11.265	2.0	1.5	7.0
cloudify	21.107	45.252	16.955	2.0	2.0	16.0
concerto	39.452	45.974	3.865	2.0	2.0	30.0
fog	69.775	83.73	26.587	2.0	1.5	7.0
geoserver	87.023	99.491	62.002	2.0	2.0	4.0
gradle	37.591	44.548	0.883	2.0	2.0	53.0
graylog2-server	30.709	34.996	1.225	2.0	2.0	12.0
heroku	36.267	53.333	14.578	2.0	2.0	11.0
jruby	73.061	108.406	79.32	2.0	2.0	3.0
loomio	54.104	73.485	17.298	2.0	1.5	10.0
metasploit-framework	34.04	37.476	1.023	2.0	1.5	80.0
opal	35.399	43.519	3.49	2.0	1.5	19.0
open-build-service	51.968	69.357	16.542	2.0	2.0	7.0
orbeon-forms	63.333	80.085	64.444	2.0	2.0	2.0
puppet	31.206	35.303	1.734	2.0	2.0	22.0
rails	44.471	72.799	24.109	2.0	1.5	6.0
rubinius	73.997	84.878	30.751	2.0	2.0	5.0
sonarqube	48.363	56.608	8.187	2.0	2.0	17.0
sufia	47.222	51.797	4.412	2.0	1.5	23.5
vagrant	26.429	39.552	2.378	2.0	2.0	27.0

Table 10: Comparing Median Delays of ML-CI variants with Baseline Algorithms

The Wilcoxon Signed Rank tests compare each baseline batching algorithm against the corresponding ML-CI variant.

	Test Used	p-values		
		MLCI-Batch4	MLCI-BatchBisect	MLCI-BatchStop4
SmartBuildSkip	Kruskal-Wallis		**2.238e-11**	
	Pairwise Wilcoxon	**8.9e-08**	**3.8e-06**	**8.8e-05**
Baseline Batch4	Wilcoxon Signed Rank	**0.004294**	n/a	n/a
Baseline BatchBisect	Wilcoxon Signed Rank	n/a	0.1554	n/a
Baseline BatchStop4	Wilcoxon Signed Rank	n/a	n/a	0.6266

delay of 408 builds, while it ranges from a median 5 to 80 builds, with maximum delay of 456 builds, in SmartBuildSkip.

A Kruskal Wallis test confirmed the significant difference between the median delay induced by the different ML-CI variants and SmartBuildSkip with $p - value = 2.238e - 11$ at $\alpha = 0.05$. The Pairwise Wilcoxon post-hoc test fur-

Table 11: Comparing Percentage of Builds Required by ML-CI variants with Baseline Algorithms

The Wilcoxon Signed Rank tests compare each baseline batching algorithm against the corresponding ML-CI variant.

	Test Used	p-values		
		MLCI-Batch4	MLCI-BatchBisect	MLCI-BatchStop4
SmartBuildSkip	Kruskal-Wallis	**3e-05**		
	Pairwise Wilcoxon	**0.00010**	**0.00024**	**0.00010**
Baseline Batch4	Wilcoxon Signed Rank	**0.05589**	n/a	n/a
Baseline BatchBisect	Wilcoxon Signed Rank	n/a	**0.03752**	n/a
Baseline BatchStop4	Wilcoxon Signed Rank	n/a	n/a	0.4612

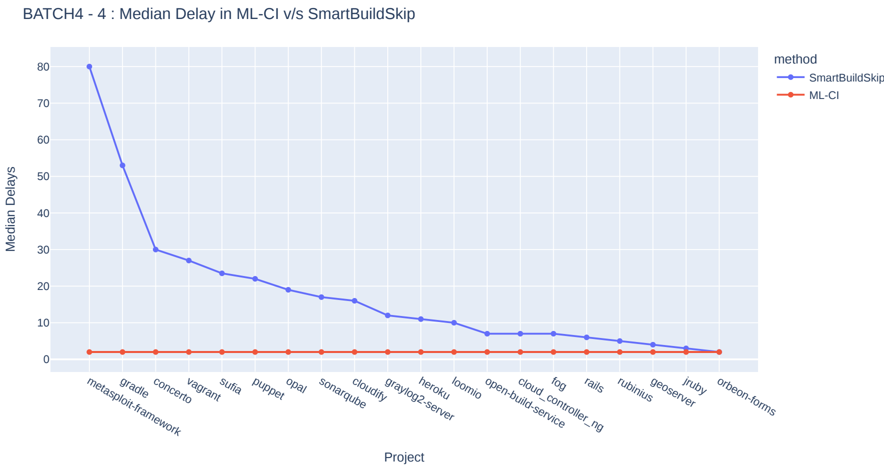


Fig. 10: Median Combined Delay in MLCI-Batch4 vs SmartBuildSkip algorithm.

ther showed that the median delay of SmartBuildSkip is significantly higher than the median delay of each of the ML-CI variants (Batch4, BatchBisect, BatchStop4) with $p-values$ of $(8.9e-08, 3.8e-06, 8.8e-05)$ and a large effect size of $(0.59, 0.51, 0.43)$.

ML-CI requires a median of 24.66% more builds compared to SmartBuildSkip. On the other hand, Figure 11 shows that a median percentage of 45.84% of builds across the 20 projects are run with ML-CI. All projects require more than 20% of builds (minimum of 21.10% in Cloudify). In contrast with SmartBuildSkip, the projects require a median of 12.92% of builds to schedule, with all but 3 projects requiring less than 31% of builds. The maximum percentage of builds required is about 79.31% for JRuby, while the same project requires 73.06% builds with ML-CI.

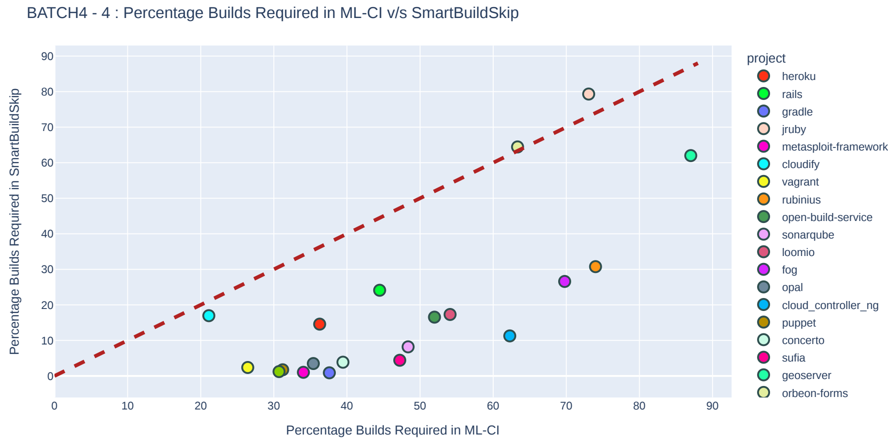


Fig. 11: Percentage of Required builds in ML-CI (using Batch4) vs SmartBuildSkip algorithm.
The diagonal represents x=y, hence all points below the line are instances where SmartBuildSkip performs better than ML-CI.

A Kruskal Wallis test performed to analyze the statistical difference in the percentage of builds required by SmartBuildSkip and the three variants of ML-CI revealed a significant difference between them with a $p - value = 0.06541$ at $\alpha = 0.05$. The pairwise Wilcoxon rank sum post-hoc test further calculated a $p - value = 0.00010$ between SmartBuildSkip and each MLCI-Batch4 and MLCI-BatchStop4 variant, with a large effect size of $r = 0.43$. Similarly, we also calculated a $p - value = 0.00024$ with a large effect size of $r = 0.41$ between SmartBuildSkip and the MLCI-BatchBisect variant. From Table 3 and Table 6, we know that SmartBuildSkip requires much less builds (median of 12.92%) as compared to the MLCI-variants.

8.2.3 Comparing to Batch4 Baseline

We see a close call between ML-CI and the Baseline Batch4 algorithm in terms of percentage of builds required and median combined delay, with an improvement of small effect size in grouping delay for the Batch(Stop)4 variants of ML-CI. In Figures 12 and 13, we see that both the ML-CI Batch4 variant and the baseline Batch4 algorithm have similar values of Median Delay and Percentage of Builds Required. All projects have a median delay of 1.5-2 builds for the baseline Batch4, and a similar median of 2 builds for the Batch 4 ML-CI variant.

In Figure 13, all data points lie near or slightly above the $y = x$ reference line, indicating that in most cases ML-CI requires slightly less builds compared to the baseline Batch4 algorithm. The maximum percentage of builds is 108.41% in baseline Batch4 (JRuby) and 87.02% for the MLCI-Batch4 variant (Geoserver).

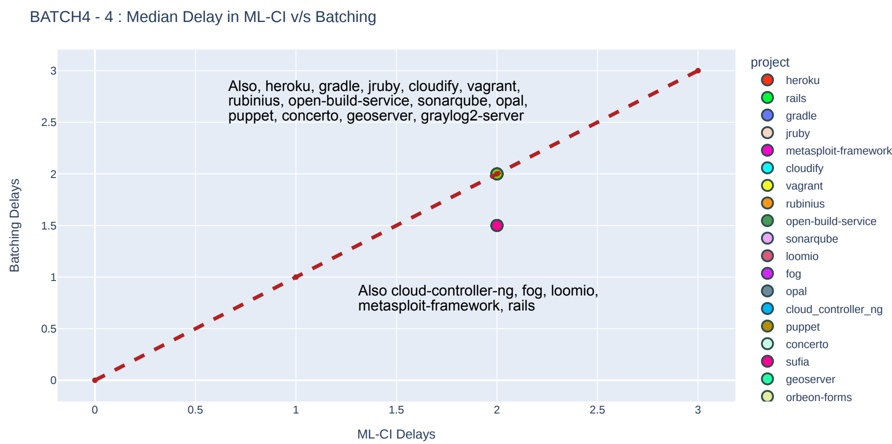


Fig. 12: Median Combined Delay in ML-CI (using Batch4) vs. Baseline Batch4 Batching algorithm.
The diagonal represents x=y, hence all points below the line are instances where baseline batching performs better than ML-CI.

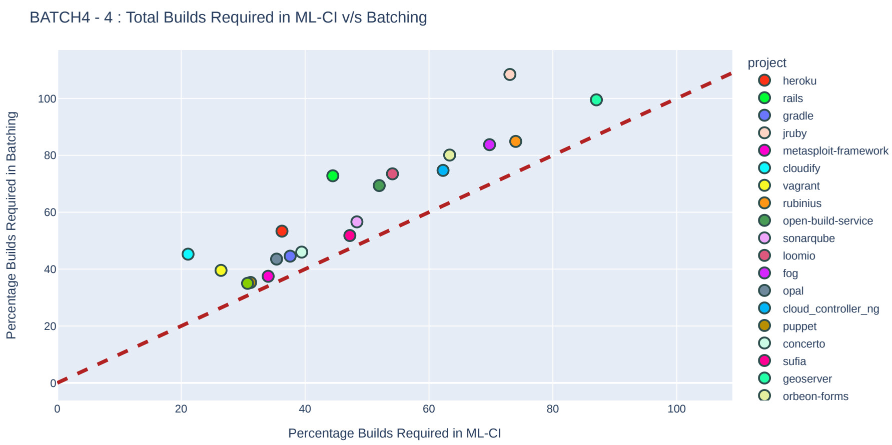


Fig. 13: Percentage of Required builds in ML-CI (using Batch4) vs Baseline Batch4 Batching algorithm.
The diagonal represents x=y, hence all points above the line are instances where ML-CI performs better than baseline batching.

The lowest percentage of builds requires 21.01% builds for MLCI-Batch4(cloudify), and 34.99% of builds for the baseline Batch4(Graylog2-server) technique.

We performed a Wilcoxon Signed Rank test to determine statistical differences between the percentage of builds required by each variant of ML-CI and its corresponding Baseline variant (e.g., MLCI-Batch4 variant compared to the baseline Batch4 approach). As shown in Table 19, the test calculated p-values=

$(0.05589, 0.03752, 0.4612)$ for the Batch4, BatchBisect and BatchStop4 ML-CI variants, respectively, revealing no significant differences in percentage of builds required for the MLCI-Batch4 and MLCI-BatchStop4 variants and their corresponding baseline algorithms. However, we see that the MLCI-BatchBisect variant has a significantly better performance in comparison to Baseline BatchBisect algorithm with a large effect of $r = 0.23$.

One possible explanation could be that ML-CI performs very well with small batch sizes. We have found that projects like Fog, Rails, Geoserver etc., that work best with smaller batch sizes in ML-CI, have higher failure rates. Due to the incorporation of SmartBuildSkip in the ML-CI algorithm, we find that ML-CI can schedule less builds using a smaller batch size than baseline batching algorithms. For example, from Table 3, we see that project Cloud-controller-ng requires 80.40% builds with a batch size of 4 commits and that project Cloudify requires 42.86 builds with a batch size of 16 commits using baseline BatchBisect algorithm. On the other hand, from Table 6 we see that the same projects require a median of 55.56% less builds with batch size 2 and 23.95% less builds with batch size 4.

However, we also see that ML-CI variants require similar builds as their baseline counterparts for projects like Graylog2-server and Metasploit-framework. Particularly, these projects have a low failure rate of less than 4%. This would mean that the ML-CI models trained on these projects have not seen enough build failures during the training phase, making it difficult for them to identify build failures during testing. Hence, the performance of the two techniques becomes similar to each other.

Table 18 shows the statistical difference of the median combined delays. We obtained a $p-value = 0.004294$ for the Batch4 variant, $p-value = 0.1554$ for the BatchBisect variant and $p-value = 0.6266$ for the BatchStop4 variant, indicating that there are differences between the median delays of the ML-CI Batch4 variant and Baseline Batch4 at a confidence interval of $\alpha = 0.05$. That said, we computed an effect size of $r = 0.31$, which denotes a small effect.

8.3 Discussion

From the ML-CI approach, we see that predicting the build outcome (including a time-out feature) and using this prediction to skip and group builds does not consistently improve on either SmartBuildSkip or Batching. While it can significantly reduce the median number of skipping delays with respect to SmartBuildSkip, it also increases the percentage of required builds, or it only improves on the percentage of builds required with a small effect size compared to baseline batching approaches.

To better understand these findings, we decided to study the role of individual features on the overall prediction of ML-CI. For this, we measured feature importance of each feature used in the ML-CI models using impurity-based importance [32]. Table 12 shows for each feature the median feature importance score, ranking the features from highest to lowest median.

The analysis shows how the three features with the highest feature importance scores are *gh_commits_on_files_touched*, *time_out* and *git_diff_src_churn*. While *gh_commits_on_files_touched* and *git_diff_src_churn* are relatively "generic" features (i.e., applicable to other fields than CI), feature *time_out* is highly specific to CI and

Table 12: Most Important Features of ML-CI

Order	Median Score	Feature Name	Feature Description
1	0.295	gh_commits_on_files _touched	Frequency of commits made on the files touched by commit
2	0.203	time_out	Most recent number of consecutive predicted build passes or build skips leading up to current build
3	0.184	git_diff_src_churn	Size of commit sent to build
4	0.098	git_diff_files_modified	Number of files modified by current build
5	0.088	git_diff_test_churn	Number of lines of test file changes
6	0.043	git_num_all_built _commits	Number of commits added to current build
7	0.039	git_diff_files_added	Number of files added in current build
8	0.018	git_diff_files_deleted	Number of files deleted by current build
9	0.005	git_num_commit _comments	Number of comments added in the commit

its high median importance seems to suggest that a kind of time-out mechanism is crucial when predicting build outcomes. Notably, the higher the number of passes, the higher the probability that builds are scheduled by ML-CI.

In contrast, SmartBuildSkip does not have any such mechanism to "reset" its internal state, such that a very long streak of predicted build passes might lead SmartBuildSkip not to schedule any build for a very long time, causing the high skipping delays of Figure 10. Batch4 does have a fixed batch size, but it ignores prior success/failure.

Hence, given these issues of SmartBuildSkip and Batch4, as well as the observations about the role played by *time_out* in the ML-CI model, we analyze in the next RQ whether the Timeout Rule proposed in Section 6.2 is able to improve on SmartBuildSkip's lack of reset mechanism.

Summary of RQ2: *We observe that the performance of the ML-CI Model is on par with (or at least only slightly better than) that of commit grouping algorithms and requires a median of 26.44% more builds than SmartBuildSkip, albeit with a median skipping delay of 9.5 builds less.*

9 Performance Results for Timeout Rule (RQ3)

This section compares the performance of the Timeout Rule against the baseline SmartBuildSkip and Batching algorithms. Section 9.1 reports the results comparing the variants of the Timeout Rule. Sections 9.2.2 and 9.2.3 compare the results of the Timeout Rule with SmartBuildSkip and Batch4, respectively.

9.1 Comparing the variants of the Timeout Rule

9.1.1 Approach

Similar to ML-CI, we derive three main variants of the Timeout Rule as detailed in Table 13. We study the performance of the three main variants of the Timeout Rule in this section. By varying the batch size used with the Timeout Rule we obtained 8 sub-variants. Other variants of the Timeout Rule can be derived by using other commit-grouping techniques that are not used in this work.

The percentage of builds required and the median delay observed for the 20 studied projects, for each variant of the Timeout Rule is listed in Table 14. Similar to ML-CI, for the Batch4 column in the table, each entry is obtained by using a batch size of 4 builds. For the BatchBisect column, the batch size can vary between [1, 2, 4, 8, 16] whereas for the BatchStop4 column, the batch size can be 8 or 16 builds. Figure 14 depicts the box plots of Percentage of Builds required, median combined delays and best batch sizes used by the Timeout Rule variants across 20 projects. While Figure 14c depicts the distribution of best performing timeout values across 20 projects, each of the 20 data points in Figures 14a and 14b represent the percentage of builds required and median delay obtained with the best performing batch size for that Timeout Rule variant.

In Section 9.2, we further choose the TR-Batch4 variant to dive into deeper analysis to compare the performance between the Timeout Rule, Baseline Batch4 and SmartBuildSkip algorithms. We choose the Batch4 variant as a representative of baseline commit grouping algorithms to simplify comparisons as it uses a single batch size of 4 commits throughout the CI process for all projects.

It is important to note that we do not use the same fixed size of Timeout value across all analyzed projects. On evaluating each Timeout Rule sub-variant with Timeout values in the range [2, 20], we choose for each project the confidence value that gives the best results. For example, the best Timeout value used for project Rails with TR-Batch4 variant is 20, but project Rubinius works best with the Timeout value of 11 and Orbeon-forms uses a Timeout value of 16.

We also use the Kruskal-Wallis and Wilcoxon Signed Rank statistical tests to statistically compare the median delay and percentage of builds required by the variants of Timeout Rule amongst each other or with baseline heuristics. We use two Kruskal Wallis tests, one to compare median delay and another to compare percentage builds required by the three main variants of the Timeout Rule in Table 14. We use the post-hoc Pairwise Wilcoxon test at confidence values of $\alpha = 0.05$ to further investigate any statistical differences indicated by the Kruskal-Wallis Test. We also use a Wilcoxon signed rank test to calculate statistical differences between each Timeout Rule variant in Table 14 with its corresponding commit grouping variant in Table 3.

9.1.2 Results

The Timeout Rule requires a median percentage of builds of 29.30% for the BatchBisect variant, 34.96% builds for the Batch4 variant, and 41.11% for the BatchStop4 variant, demonstrating a significant reduction in the percentage of builds required during CI. Three quarters of projects require less than 46.40% builds for the BatchBisect variant, less than

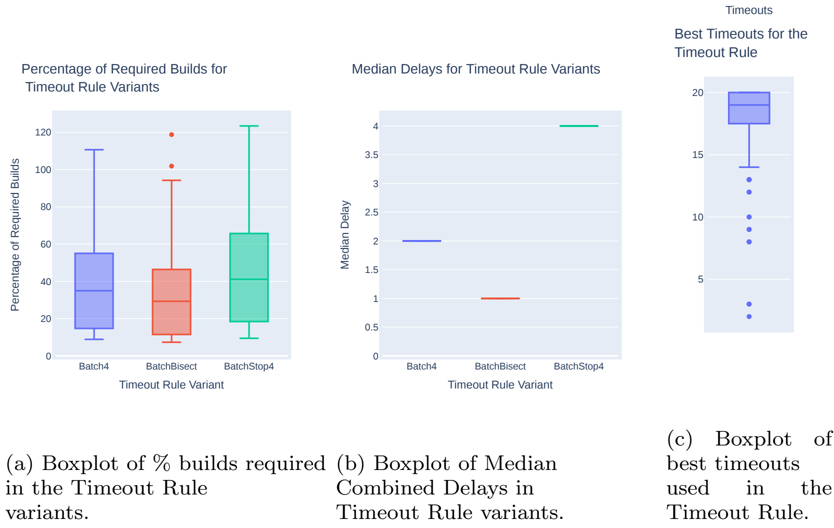
Table 13: Variants of the Timeout Rule

Grouping Heuristic	Batch Size Used	Obtained Variant
Batch4	4	*TR-Batch4*
BatchBisect	1	*TR-BatchBisect-1*
	2	*TR-BatchBisect-2*
	4	*TR-BatchBisect-4*
	8	*TR-BatchBisect-8*
	16	*TR-BatchBisect-16*
BatchStop4	8	*TR-BatchStop4-8*
	16	*TR-BatchBisect-16*

Table 14: Results of Timeout Rule

Project	Timeout Rule								
	TR-Batch4			TR-BatchBisect			TR-BatchStop4		
	% Builds Required	Median Delay	Batch Size	% Builds Required	Median Delay	Batch Size	% Builds Required	Median Delay	Batch Size
cloud-controller-ng	38.812	2.0	4	30.247	1.0	2	48.457	4.0	8
cloudify	34.025	2.0	4	33.064	1.0	2	39.216	4.0	8
concerto	16.667	2.0	4	12.399	1.0	2	20.692	4.0	8
fog	56.217	2.0	4	49.008	1.0	2	65.542	4.0	8
geoserver	85.496	2.0	4	94.232	1.0	2	98.388	4.0	8
gradle	8.879	2.0	4	7.373	1.0	2	9.761	4.0	8
graylog2-server	9.099	2.0	4	7.437	1.0	2	9.449	4.0	8
heroku	35.911	2.0	4	28.356	1.0	2	43.022	4.0	8
jruby	110.664	2.0	4	118.779	1.0	2	123.429	4.0	8
loomio	42.677	2.0	4	34.596	1.0	2	51.641	4.0	8
metasploit-framework	9.064	2.0	4	7.334	1.0	2	10.38	4.0	8
opal	15.527	2.0	4	11.823	1.0	2	19.088	4.0	8
open-build-service	43.049	2.0	4	34.479	1.0	2	54.061	4.0	8
orbeon-forms	95.043	2.0	4	101.88	1.0	2	109.829	4.0	8
puppet	11.505	2.0	4	8.511	1.0	2	13.16	4.0	8
rails	53.852	2.0	4	43.802	1.0	2	65.854	4.0	8
rubinius	69.848	2.0	4	61.78	1.0	2	81.835	4.0	8
sonarqube	28.48	2.0	4	22.164	1.0	2	35.322	4.0	8
sufia	18.219	2.0	4	14.379	1.0	2	22.958	4.0	8
vagrant	13.946	2.0	4	11.157	1.0	2	17.787	4.0	8
Median	34.968	2.0		29.301	1.0		41.119	4.0	

55.03% for the Batch4 variant, and less than 65.69% for the BatchStop4 variant. Project Metasploit-Framework requires the least builds, i.e., 7.33% using TR-BatchBisect-2 variant, while the maximum percentage of builds required occurs for JRuby, i.e., 123.42% using TR-BatchStop4-8 variant. Having more than 100% builds required means that there are more builds scheduled than the number of incoming commits. This could happen in instances where commit grouping is ap-



(a) Boxplot of % builds required in the Timeout Rule variants.

(b) Boxplot of Median Combined Delays in Timeout Rule variants.

(c) Boxplot of best timeouts used in the Timeout Rule.

Fig. 14: Boxplots demonstrating range of Percentage Build Required, Median Delays, and best timeouts to use in Timeout Rule.

Table 15: Comparing Median Delays of the Timeout Rule variants with each other

	Test Used	p-values		
		TR-Batch4	TR-BatchBisect	TR-BatchStop4
	Kruskal-Wallis	**1.543e-13**		
TR-Batch4	Pairwise Wilcoxon	–	**4.7e-10**	**4.7e-10**
TR-BatchBisect	Pairwise Wilcoxon	–	–	**4.7e-10**

Table 16: Comparing Percentage of Builds Required by the Timeout Rule variants amongst each other

	Test Used	p-values		
		TR-Batch4	TR-BatchBisect	TR-BatchStop4
	Kruskal-Wallis	0.3688		
TR-Batch4	Pairwise Wilcoxon	Not needed, since Kruskal-Wallis test shows no significant difference		
TR-BatchBisect	Pairwise Wilcoxon			

plied for a batch full of failing commits, yielding to massive bisection activity. In these cases, building each incoming commit individually is better than using commit grouping algorithms.

In order to determine any statistical differences between the three variants of the Timeout Rule, we used the Kruskal Wallis test, as recorded in Table 16. By performing the test at a confidence level of $\alpha = 0.05$, we computed $p - value =$

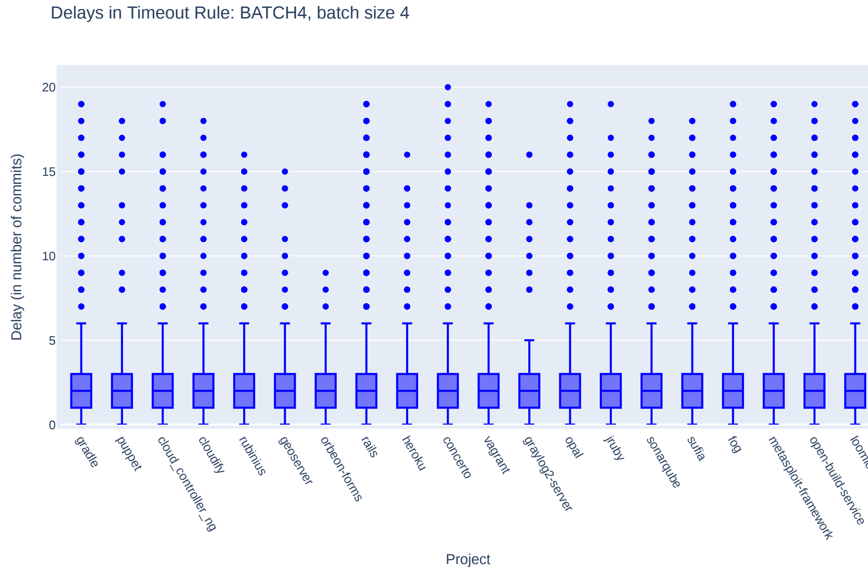


Fig. 15: Median combined delays made per project by TR-Batch4 variant.

0.3688, which signifies no statistical difference between the builds required for the 3 variants of this rule.

The median combined delays in a project for the Timeout Rule range from 1 to 4 builds, with a 2 build median delay across the TR-Batch4 variant, 1 build delay for the TR-BatchBisect variant and 4 build delay for the TR-BatchStop4 variant. Figure 15 shows the distribution of delays for the TR-Batch4 variant. The plot shows that the median delay is 2 builds for all projects. This means that if a failing build is skipped by the Timeout Rule, that build is frequently identified within 2 builds. The plots for the other two variants look similar, but with a different (consistent) median delay across all projects. With the Timeout Rule, we find that projects yield a consistent median delay for a given variant (cf. Table 14). This is because, with the Timeout Rule, builds are scheduled more frequently to identify build failures as early as possible. Doing so ensures that skipping-based delays are reduced to a minimum. Since builds that are scheduled are also scheduled in batches, this increases the number of grouping-based delays. Hence, the median delay is majorly influenced by the batch size used by the Timeout Rule variant. If two projects use the same batch size, chances are that they make the same median delay in failing build identification.

On performing a Kruskal Wallis test, we measured $p - value = 1.543e - 13$ for the 3 Timeout Rule variants, which indicates a statistical difference with large effect size ($r = 0.80$) between the median delays of the 3 variants. The Pairwise Wilcoxon post-hoc test at a confidence interval of $\alpha = 0.05$ further confirmed that statistically significant differences arise between the three variants.

We find that smaller batch sizes are more effective with the Timeout Rule. Table 14 shows the performance numbers of the 3 variants of the Timeout

Rule based on their batch sizes. We see that similar batch sizes have similar performance across each variant, while varying the batch size can improve or impair the performance. Larger batch sizes reduce the percentage of builds required along with reducing the number of skipping-based delays. However, they also increase the wait time of builds in the build queue while new incoming builds fill up the queue. This increases the cumulative delay for a build.

9.2 Comparison to Baselines

9.2.1 Approach

In the following sections of the paper, we compare the Timeout Rule to the Baseline Batch4 and SmartBuildSkip algorithms, and we also study the impact of different timeout thresholds. Similar to ML-CI, the following subsections separately compare the Timeout Rule to SmartBuildSkip and to the Batch4 Batching algorithm. We again focus on the comparison of the Timeout Rule to Batch4 as it has a performance comparable or better than other baseline batching heuristics [5]. Figure 17 and 19 represent the scatter plots that plot the Percentage of Builds Required, whereas the scatter plots in Figures 16 and 18 plot the median delay, each of which compares the Timeout Rule with the SmartBuildSkip and Batch4 baselines, respectively. These results are also shown in Table 17.

9.2.2 Comparing to SmartBuildSkip

The Timeout Rule induces a median skipping delay of 9.5 commits less than SmartBuildSkip, but requires a median of 20.81% more builds. In Figure 16, we see the median delay incurred for SmartBuildSkip and the TR-Batch4 variant. We see that all the data points for the SmartBuildSkip heuristic are above those of the TR-Batch4 line, meaning that the delays are lower for the Timeout Rule than for SmartBuildSkip. Compared to the median delay of 11.5 builds required by SmartBuildSkip, the delays are substantially lower in the Timeout Rule. All the data points have a median skipping delay of 2 builds for the TR-Batch4 variant, whereas more than half of the projects have a median delay of more than 10 commits for SmartBuildSkip. Similarly, the median delay of TR-BatchBisect and TR-BatchStop4 variants is 4 builds. This indicates that the Timeout Rule reduces median delays by 7.5-10.5 builds in comparison to SmartBuildSkip. The maximum median delay with SmartBuildSkip for 80 builds with Metasploit-Framework, only has 2 builds of median delay with TR-Batch4. The minimum median delay of 2 commits is made by Orbeon-forms in SmartBuildSkip, which also yields 2 commits of delay with the TR-Batch4.

On the other hand, SmartBuildSkip requires 20.81% less builds compared to the Timeout Rule in most projects. In fact, the highest percentage of builds required by SmartBuildSkip is 79.32% in project JRuby, compared to 110.66% of builds for the TR-Batch4 variant. However, with a few exceptions like Rails, Orbeon-forms, Rubinius, Fog, JRuby and Geoserver, all other projects require less than 50% of total builds using the Timeout Rule, with 8 of them requiring less than 20% builds. In other words, Figures 16 and 17 clearly show a trade-off between percentage of builds required and median skipping delay for the Timeout

Table 17: Comparing TR-Batch4 with Baseline algorithms

Project	Builds Required			Median Delay		
	TR-Batch4	Baseline Batch4	Smart-Build-Skip	TR-Batch4	Baseline Batch4	Smart-Build-Skip
cloud-controller-ng	38.812	74.691	11.265	2.0	1.5	7.0
cloudify	34.025	45.252	16.955	2.0	2.0	16.0
concerto	16.667	45.974	3.865	2.0	2.0	30.0
fog	56.217	83.73	26.587	2.0	1.5	7.0
geoserver	85.496	99.491	62.002	2.0	2.0	4.0
gradle	8.879	44.548	0.883	2.0	2.0	53.0
graylog2-server	9.099	34.996	1.225	2.0	2.0	12.0
heroku	35.911	53.333	14.578	2.0	2.0	11.0
jruby	110.664	108.406	79.32	2.0	2.0	3.0
loomio	42.677	73.485	17.298	2.0	1.5	10.0
metasploit-framework	9.064	37.476	1.023	2.0	1.5	80.0
opal	15.527	43.519	3.49	2.0	1.5	19.0
open-build-service	43.049	69.357	16.542	2.0	2.0	7.0
orbeon-forms	95.043	80.085	64.444	2.0	2.0	2.0
puppet	11.505	35.303	1.734	2.0	2.0	22.0
rails	53.852	72.799	24.109	2.0	1.5	6.0
rubinius	69.848	84.878	30.751	2.0	2.0	5.0
sonarqube	28.48	56.608	8.187	2.0	2.0	17.0
sufia	18.219	51.797	4.412	2.0	1.5	23.5
vagrant	13.946	39.552	2.378	2.0	2.0	27.0

Rule. Although the number of builds are slightly higher than the SmartBuildSkip algorithm, the Timeout Rule can significantly reduce the delays compared to the former.

To validate the improvement in performance for the Timeout rule, we used Kruskal Wallis tests between the performance of the four methods (i.e., SmartBuildSkip and the three Timeout Rule variants, with best performing batch size and timeout thresholds for each project). With $\alpha = 0.05$, the test yields a $p-value = 0.0070$ for percentage of builds required and $p-value = 8.581e-16$ for the median skipping delay , indicating a significant difference in the performance of SmartBuildSkip and the three variants. Further analysis using the pairwise Wilcoxon rank sum post-hoc test confirmed that the Timeout Rule variants require significantly lower median delays compared to SmartBuildSkip with $p-values = (3.6e-08, 1.2e-08, 1.2e-05)$ and effect sizes $r = (0.61, 0.63, 0.48)$, denoting a significant effect.

9.2.3 Comparing to Batch4

In comparison to the Baseline Batch4 algorithm, the Timeout Rule reduces the percentage of builds required during CI and makes either similar or 0.5 commits higher median delays. In Figures 18 and 19, we

Table 18: Comparing Median Delay of Timeout Rule variants to Baseline Algorithms.

The Wilcoxon Signed Rank tests compare each baseline batching algorithm against the corresponding ML-CI variant.

	Test Used	p-values		
		TR-Batch4	TR-BatchBisect	TR-BatchStop4
SmartBuildSkip	Kruskal-Wallis	**8.581e-16**		
	Pairwise Wilcoxon	**3.6e-08**	**1.2e-08**	**1.2e-05**
Baseline Batch4	Wilcoxon Signed Rank	**0.004294**	n/a	n/a
Baseline BatchBisect	Wilcoxon Signed Rank	n/a	**9.618e-05**	n/a
Baseline BatchStop4	Wilcoxon Signed Rank	n/a	n/a	1

Table 19: Comparing the Percentage of Builds Required by Timeout Rule variants to Baseline Algorithms.

The Wilcoxon Signed Rank tests compare each baseline batching algorithm against the corresponding ML-CI variant.

	Test Used	p-values		
		TR-Batch4	TR-BatchBisect	TR-BatchStop4
SmartBuildSkip	Kruskal-Wallis	**0.007021**		
	Pairwise Wilcoxon	**0.022**	**0.061**	**0.011**
Baseline Batch4	Wilcoxon Signed Rank	**0.006715**	n/a	n/a
Baseline BatchBisect	Wilcoxon Signed Rank	n/a	**0.004267**	n/a
Baseline BatchStop4	Wilcoxon Signed Rank	n/a	n/a	0.1081

see that performance of the Timeout Rule compared to the Baseline Batch4 algorithm. With traditional batching algorithms, commits bide their time in the build queue as they wait for it to fill up with new incoming commits. However, with the Timeout Rule, we reduce the number of commits that need to be built, hence reducing the median delay of these commits, by selectively building batches of incoming commits based on prediction of build failure-proneness. With the reduction in build wait time, we see that the Timeout Rule has either very similar delays to that of Baseline Batch4 or higher.

Similarly, by choosing only few builds to group together, the Timeout Rule also reduces the total number of builds made in a project. Hence, in extreme cases we can see a reduction in the percentage of builds required compared to traditional batching from 80.09% to 30.24% in the case of Cloud Controller, 82.82% to 34.59% in Loomio and so on. In general, the Timeout Rule yields 26.10% less builds than Baseline Batch4, 29.67% less builds than Baseline BatchBisect and 15.67% less builds than Baseline BatchStop4.

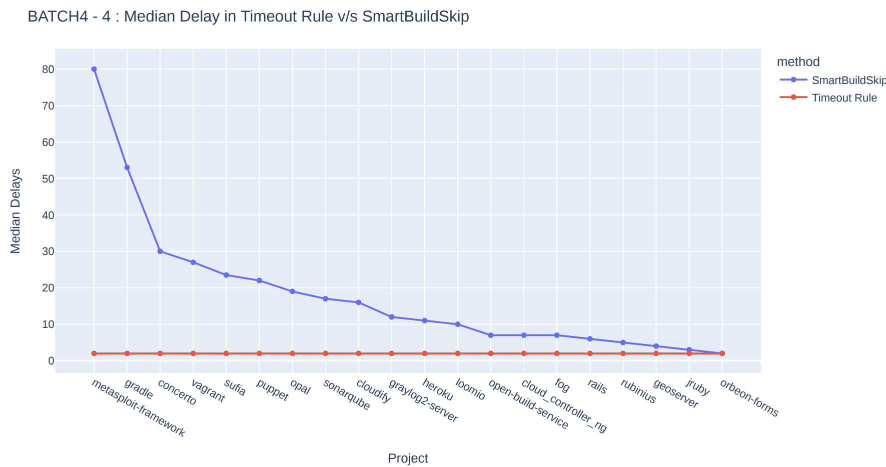


Fig. 16: Median Combined Delay for the Timeout Rule (using Batch4 and best performing timeout threshold for that project) vs SmartBuildSkip algorithm.

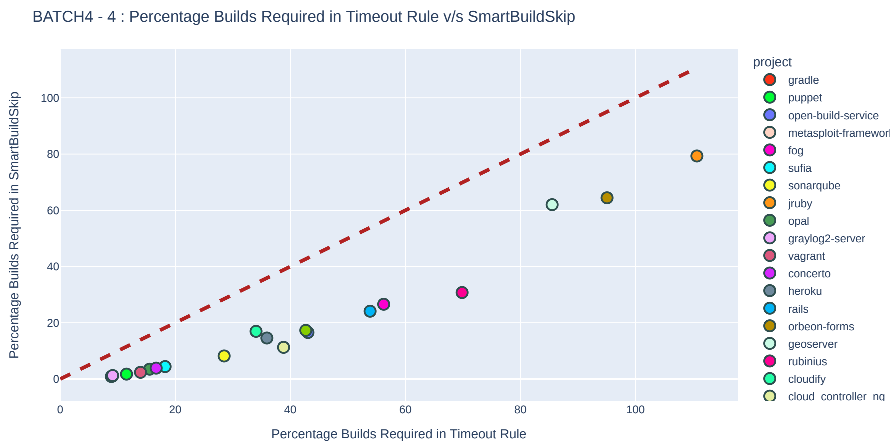


Fig. 17: Percentage of Required builds for the Timeout Rule (using Batch4 and best performing timeout threshold for that project) vs SmartBuildSkip algorithm. The diagonal represents x=y, hence all points below the line are instances where SmartBuildSkip performs better than the Timeout Rule.

However, we also note that the Timeout Rule does not always yield less builds than Baseline Batching algorithms. For example, we see from Figure 19 that projects JRuby and Orbeon-forms require less builds with the baseline Batch4 algorithm. We attribute this to the higher build failure rate of the two projects. Additionally, with a baseline batching technique all incoming commits are batched together, which means that both successful and failing commits are grouped together in a batch. Moreover, some of the batches may also comprise more successful

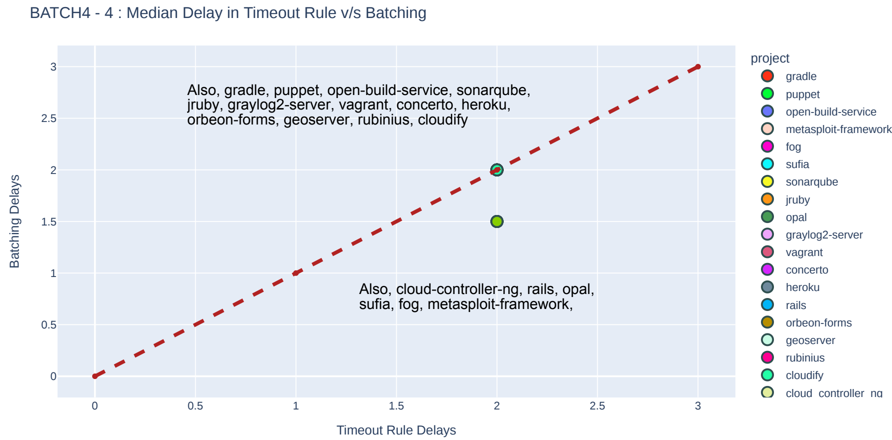


Fig. 18: Median Combined Delay in the Timeout Rule (using Batch4 and best performing timeout threshold for that project) vs Baseline Batch4 Batching algorithm.
The diagonal represents x=y, hence all points below the line are instances where baseline batching performs better than the Timeout Rule.

than failing commits. However, with the Timeout Rule, we begin to build commits only when the SmartBuildSkip model identifies a higher probability of build failure. In doing so, batches of the Timeout Rule comprise more failing commits than successful commits, which take more builds to identify. In the real world, however, batching techniques can be leveraged to increase the probability of a bug-fixing commit pushed into the same batch as a bug-inducing commit. For example, if developers observe small, transient failures in a commit pushed to CI, they can push a bug-fixing commit in the same batch and avoid build failures.

A Wilcoxon Signed Rank test at a confidence interval of $\alpha = 0.05$, between the median delays of each pair of Timeout rule variant and baseline, we found that there is a significant difference between the performance of each variant of the Timeout Rule and its corresponding baseline batching algorithm. Between the Timeout rule variants (Batch4, BatchBisect, BatchStop4) and their corresponding baseline algorithms, we computed $p - values = (0.0067, 0.0042, 0.1081)$ for percentage of builds required, and $p - values = (0.0042, 9.618e - 05, 1)$ for median delays. This indicates that TR-Batch4 and TR-BatchBisect variants have significant performance differences in comparison to their baseline counterparts.

9.3 Comparing Timeout values of the Timeout Rule

By increasing the timeout threshold and lowering the batch size of the Timeout Rule variants, we can achieve better performance.

In Figure 20, we analyse the Rails project to inspect the effects of parameters of the Timeout Rule on its performance. We study Rails as it requires the lowest median delay when using SmartBuildSkip, and requires less than 100% builds

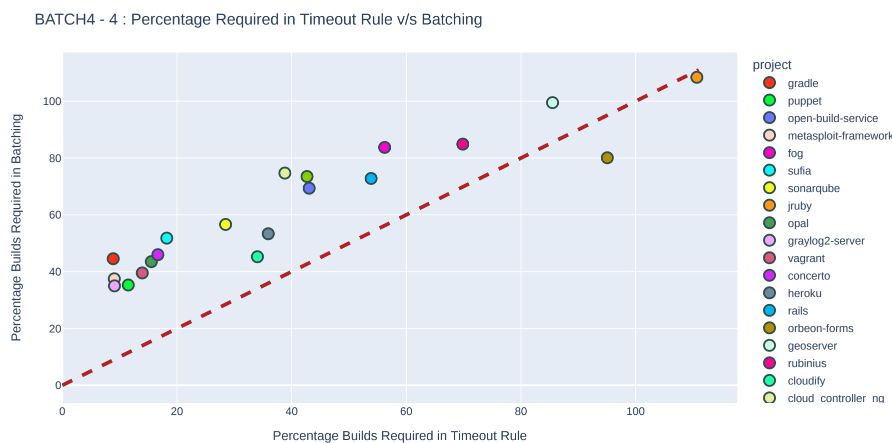


Fig. 19: Percentage of Required builds in the Timeout Rule (using Batch4 and best performing timeout threshold for that project) vs Baseline Batch4 Batching algorithm.
The diagonal represents x=y, hence all points above the line are instances where the Timeout Rule performs better than baseline batching.

with commit-grouping techniques, making its builds and delays easily comparable to the delays of our Timeout Rule variants. The graphs for other projects are in the online replication package. To do so, we record the performance of the Timeout Rule variants for varying batch sizes and timeout thresholds. The lower the median delay and the percentage of builds required, the better the performance of the corresponding heuristic, i.e., the closer a data point is to the origin, the better performance it denotes.

Although SmartBuildSkip requires close to 24.1% builds, it induces a median skipping delay close to 6 builds. From Figures 6 and 15, we also learnt that the median delay for the Batch4-Timeout variant in Rails is 2 builds. On the other hand, we see that reducing the batch size of the baseline batching algorithms decreases the batching based delays.

In particular, we notice from the figure that the delays induced by the Timeout Rule can be reduced by decreasing the batch size used, and the percentage of builds required by the Rule can be reduced by increasing the timeout value. Batch sizes 2 and 4 with higher timeout values of 16 and above are closer to the origin, indicating that these configurations of the Timeout have the best performance compared to the rest of the configurations in comparison to other heuristics for Rails. As we increase the timeout value, the skipping-based delay can also increase due to the larger gap between two mandatory builds. However, for each mandatory build of batch size 'N' commits, 'N-1' number of grouping-based delays are incurred. This makes the number of grouping-based delays significantly higher than the number of skipping-based delays. Due to which we notice the median combined-delays to shift and reflect the grouping-based delays, which remain more or less constant for a given batch size.

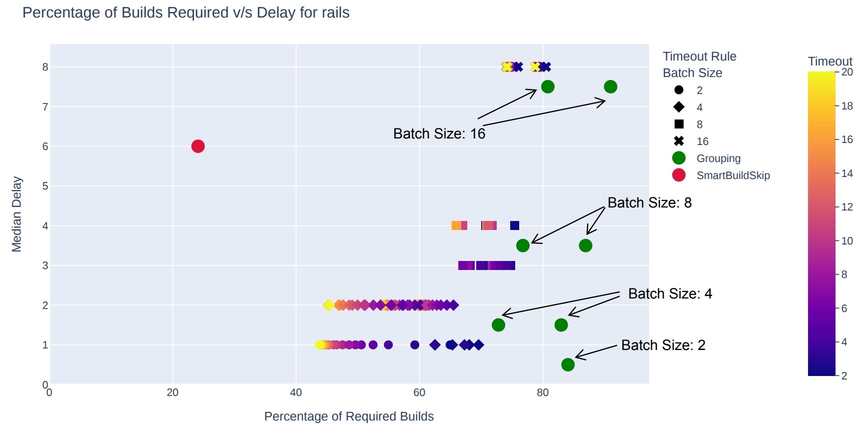


Fig. 20: Percentage of Builds Required vs. Median Delay for each timeout threshold, batch size and batching algorithm of the Timeout Rule for Rails.

Summary of RQ3: *The percentage of builds required by the Timeout Rule is 26.10% lower than for Batch4, but 20.81% greater than SmartBuildSkip. The median delay induced by the Timeout Rule is, however, 9.5 builds builds lower than for SmartBuildSkip, but similar to that of baseline Batch4. The performance of the Timeout Rule can also be improved by reducing the batch size or increasing the timeout threshold.*

10 Study Implications

In this section, we discuss the implications of our study and its practical usage in real-world CI scenarios.

10.1 Trade-offs between Number of Builds and Delays

In Section 5, we discussed the limitations of the existing SmartBuildSkip algorithm and commit grouping techniques. On the one hand, we see that while reducing the number of builds required during CI, SmartBuildSkip induces larger delays in identifying failing builds. On the other hand, commit grouping techniques schedule more builds, which helps to reduce median delays. By combining these two

heuristics together in ML-CI and Timeout Rule, we provide an opportunity for practitioners to make use of the trade-off between the two performance metrics.

Depending on the priorities of the project and company, practitioners can choose to schedule a minimum number of builds by using larger batch sizes in ML-CI, or larger timeout values and batch sizes in the Timeout Rule. By increasing the batch size, practitioners also provide more opportunities for a transient build failure in an existing commit to be fixed by a subsequent commit of the same batch, allowing the entire batch to compile successfully (avoiding costly bisection). Additionally, increasing the timeout threshold will also limit the frequency at which builds are scheduled without any build-fail prediction. However, increasing the batch size or timeout values will increase the median delay by either increasing the wait time spent in the build queue or increasing the delay of failing build identification.

With ML-CI and Timeout Rule, practitioners can choose to either only reduce the number of builds required, only reduce the delays or find an optimal balance between both performance metrics for their specific context/project. For example, from Figure 20, we see that Rails requires the least percentage of builds and lowest delays with a batch size of 2 and a timeout value of 20, compared to it using a batch size of 8 and timeout value of 16. Builds can be further reduced by further increasing the timeout value.

10.2 Time Saved by Build Saving Heuristics

Reducing the builds required helps reduce the resources required in CI. For example, the less the builds are scheduled, the less energy consumed by servers. Another way in which we could analyse the impact of reducing CI builds is by estimating the time reduced by using these heuristics. This requires a number of assumptions. First, we consider as baseline the sum of the duration of all builds. For commit grouping techniques, the duration of a batch build is estimated to be the duration required to individually compile the last commit of the batch. For Smart-Build-Skip, ML-CI and Timeout Rule, we estimate the required build time as the sum of the durations of the scheduled builds. Table 20 shows the time saved by SmartBuildSkip, baseline commit grouping techniques, ML-CI and the Timeout Rule. When a heuristic saves X% time, it means that it requires X% less time than what is required while scheduling individual builds.

We note that the less builds are required for a heuristic, the more actual time is saved. By saving 91.67% time, SmartBuildSkip saves the maximum amount of time, followed by the Timeout Rule which saves 63.36% time. Although SmartBuildSkip saves more time and requires less builds, our findings in RQ3 showed how the Timeout Rule yields the least median delays, allowing build failures to be caught as early as possible. Commit grouping techniques and ML-CI both yield a similar number of builds scheduled to each other, and save similar a amount of time (build).

Table 20: Median % Time Saved by Build Saving Heuristics

Method	SmartBuildSkip	Batch4	BatchBisect				BatchStop4		**Median**
		4	2	4	8	16	8	16	
SmartBuildSkip	91.67	–	–	–	–	–	–	–	**91.67**
Commit Grouping	–	47.58	27.68	44.30	45.31	43.03	33.94	69.88	**44.30**
ML-CI	–	46.42	29.19	43.41	44.58	39.41	46.97	43.27	**43.41**
Timeout Rule	–	69.27	57.78	60.74	63.36	66.03	67.46	61.40	**63.36**

10.3 Ghost Failures in SmartBuildSkip

Fixing failing commits in CI is a tedious process. After recognising the occurrence of failing commits, the root cause of these failure needs to be identified. Once a build failure is identified, commit grouping techniques subsequently rerun the build on subsequences of commits until the culprit commits are found, making them adept at both recognizing that the build is failing and identifying their root causes. On the other hand, while SmartBuildSkip can recognize that the build is failing, it cannot detect the root cause of these failures. If SmartBuildSkip makes a false positive prediction (i.e., does not identify a failing commit), it causes a delay. These build failures remain in the CI pipeline as 'Ghost Failures' until rectified. Identifying these Ghost failures and buggy commits is an essential part of CI, one that will require additional builds to accomplish.

In our work, we are unable to fully fix this problem of SmartBuildSkip. Future work should focus on this.

10.4 Practical Adoption of ML-CI and Timeout Rule

While current CI servers do not offer build failure prediction mechanisms out of the box, leading CI services like TeamCity and Azure DevOps do provide support for advanced techniques like commit grouping heuristics. As such, we could learn from the way in which the latter are provided to CI users to understand how build failure prediction mechanisms could be integrated in the future.

In particular, most CI servers support a concept of "build trigger" (e.g., [11, 40]), which allows practitioners to configure and control when build pipelines are scheduled for a project. For example, Azure DevOps allows to set triggers to batch a sequence of commits by specifying a time interval during which the build waits for further commits before kicking off. Alternatively, flags in commit messages can also be used to prevent the CI server from including a commit in the builds [10].

Learning from these integrations, we believe one could build similar mechanisms to integrate MLCI and Timeout Rule into the CI environment to save builds. One technique could be to add to build configuration files a new type of build trigger allowing to specify a build failure prediction model to determine if a

build should be triggered or not for a given commit. Alternatively one could also imagine a prediction model to be run during code review of a pull request (similar to static analysis jobs running in parallel with code review). Upon prediction of a build pass, a flag could be inserted into the pull request's commit message to prevent the commit from being built. In both cases, the prediction model could be hosted in a web service, as is typical for many machine learning models. Furthermore, similar to traditional build triggers, humans could still manually override the build decision, if deemed necessary. As such, the technical building blocks to integrate advanced build skipping heuristics into CI environments are already available in practice.

10.5 Challenges and Recommendations to Practitioners

The build saving heuristics that we discussed in this paper provide beneficial CI cost savings in terms of energy, time and resources. However, this comes at the cost of increased median delays for developers who push their commits into the CI server. In the case of commit grouping heuristics, developers have to wait for sufficient incoming commits to fill an entire batch, while commit skipping algorithms may cause delays in failing build identification. In this study, we measure this delay in terms of the number of commits received into CI before the developer gets feedback on their commit.

The inflow of commits into a project depends on the number of developers in the project, its size, release cycle etc. Depending on these factors, delays are considered to be relative to the project. For example, a large project like JRuby may see 100 incoming commits in a few hours, but for a smaller project it may take 2 months to receive 100 commits. Depending on these factors of delay and costs, developers need to consider the trade-off that our hybrid heuristics will provide to decide which heuristic may be more beneficial to their work.

If practitioners set their primary goal to be build reduction, a commit skipping approach like SmartBuildSkip is a better option to adopt. However, if code quality and maintenance is a priority, more conservative approaches like commit grouping can be used. Hybrid heuristics like ML-CI and the Timeout Rule help to explore the trade-offs in between the two extremities of commit skipping and commit grouping techniques.

11 Threats to Validity

We discuss the major threats to the study validity in three categories: construct, internal, and external validity.

Construct Validity: Our study uses imbalanced TravisTorrent datasets that consist of data of projects with at least 2000 builds to test our proposed algorithms. These datasets may or may not be able to similar to the real-world CI scenarios. To mitigate this threat, we simulate the real-world practical application of build saving techniques by forming training and tests sets along the timeline of the studied products.

Additionally, we use metrics corresponding to build frequency and turn-around-time to measure the performance of our heuristics. While they may not accurately

demonstrate the amount of time saved by each heuristic during CI, they have been chosen due to their prior use by other developers and authors [5, 6, 24, 35].

Internal Validity: The features for predicting the build outcome have been chosen based on prior literature [19, 24]. However, software build systems evolve continuously and might be correlated with features that are currently not considered in the models. The performance of the model used for predicting build outcomes will impact the results of our proposed heuristics. However, as discussed in Section 9.3, these threats can be mitigated by adjusting the timeout value of the Timeout Rule and adjusting the batch size used in our heuristics.

Another threat to internal validity is caused by the case studies conducted using scripts by a single author. This could pose a threat regarding human biases and limitations that may have led to missing data, faulty scripts etc. This threat is mitigated by writing a test suite that checks the artifacts produced for the empirical study. We have also made the replication package available online [28].

External Validity: Our study evaluated the performance of proposed heuristics on 20 projects from the TravisTorrent dataset, where each project consists of more than 2,000 commits and has different balance ratios between failed and successful builds. The results of our study require follow-up analysis to understand their generalization to projects with less than 2,000 commits. Furthermore, results may vary with different permutations of project size and balance. Additionally, our study focused on the data extracted from TravisTorrent, which collects builds from open-source projects that use TravisCI. The results of our approaches may vary for proprietary projects and build data which would have to be analyzed separately.

We also note the many open source projects on GitHub that migrated their CI/CD platforms from TravisCI to Github Actions, making the data inspected in this project relatively older. We choose to study TravisCI due to its prevalence in existing studies [5, 6, 24]. Even though our heuristics are independent of the CI platform used (and could be easily integrated using a tool), Our choice of data may pose a threat to external validity as the performance of our heuristics may vary when used with newer CI/CD platforms. However, by examining all of the studied existing and newer heuristics on the same dataset, we are able to perform accurate comparisons of their performance, which can help developers draw reliable conclusions.

Additionally, it is important to mention that our hybrid heuristics are trained on past data of a project and tested on future data. However, data of historical CI outcomes may not be readily available in all organizations, making our technique less feasible and posing a threat to external validity. Additionally, practitioners will need to allow full build scheduling for some time to collect sufficient data for training models before they can switch to build saving heuristics. Cross-project testing, i.e., using the models trained on one project to predict the build outcomes of another project, could be analysed to accommodate the issue of lack of data availability.

Conclusion Validity: As our study compares multiple heuristics against each other, we had three or more distributions of the percentage of builds required and median delay for the same collection of commits. Hence, we used statistical tests to draw conclusions regarding the performance of our hybrid heuristics in comparison to existing baseline heuristics. We use the Kruskal-Wallis test supported with the Pairwise Wilcoxon test for post-hoc testing and recorded the p-values and effect sizes for each conclusion.

12 Conclusion

Despite its cost, continuous integration is an essential element in the development of a project. The earlier bugs are revealed within a project, the less expenses it imposes on the company, yet to achieve this level of quality assurance, CI consumes considerable resources. Reducing the number of builds scheduled can reduce this cost of CI, but the choice of what to build and what not to build remains a difficult decision. Heuristics like SmartBuildSkip and commit grouping, developed by different lines of research, each come with their own cost, in terms of percentage of builds required or delays.

While ML-CI can perform well for certain projects, its performance is mostly on par with that of traditional commit grouping algorithms, with only minor improvements in terms of grouping delay. More studies and experiments will have to be performed to improve this hybrid approach. On the other hand, the Timeout Rule can balance out the disadvantages of commit skipping by introducing a timeout threshold that "resets" the SmartBuildSkip model. The disadvantages of commit grouping are also greatly reduced by reducing the number of instances where commit grouping is used, leading to a reduction of 26.10% builds required. While SmartBuildSkip still required less builds, the Timeout Rule significantly reduced the risk of delays, while not conceding too many extra builds to be run.

Overall, our study introduces hybrid heuristics that combine two seemingly isolated lines of research with successful results, in particular for the Timeout Rule. More such heuristics could be explored in the future to improve even better trade-offs in terms of percentage of builds required and median delays of CI systems.

13 Conflict of Interest

All authors declare that they have no conflicts of interest.

14 Data Availability

The datasets generated during and/or analysed during the current study are available in the online replication package [28].

References

1. Abdalkareem, R., Mujahid, S., Shihab, E.: A machine learning approach to improve the detection of CI skip commits. IEEE Transactions on Software Engineering (TSE) **47**(12), 2740–2754 (2020)

2. Abdalkareem, R., Mujahid, S., Shihab, E., Rilling, J.: Which commits can be ci skipped? IEEE Transactions on Software Engineering **PP** (2019). DOI 10.1109/TSE.2019.2897300
3. Barrak, A., Eghan, E.E., Adams, B., Khomh, F.: Why do builds fail?—a conceptual replication study. Journal of Systems and Software **177**, 110939 (2021)
4. Basili, V., Rombach, H.: The TAME project: Towards improvement-oriented software environments. IEEE Transactions on Software Engineering (TSE) **14**(6), 758–773 (1988)
5. Bavand, A.H., Rigby, P.C.: Mining historical test failures to dynamically batch tests to save ci resources. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 217–226. IEEE (2021)
6. Beheshtian, M., Bavand, A., Rigby, P.: Software batch testing to save build test resources and to reduce feedback time. IEEE Transactions on Software Engineering (TSE) pp. 1–18 (2021)
7. Beller, M., Gousios, G., Zaidman, A.: Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. In: Proceedings of the 14th International Conference on Mining Software Repositories (MSR), pp. 356–367 (2017)
8. Cataldo, M., Herbsleb, J.D.: Factors leading to integration failures in global feature-oriented development: an empirical analysis. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 161–170 (2011)
9. Chen, B., Chen, L., Zhang, C., Peng, X.: Buildfast: History-aware build outcome prediction for fast feedback and reduced cost in continuous integration. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 42–53. IEEE (2020)
10. DevOps, A.: Skipping ci for individual pushes. URL `https://learn.microsoft.com/en-us/azure/devops/pipelines/repos/azure-repos-git?view=azure-devops&tabs=yaml#skipping-ci-for-individual-pushes`
11. DevOps, A.: Specify events that trigger pipelines. URL `https://learn.microsoft.com/en-us/azure/devops/pipelines/build/triggers?view=azure-devops`
12. Duvall, P.M., Matyas, S., Glover, A.: Continuous integration: improving software quality and reducing risk. Pearson Education (2007)
13. Elbaum, S., Rothermel, G., Penix, J.: Techniques for improving regression testing in continuous integration development environments. In: Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE), pp. 235–245 (2014)
14. Finlay, J., Pears, R., Connor, A.M.: Data stream mining for predicting software build outcomes using source code metrics. Information and Software Technology **56**(2), 183–198 (2014)
15. Fowler, M., Foemmel, M.: Continuous integration (2006)
16. Gallaba, K., Macho, C., Pinzger, M., McIntosh, S.: Noise and Heterogeneity in Historical Build Data: An Empirical Study of Travis CI. In: Proc. of the International Conference on Automated Software Engineering (ASE), p. 87–97 (2018)
17. Ghaleb, T.A., Da Costa, D.A., Zou, Y.: An empirical study of the long duration of continuous integration builds. Empirical Software Engineering **24**(4), 2102–2139 (2019)
18. Hassan, A., Zhang, K.: Using decision trees to predict the certification result of a build. In: Proceedings of the 21st International Conference on Automated Software Engineering (ASE), pp. 189–198 (2006)
19. Hassan, F., Wang, X.: Change-aware build prediction model for stall avoidance in continuous integration. In: Proceedings of the 11th International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 157–162 (2017)
20. Herzig, K., Greiler, M., Czerwonka, J., Murphy, B.: The art of testing less without sacrificing quality. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, p. 483–493. IEEE Press (2015)
21. Islam, M., Zibran, M.: Insights into continuous integration build failures. In: Proceedings of the 14th International Conference on Mining Software Repositories (MSR), pp. 467–470 (2017)
22. Jain, R., Singh, S.K., Mishra, B.: A brief study on build failures in continuous integration: Causation and effect. In: Progress in Advanced Computing and Intelligent Engineering, pp. 17–27. Springer (2019)
23. Jeffrey, D., Gupta, N.: Test case prioritization using relevant slices. In: Proceedings of the 30th International Computer Software and Applications Conference (COMPSAC), pp. 411–420 (2006)
24. Jin, X., Servant, F.: A cost-efficient approach to building in continuous integration. In: Proceedings of the 42nd International Conference on Software Engineering (ICSE), pp. 13–25 (2020)

25. Jin, X., Servant, F.: Cibench: a dataset and collection of techniques for build and test selection and prioritization in continuous integration. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 166–167. IEEE (2021)
26. Jin, X., Servant, F.: What helped, and what did not? an evaluation of the strategies to improve continuous integration. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 213–225. IEEE (2021)
27. Jin, X., Servant, F.: Which builds are really safe to skip? maximizing failure observation for build selection in continuous integration. Journal of Systems and Software **188**, 111292 (2022)
28. Kamath, D.: Replication package (2023). https://github.com/SAILResearch/replication-21-divya_kamath-build_avoiding_heuristics-code
29. Kerzazi, N., Khomh, F., Adams, B.: Why do automated builds break? an empirical study. In: 2014 IEEE International Conference on Software Maintenance and Evolution, pp. 41–50. IEEE (2014)
30. Kirinuki, H., Higo, Y., Hotta, K., Kusumoto, S.: Splitting commits via past code changes. In: 2016 23rd Asia-Pacific Software Engineering Conference (APSEC), pp. 129–136. IEEE (2016)
31. Kwan, I., Schroter, A., Damian, D.: 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), 307–324 (2011)
32. Learn", S.: "feature importances with a forest of trees". URL `https://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html`
33. Marré, M., Bertolino, A.: Using spanning sets for coverage testing. IEEE Transactions on Software Engineering (TSE) **29**(11), 974–984 (2003)
34. Meyer, M.: Continuous integration and its tools. IEEE Software **31**(3), 14–16 (2014). DOI 10.1109/MS.2014.58
35. Najafi, A., Rigby, P., Shang, W.: Bisecting commits and modeling commit risk during testing. In: Proceedings of the 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 279–289 (2019)
36. Ni, A., Li, M.: Cost-effective build outcome prediction using cascaded classifiers. In: Proceedings of the 14th International Conference on Mining Software Repositories (MSR), pp. 455–458 (2017)
37. Rausch, T., Hummer, W., Leitner, P., Schulte, S.: An empirical analysis of build failures in the continuous integration workflows of Java-based open-source software. In: Proceedings of the 14th International Conference on Mining Software Repositories (MSR), pp. 345–355 (2017)
38. Rehkopf, M.: What is continuous integration? URL `https://www.atlassian.com/continuous-delivery/continuous-integration`
39. Seo, H., Sadowski, C., Elbaum, S., Aftandilian, E., Bowdidge, R.: Programmers' build errors: A case study (at Google). In: Proceedings of the 36th International Conference on Software Engineering (ICSE), pp. 724–734 (2014)
40. TeamCity: Configuring vcs triggers. URL `https://www.jetbrains.com/help/teamcity/8.0/configuring-vcs-triggers.html#per-check-in-triggering`
41. Tomczak, M., Tomczak, E.: The need to report effect size estimates revisited. an overview of some recommended measures of effect size (2014)
42. Tufano, M., Sajnani, H., Herzig, K.: Towards predicting the impact of software changes on building activities. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), pp. 49–52. IEEE (2019)
43. Vassallo, C., Schermann, G., Zampetti, F., Romano, D., Leitner, P., Zaidman, A., Di Penta, M., Panichella, S.: A tale of CI build failures: An open source and a financial organization perspective. In: Proceedings of the 33rd International Conference on Software Maintenance and Evolution (ICSME), pp. 183–193 (2017)
44. Wu, Y., Zhang, Y., Wang, T., Wang, H.: An empirical study of build failures in the Docker context. In: Proceedings of the 17th International Conference on Mining Software Repositories (MSR), pp. 76–80 (2020)
45. Xia, X., Zhou, X., Lo, D., Zhao, X., Wang, Y.: An empirical study of bugs in software build system. IEICE TRANSACTIONS on Information and Systems **97**(7), 1769–1780 (2014)

46. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: A survey. Softw. Test. Verif. Reliab. **22**(2), 67–120 (2012). DOI 10.1002/stv.430. URL `https://doi.org/10.1002/stv.430`
47. Zhang, C., Chen, B., Chen, L., Peng, X., Zhao, W.: A large-scale empirical study of compiler errors in continuous integration. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 176–187 (2019)
48. Ziftci, C., Reardon, J.: Who broke the build? Automatically identifying changes that induce test failures in continuous integration at Google Scale. In: Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), pp. 113–122 (2017)
49. Zolfagharinia, M., Adams, B., Guéhénuc, Y.G.: Do not trust build results at face value-an empirical study of 30 million cpan builds. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 312–322. IEEE (2017)