Pragmatic Approaches to Schedule Less Builds in Continuous Integration

by

Divya Madhav Kamath

A thesis submitted to the

School of Computing

in conformity with the requirements for

the degree of Master of Science

Queen's University

Kingston, Ontario, Canada

August 2023

Copyright © Divya Madhav Kamath, 2023

Abstract

CONTINUOUS integration (CI) is an efficient practice used for code integration and testing of software products. Yet, for large software projects, developers push hundreds of commits to the main code base everyday. Ensuring the quality of all these commits involves building, i.e., compiling and testing, every incoming commit in the CI environment. At the same time, dozens of additional builds also need to be performed to ensure that the software can reliably be executed on all supported platforms, hardware and configurations. Hence, CI becomes an expensive process.

In this thesis, we study techniques that can help reduce the cost of CI by scheduling less builds. We study the existing approaches SmartBuildSkip, CI-Skip rules and Batching algorithms (Batch4, BatchStop4, BatchDivide4 and BatchBisect), then perform empirical studies to analyse the advantages and drawbacks of existing techniques and propose 3 novel heuristics based on them i.e., ML-CI, Timeout Rule and Lightweight

Dynamic batching. In our two large scale studies of 79,482 builds from 20 open-source projects and 286,848 commits from 50 open-source projects, we find that our novel heuristics can match the performance of state-of-the-art heuristics and even improve upon them by enhancing them with simple history based features.

In particular, we observe that by combining the prediction power of SmartBuildSkip with the ability of Batching algorithms to drastically reduce the number of scheduled builds during CI in our ML-CI and Timeout rule approach, we require 38.21% less builds than batching algorithms and 96% less turn-around-time than SmartBuildSkip. Moreover, we find that we can use simple and lightweight techniques to further enhance the performance of static batching algorithms by 4.3% to 11.9% and can also match the performance of the more complex state-of-the-art dynamic batching algorithm.

Acknowledgments

I cannot begin to express my thanks to Dr. Bram Adams, who along with guidance and supervision also provided me with encouragement and patience throughout the duration of my research. His kindness and empathy are my biggest motivators and have not only helped me come so far but also pursue the next steps of my academic career. I would like to express my deepest gratitude to Dr. Ahmed Hassan who has never wavered in his support and provided insightful suggestions. I have learnt a lot from Dr. Adams and Dr. Hassan and I will forever be indebted to them for setting a strong foundation of research in me and helping me navigate through the tough world of academia.

I would also like to thank my examination committee members for providing their valuable comments and feedback.

I want to express sincere thanks to my Professors, Dr. Ying Zou, Dr. Emad Shihad, Dr. Diego Costa and Dr. Ahmad Abdallatif for their guidance and support during

my research. My heartfelt appreciation goes to my Professor, Dr. Tanja Tajmel for her unwavering guidance and unparalleled support during some of the toughest days of my life. I am blessed to be a member of SAIL and MCIS, where I met the most amazing people who I am proud to call my friends. I am thankful to all my lab members for providing me with moral support, feedback, inspiration, invaluable insights, love and laughter throughout my M.Sc. journey. A very special thanks to friends Avinash, Varshini, Nihal, Nikhil and Jazlyn who have showered me with love and strength, without whom I definitely wouldn't be here.

I am deeply indebted to my family. My father Dr. Madhav Kamath and mother Bhagyalaxmi Kamath have helped me reach every opportunity and giving me the strength to reach my goals. My sisters Shubha Kamath, Kiran Kamath, and my brother-in-law Jayaram Shenoy for always believing in me and helping me aim higher and higher. My little nephews Shiv and Ved whose giggles, quirks and stunts are my biggest stress-busters and are my constant reminders that life is beautiful.

I would like to dedicate this thesis to Avinash, Roger, Laurie and Terry - the people I will never be able to repay, the ones who gave me my life back and have helped me overcome the most difficult days I have ever seen. Its their hard work and kindness that gives me energy and keeps me going.

Co-authorship

In all thesis chapters and related publications, my contributions are drafting the research plan; conducting the background study and related work; data collection; conducting the experiments and data analysis; performing statistical tests, plotting the graphs; and writing and writing and polishing drafts.

Earlier versions of the work in the thesis are under major revision/review as listed below:

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

   Divya M. Kamath, Eduardo Fernandes, Bram Adams, Ahmed E. Hassan, under major revision for the Journal of Empirical Software Engineering.

2. **Lightweight Dynamic Batching Algorithms for Continuous Integration**

   Divya M. Kamath, Bram Adams, Ahmed E. Hassan, under review for the Journal of Empirical Software Engineering.

Table of Contents

Abstract i

Acknowledgments iii

Co-authorship v

List of Tables viii

List of Figures ix

1 Introduction 1
- 1.1 Thesis Statement . . . 3
- 1.2 Thesis Overview . . . 3
- 1.3 Thesis Contributions . . . 6

2 Background and Related Work 8
- 2.1 Related Work . . . 8
- 2.2 Background . . . 15

3 On Combining Commit Grouping and Build Skip Prediction to Reduce Redundant Continuous Integration Activity 27
- 3.1 Introduction . . . 28
- 3.2 Hybrid Heuristics . . . 32
- 3.3 Data collection . . . 39
- 3.4 Study Results . . . 47
- 3.5 Threats to Validity . . . 77
- 3.6 Chapter Summary . . . 78

4 Lightweight Dynamic Batching Algorithms for Continuous Integration 80
- 4.1 Introduction . . . 81
- 4.2 Lightweight Dynamic Batching Algorithm and Variants (LWD) . . . 84
- 4.3 Data Collection . . . 92

4.4 Study Results 97
4.5 Threats to Validity 109
4.6 Chapter Summary 111

5 Conclusions and Future Work **112**
5.1 Thesis Contributions 113
5.2 Future Work 114

Bibliography **116**

List of Tables

2.1 Turnaround Time for skipping-based heuristics. 24

3.1 Commit-level Metrics 35
3.2 Studied Software Projects (ordered by #commits) 43
3.3 Percentage of Builds Required by Individual Heuristic 48
3.4 Results of ML-CI 54
3.5 Most Important Features of ML-CI 64
3.6 Results of Timeout Rule 67

4.1 Batch size update rules for 3 atomic and 2 composite LWD variants . . . 89
4.2 Variants of State-of-the-art Heuristics 95
4.3 p-values for for the LWD Batching Updating Variants (Table 4.4 shows the corresponding best factor value). Bold indicates statistical significance for $\alpha = 0.05$ 100
4.4 Best factor values for each combination of LWD Batching Updating Variant and fallback algorithm, as well as best LWD variant per fallback mechanism. Bold indicates the factor values that were statistically significantly better than other factor values according to Table 4.3, while other factor values were determined based on higher median performance values (no significant improvement). 100
4.5 p-values comparing Static Batching Variants(columns) against LWD Batching Variants(rows) (Table 4.4 shows the corresponding best factor value). Bold indicates statistical significance for $\alpha = 0.05$ 104
4.6 p-values of Wilcoxon Tests comparing LWD Batching Variants (rows) against Baseline Dynamic Batching Variants (columns). (Table 4.4 shows the corresponding best factor value). Bold indicates statistical significance for $\alpha = 0.05$. 106

List of Figures

2.1 This figure illustrates the number of builds required by the BatchBisect commit grouping algorithm. In the figure, red boxes signify failing builds and green boxes signify successful builds. Across the 8 commits, BatchBisect will require a total of 8 builds to be scheduled due to the bisection used for the failed second batch build. 16
2.2 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. 22
2.3 This figure illustrates the delays incurred during the SmartBuildSkip algorithm. 25

3.1 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. 37
3.2 Training process and data segregation. 42
3.3 Builds Required by Baseline Algorithms. 47
3.4 Boxplots of delays made per project by SmartBuildSkip algorithm. 51
3.5 Boxplots of the number of bad skips made per project by SmartBuildSkip. Projects are shown in the same order as Figure 3.4. 52
3.6 Boxplots demonstrating range of Percentage Build Required, Median Delays and the best performing batch sizes in ML-CI. 53
3.7 Boxplots of delays made per project by the ML-CI Batch4 variant. 55
3.8 Median Combined Delay in ML-CI (using Batch4) vs SmartBuildSkip algorithm. 59
3.9 Percentage of Required builds in ML-CI (using Batch4) vs SmartBuildSkip algorithm. 60
3.10 Median Combined Delay in ML-CI (using Batch4) vs. Baseline Batch4 Batching algorithm. 61
3.11 Percentage of Required builds in ML-CI (using Batch4) vs Baseline Batch4 Batching algorithm. 62

3.12 Boxplots demonstrating range of Percentage Build Required, Median Delays, and best timeouts to use in Timeout Rule. 66
3.13 Median combined delays made per project by SSR-Batch4 variant. 68
3.14 Median Combined Delay for the Timeout Rule (using Batch4 and best performing timeout threshold for that project) vs SmartBuildSkip algorithm. 71
3.15 Percentage of Required builds for the Timeout Rule (using Batch4 and best performing timeout threshold for that project) vs SmartBuildSkip algorithm. 72
3.16 Median Combined Delay in the Timeout Rule (using Batch4 and best performing timeout threshold for that project) vs Baseline Batch4 Batching algorithm. 74
3.17 Percentage of Required builds in he Timeout Rule (using Batch4 and best performing timeout threshold for that project) vs Baseline Batch4 Batching algorithm. 75
3.18 Percentage of Builds Required v/s Median Delay for each timeout threshold of the Timeout Rule for JRuby 76

4.1 Comparing the performance of the LWD variants, with each boxplot covering the distribution of percentage of builds saved across all factor values and projects. 98
4.2 Performance of all the lightweight batch updating techniques. 101
4.3 Comparison of the performance of the best incarnation of our LWD variants against all variants of Static Batching techniques 103
4.4 Comparing the performance of LWD variants against the baseline Dynamic Batching techniques 105
4.5 Comparing the performance of Random Batch Size Updating variants against the baseline Dynamic Batching techniques 106
4.6 Comparing the performance of Lightweight Dynamic Batching technique with and without CI-Skip rules 108
4.7 Comparing the performance of Baseline Static Batching (batch size 16) technique with and without CI-Skip rules 109
4.8 Comparing the performance of Baseline Dynamic Batching technique with and without CI-Skip rules 110

CHAPTER 1

Introduction

CONTINUOUS integration (CI) is a code maintenance and quality assurance practice that periodically integrates code changes from multiple contributors into the current snapshot of the code base. One of the main functions of a CI tool (like GitHub Actions and TravisCI) is ensuring code correctness by performing automated builds (i.e., compilation and testing) of every commit merged into the shared code base. For large software products, this means confirming proper software functioning on all the supported platforms, configurations and hardware. As a result, a full CI process from pushing a commit, to merging and testing it can take hours to complete. Ultimately, CI is an expensive process eating time, energy and computing resources reported to cost millions of dollars for companies like Google and Mozilla (25; 46).

CI engineers and researchers have since introduced techniques to help reduce the

cost of CI. Some techniques save cost by speeding up the scheduled CI builds like test case selection and test case prioritization. For example, Elbaum et al. (13) presented techniques that speed up CI testing during scheduled builds by tracking the past results of executed test suites, then applying test selection and priorization techniques to reduce the number of tests to run on a pull request during CI (pre-merge), and prioritize the full test suite run after merging a pull request in the code base.

Other techniques like commit batching and commit skipping save costs by reducing the number of builds scheduled during CI itself. Jin et al. (29) presented commit skipping techniques like SmartBuildSkip, PreciseBuildSkip and HybridCISave that use prediction models to reduce the number of builds scheduled for CI testing. Abdalkareem et al. (2) also presented CI-Skip rules to identify commits that can be skipped, i.e., do not trigger a CI build. Beheshtian and Bavand et al. (6; 7) presented commit batching techniques that can group multiple commits scheduled to be built together. In this thesis, we focus on the two latter groups of heuristics, i.e, commit skipping and commit batching.

However, any technique comes with its own limitation and risk. Some existing build scheduling heuristics limit their usage based on restrictions on programming language, historical software quality or software size. Furthermore, heuristics that use machine learning impose additional costs for data collection, model training and prediction errors. These costs limit the use of these existing heuristics by smaller companies that need to use CI. In this thesis, we particularly study simple, yet effective, history-based features that can enhance existing heuristics and further reduce builds scheduled by them. We introduce three novel heuristics, namely ML-CI, Timeout Rule

and Lightweight Dynamic batching that augment existing commit skipping and commit batching techniques. We empirically compare the performance of our novel heuristics to the state-of-the-art heuristics.

With our study, our goal is to provide insights to build engineers about existing and new build reduction heuristics, along with their strengths and limitations.

1.1 Thesis Statement

The use of simple history-based features can reduce the number of builds required by existing commit skipping and commit batching heuristics.

1.2 Thesis Overview

1.2.1 Chapter 2: Background and Related Work

In this chapter, we discuss the algorithms along with the theory behind existing build reduction heuristics. We provide an overview of SmartBuildSkip, Static Batching algorithms like BatchBisect, BatchStop4, Batch4, BatchDivide4, Dynamic Batching, and CI-Skip rules. We also present related work on the study of build failures and on techniques used to speed up CI.

1.2.2 Chapter 3: On Combining Commit Grouping and Build Skip Prediction to Reduce Redundant Continuous Integration Activity

In this thesis, we bring together two separate lines of research, i.e., machine learning based commit skipping (SmartBuildSkip algorithm) and static commit batching (BatchBisect, Batch4 and BatchStop4). These two families of heuristics each help to reduce the CI cost by reducing the number of builds scheduled during CI. On the one hand, SmartBuildSkip is a powerful prediction model that drastically reduces the number of builds required for a project by skipping commits whose build is predicted to be successful. On the other hand, Batching algorithms have a better turn-around-time, i.e., they can detect failing builds faster and deliver build results to developers more efficiently by executing builds once for a batch of successive commits. Given the strengths and weaknesses of both, this chapter proposes two hybrid heuristics - ML-CI and Timeout Rule that can combine the two families and improve turn-around-time as well as reduce number of builds scheduled in CI.

In both our hybrid heuristics, SmartBuildSkip predicts the outcome of the build of an incoming commit. If the model predicts the build to be successful, no build is scheduled. A build is scheduled only if the model predicts the build to be unsuccessful. We introduce a simple history-based 'timeout' metric that keeps track of the length of the most recent sequence of skipped commits during CI, with the assumption that finding a balance in the number of builds skipped (i.e., finding the optimal value of 'timeout') can help to cut the shortcomings of both families of heuristics. In ML-CI, the timeout metric and other commit-related metrics (like size, number of comments etc.) are used to train the model that predicts the outcome of a build. On the other

hand, in the Timeout Rule, the 'timeout' feature is used to override the decision of the SmartBuildSkip algorithm and schedule a build regardless of its prediction.

In this chapter, we perform an empirical study on 79,482 builds from 20 large open-source projects to identify the turn-around-time and percentage of builds required by the new and existing heuristics. We find that the performance of the ML-CI model is similar to that of the commit batching baseline algorithms, while the Timeout Rule can schedule 38.21% less builds than it. We also note that both ML-CI and the Timeout Rule can reduce the turn-around-time by 96% in comparison to SmartBuildSkip, while the latter schedules the lowest percentage of builds of all techniques. Our hybrid approach can provide a trade-off between turn-around-time and build frequency without sacrificing the quality of the merged code.

1.2.3 Chapter 4: Lightweight Dynamic Batching Algorithms for Continuous Integration

In Chapter 3, we presented ML-CI and the Timeout Rule, i.e., two hybrid heuristics that combined build outcome prediction and commit batching. While they found success, particularly the Timeout Rule, we aim to investigate simpler solutions, since effective machine learning-based solutions add the overhead of data collection, model training, re-training and maintenance. In this chapter, we focus on simpler rule based techniques that are motivated by prior studies and observations on CI builds. In particular, our efforts here are directed towards the state-of-the-art commit batching techniques.

Currently, there exist two types of commit batching techniques - static batching

(where the number of commits grouped into a batch is constant) and dynamic batching (where a variable number of commits is batched for each scheduled build). Although static batching is relatively simple to use, it is not reliable for projects with higher than 25% historical failure rate (44). On the other hand, existing dynamic batching techniques are complex - they rely on calculations of weighted failure rates by keeping track of all previous build outcomes, and require an offline simulation to predict ideal batch sizes to be used for each corresponding failure rate.

In this chapter, we introduce Lightweight Dynamic Batching (LWD) algorithms - simple techniques that can dynamically adapt the batch size required during commit batching to provide optimal build savings. Batch sizes are updated according to a simple history-based metric, i.e., the failure rate of the most recent batch built in CI. We introduce 5 main variants of LWD, each of which can be customized according to the needs of the project. By varying the customizable parameters of the LWD variants, we introduce and evaluate 13 sub-variants of LWD in this study.

An empirical study on 286,848 commits from 50 open- source projects shows that our LWD techniques perform equally well as the more complex state-of-the-art dynamic batching techniques and also outperform static batching algorithms by saving a median 7.73% more builds.

1.3 Thesis Contributions

In this thesis, we empirically study pragmatic heuristics to reduce the number of builds scheduled in CI using the TravisCI dataset. We introduce 3 novel heuristics towards this goal and empirically compare the performance of our techniques to existing state-of-the-art heuristics. Our main contributions in this thesis are as follows:

1. We introduce 3 novel heuristics - ML-CI, Timeout Rule and Lightweight Dynamic Batching to help reduce the builds scheduled in CI.

2. In chapters 3 and 4, we evaluate and combine two different lines of research, i.e., commit skipping and commit batching heuristics, and empirically evaluate their strengths and weaknesses.

3. We perform extensive empirical evaluations of our novel heuristics and the baseline heuristics and compare the performance of these techniques against each other.

4. We offer insights to build engineers about diversity and scope of build avoidance heuristics and how they can be used to help reduce CI costs.

5. Our replication packages containing the scripts and data for our novel heuristics are made available online (33; 34).

CHAPTER 2

Background and Related Work

THIS section revisits the static and dynamic algorithms proposed in prior work, which we use as baselines in this thesis.

2.1 Related Work

2.1.1 Study of build failures

The study of builds and build outcome prediction began in the early 2000s. Hassan et al. (20) 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. (38) identified that higher social co-ordination in a project correlates with more build failures for certain build

types. Cataldo et al. (10) studied the impact of project attributes and cross-project feature interactions on software integration failures. Similarly, Islam et al. (26) 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 (36) are negatively correlated with build failures.

Researchers also used an alternate approach of monitoring and sending pre-emptive alerts to developers when their changes can impact build duration and/or build failures of future build activities (52). In parallel, Zhang et al. and Seo et al. (49; 58) analyzed the frequency of compiler errors that cause build failures and the corresponding fix times and fix patterns. Kerzazi et al. (36) 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. (59) 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. (19) 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. (56) analyzed 4 major build systems to study and categorize their bugs and fixes. Vassallo et al. (54) 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. (27), who inspected the effects of a large team size on build failures. Rausch et al. (47), analyzed build failures from 14 OS Java projects and identified error categories and noisy build data, while Zolfagharinia et al. (60) studied

the impact of OS changes and environments on build failures. Finally, Wu et al. (55) 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.

2.1.2 Speeding up Continuous Integration (CI)

Continuous Integration is the process of integrating and testing commits pushed by multiple contributors (12). Large software companies like Google and Facebook have a massive number of commits to compile everyday. As mentioned earlier, CI can be expensive (26). To reduce CI costs, researchers introduced techniques that tackle this problem from different angles, such as reducing the number of builds to be scheduled, speeding up scheduled builds using faster build systems and architectures, and reducing the number of test cases to be run. These techniques include the widely popular Test Case Prioritization (TCP), Regression Test Selection (RTS), build priorization, commit skipping and commit batching. Broadly, these studies can be categorized into two approaches: (1) approaches that reduce the number of builds scheduled during CI and (2) approaches that speed up test suite execution within scheduled CI builds.

Speeding up Scheduled Builds

Speeding up test suites covers a wide range of test case prioritization and test case selection techniques. In test case prioritisation technique, 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 selected prior to testing.

Various authors have explored approaches for Test Case Prioritization (TCP) and Regression Test Selection (RTS), aiming to reduce test duration in CI (8; 13; 14; 22; 40; 50). Kazmi et al. (35) presented a literature survey of 47 empirical studies of RTS techniques like learning-based regression and mining, their study design and the cost models of studied RTS techniques. Elbaum et al. present cost-effective techniques that use time windows to track executed test suites and apply RTS and/or TCP during CI testing. Jin et al. (30; 31) also evaluated 10 selection and prioritization strategies on build and test level granularities to understand design decisions that could help to save the cost of CI. The effectiveness of these techniques are explored extensively in literature (13; 28; 41; 57). In parallel, companies like Microsoft, Facebook and Google have also introduced distributed build tools like Buck, CloudBuild (15), Bazel (42) that reduce build duration by performing incremental builds.

In recent years, the study of build outcome prediction has further evolved as Finley et al. (16) 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. (21) 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. (45) and Barrak et al. (4) who used cascaded classifiers and multivariate models, respectively, to predict build outcomes. Another

work (9) on builds performed on Travis CI[1] 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 does not focus on speeding up scheduled builds, but on reducing the number of builds scheduled altogether. For that, other factors such as the time elapsed since previous scheduled builds and commit factors are investigated, 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.

Reducing the Number of Scheduled Builds

Overall, there are three broad techniques introduced to reduce the volume of builds scheduled 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.

Commit skipping is a build scheduling technique trying to determine which builds are useful to schedule, and which ones are redundant (i.e., should not be scheduled) during CI testing. After identifying 1,813 commits that were manually labeled by developers to be skipped from CI testing, Abdalkareem et al. (3) presented CI-Skip rules able

[1]https://travis-ci.org/

to identify commits for which a build is redundant, then presented a rule-based technique to automatically detect and label CI-Skip commits. Furthering their study, Abdalkareem et al. (1) presented a machine learning approach to improve the detection of CI-Skip commits. This study identified that the number of developers, the content of commit messages, along with the CI-Skip rules are important factors for detecting CI-Skip commits. They extracted 23 features from the historical data of 10 software repositories and built a decision tree classifier to detect these commits. Jin et al. (32) further studied the cost-saving ability and the safety of CI-Skip rules with respect to the proportion of failing builds that they can identify and determined that the rules are not completely safe to use, as they skip some failing builds too. They further presented a safer, complementary collection of CI-Run rules that are used in their novel PreciseBuildSkip technique to predict build failures.

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. 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. The basic principle combines together a finite number of individual commits into a single build/test. Only if the build/test fails, the commit(s) responsible for the failure must be identified through additional builds based on a so-called fallback algorithm. Hence, commit batching requires choosing a batch size (i.e., finite number of commits) and a fallback algorithm (to identify culprit commits).

Najafi et al. (44) presented a study on identifying patterns of commit grouping and bisection. They identified that batching and bisection are effective when performed

in batch sizes of up to 9 builds. Beheshtian et al. (7) presented and examined 4 commit bisection techniques and 2 risk modeling techniques to perform batching in CI. Furthermore, Bavand et al. (6) presented a study on dynamic commit batching, which uses a non-trivial mathematical model to determine and evolve the ideal batch size to be used during CI. In Chapter 4, we explore and evaluate the use of simpler, arithmetic and random operations to dynamically determine batch sizes to be used during CI. We compare our study results against the above works to provide build engineers with an in-depth insight into commit batching and bisection.

Chen et al., (11) 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. (29; 30; 31; 32) 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 (31), 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 Chapter 3, we focus on build reduction 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 into batches, but our prediction models choose which commits need to be built, then proceeds to build them in a batch. On the other hand, we avoid the use

of machine learning models in Chapter 4, and instead apply the CI-Skip rules along with our proposed dynamic batching techniques to examine if the two lighterweight rule-based techniques can be combined to achieve better build savings.

2.2 Background

This section defines more formally the algorithms and the rationale behind the state-of-the-art techniques used as baselines in this thesis.

2.2.1 TestAll Baseline

TestAll is a performance measuring baseline used by Bavand et al. (6) and Beheshtian et al. (7) that we follow in this work as well. In TestAll, all incoming commits are compiled individually during CI, without any batching. Since TestAll does not save any build at all, we measure the percentage of builds saved by the studied heuristics with reference to TestAll - i.e., Heuristic A saves X% of builds w.r.t TestAll.

2.2.2 Commit Batching Techniques

Static Batching

During static batching, a finite number of incoming commits are combined into a single batch for build execution (7), see Algorithm 1. If a batch of 'N' commits compiles successfully, it saves N-1 builds from being executed. However, if the batch build is unsuccessful, fallback algorithms like BatchBisect (Figure 2.1), BatchStop4, BatchDivide4, etc. are used to identify the culprit commit(s). Algorithms 2-4 summarize the steps of these studied fallback algorithms.

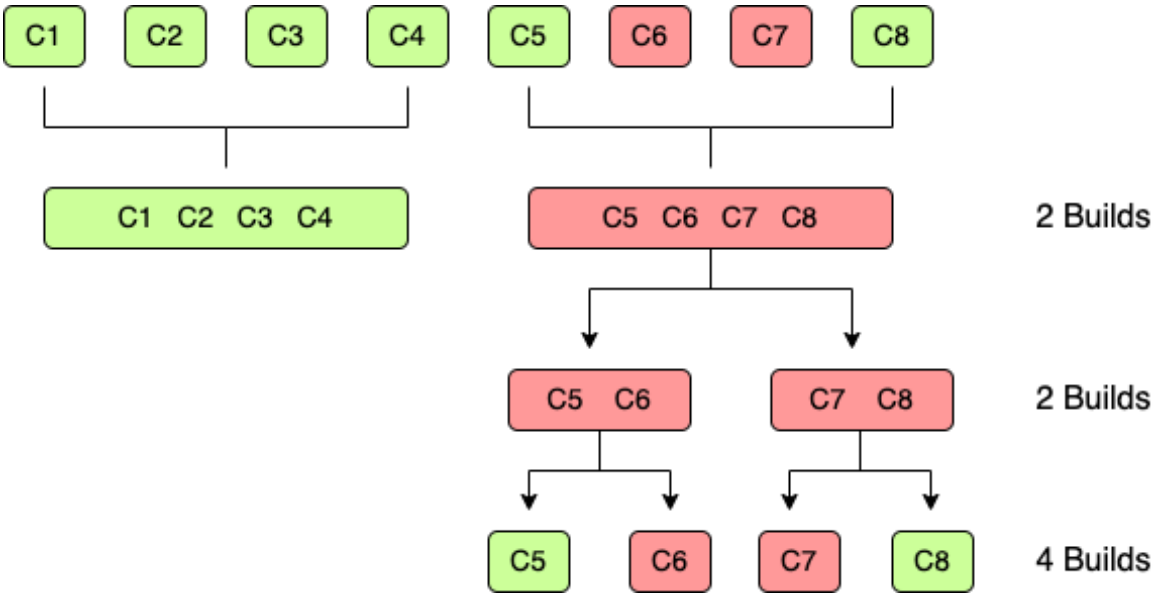


Figure 2.1: This figure illustrates the number of builds required by the BatchBisect commit grouping algorithm. In the figure, red boxes signify failing builds and green boxes signify successful builds. Across the 8 commits, BatchBisect will require a total of 8 builds to be scheduled due to the bisection used for the failed second batch build.

In BatchBisect, culprit commits are identified by halving the batch size (bisection) until a batch size of 1 commit is reached for failing sub-batches. However, with BatchStop4 and BatchDivide4, bisection is performed only until the batch size of 4 is reached, after which all 4 commits of a failing sub-batch are built individually to reveal any failing commits. BatchDivide4 further generalizes BatchStop4, by first grouping one batch into 4 groups of similar sizes, then implementing BatchStop4 on each batch to reveal any culprit commit(s).

The largest drawback with static batching is the bisection overhead when a compiled batch consists of many failing commits. Since build failures are expected to occur consecutively in CI, intuitively, a stream of failing commits are better compiled individually to avoid the overhead of bisection builds. For instance, let's assume a worst case scenario involving a batch of 8 failing commits. To identify that all commits in

Algorithm 1 Batch* Driver

```
1:  procedure FILLBATCH(batchSize, incomingCommit)
2:      i ← 0
3:      batch ← []
4:      fails ← []
5:      while i < batchSize do
6:          batch.append(incomingCommit)
7:          i ← i + 1
8:      end while
9:      outcome ← BatchBisect(batch, fails)          ▷ or other algorithms
10: end procedure
```

Algorithm 2 BatchStop4 (Beheshtian et al. (7))

```
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:         return
11:     end if
12:     outcome ← Build(batch)
13:     if outcome == False then
14:         half ← len(batch)/2
15:         BatchStop4(batch[:half])
16:         BatchStop4(batch[half:])
17:     end if
18: end procedure
```

Algorithm 3 BatchBisect (Beheshtian et al. (7))

```
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
```

Algorithm 4 BatchDivide4 (Bavand et al. (6))

```
1:  procedure BATCHDIVIDE4(batch, fails)
2:      if len(batch) <= 10 then
3:          return BatchStop4(batch)
4:      else
5:          limit ← len(batch)÷4
6:          batch_1 ← batch[ : limit]
7:          outcome ← BatchStop4(batch_1)
8:          batch_2 ← batch[limit : 2*limit]
9:          outcome ← outcome & BatchStop4(batch_2)
10:         batch_3 ← batch[2*limit : 3*limit]
11:         outcome ← outcome & BatchStop4(batch_3)
12:         batch_4 ← batch[3*limit : ]
13:         outcome ← outcome & BatchStop4(batch_4)
14:     end if
15: end procedure
```

the batch are failing, BatchBisect would require to make 15 build executions (see Figure 2.1). BatchDivide4 and BatchStop4 would require 11 builds each. This is an overhead of $15-8=7$ and $11-8=3$ builds, respectively.

Dynamic Batching

Dynamic Batching, introduced by Bavand et al. (6) uses a weighted moving average of build failures to dynamically estimate an optimal batch size that could be used to perform commit batching during CI. The $WeightedFailureRate$ of the last 'C' commits is computed using:

$$WeightedFailureRate = \frac{\sum_{c=1}^{C} Smoothing(c) * IsFailure(c)}{\sum_{c=1}^{C} Smoothing(c)} \quad (6)$$

Here, 'c' is the position of a commit relative to the current commit, while $Smoothing$ is a function that smoothes the weights for each commit. Examples of such a function are 1, $log(c)$, e^c or $\frac{1}{c}$, most of which ensure that newer commits have a higher weight than older commits. $IsFailure$ is a predicate indicating whether commit 'c' is a failing build.

Algorithm 5 describes the steps of the state-of-the-art dynamic batching technique. The technique uses an offline simulation of 100k randomly generated pass/fail occurrences for a range of failure rates between (0, 1) and for varying batch sizes in order to identify the expected number of build executions for each measured combination of build failure rate and batch size. Using this data, the *LookupBatchSize* function is able to return the batch size that expects the minimum number of executions (according to the simulation), for a provided build failure rate. Hence, once the CI and batching process begin, the function *WeightedFailureRate* first computes the build failure rate for a finished batch, using the formula provided above, after which the optimal batch

size can then be determined using *LookupBatchSize*.

Once the dynamic batching algorithm forms batches and schedules a build, it relies on the basic fallback algorithms BatchBisect, BatchStop4 or BatchDivide4 to identify culprit commits (if any). Hence, there exist three variants of this technique, each using one of the three fallback algorithms.

This state-of-the-art dynamic technique, while an excellent heuristic saving a median of 43.26% builds, depends on complex calculations such as weighted failure rates and offline simulation of expected values. The technique also allows little flexibility for build engineers to tweak the parameters of the algorithm. As such, we aim to eliminate this dependency and present lightweight techniques that can provide similar build saving outcomes.

Algorithm 5 State-of-the-art Dynamic Batching, (Bavand et al. (6))

```
1:  procedure FILLBATCH
2:      commit_outcomes ← [ ]
3:      i ← 0
4:      while incomingCommit do
5:          while i < batchSize  do
6:              batch.append(incomingCommit)
7:              i ← i + 1
8:          end while
9:          BatchBisect(batch, fails)                    ▷ or BatchStop4 or BatchDivide4
10:         commit_outcomes.extend(fails)
11:         failure_rate ← WeightedFailureRate(Smoothing, commit_outcomes)
12:         batchSize ← LookupBatchSize(failure_rate)
13:         i ← 0                                        ▷ Resetting for next batch
14:         batch ← [ ]
15:         fails ← [ ]
16:     end while
17: end procedure
```

2.2.3 Skipping Heuristics

SmartBuildSkip

The SmartBuildSkip algorithm introduced by Jin et al. (29) 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.

The algorithm (see Algorithm 6) 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 2.2 illustrates the SmartBuildSkip algorithm on the same data set as Figure 2.1. 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 SmartBuildSkip 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

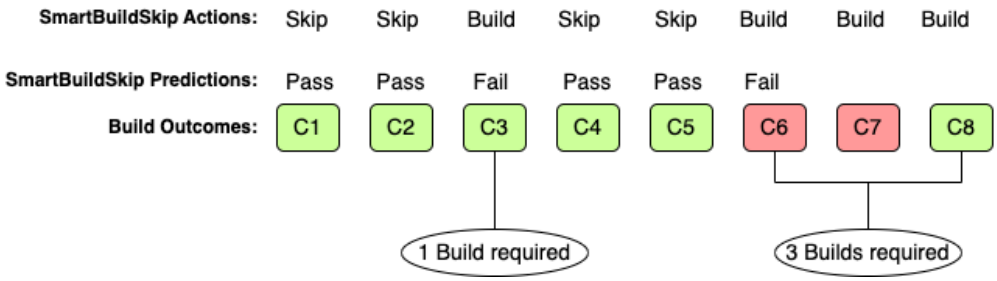


Figure 2.2: This figure illustrates the number of builds required in the SmartBuildSkip algorithm. In the figure, red boxes signify failing builds and green boxes signify successful builds.

commit C6 to fail, a build is run, which reveals an unsuccessful build. This prompts the algorithm to switch to the deterministic phase 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 predicting 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.

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 2.3 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

Algorithm 6 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
```

C4 and C10). When these commits are built, the previous build failures are also caught. In this thesis, we refer to these delays as "skipping-based delays". Such delays are also discussed by Jin et al. (29), 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 turnaround time varies according to the actual build outcome of a given commit, as shown in Table 2.1. 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 turnaround time equals the actual build duration. Only when a pass prediction was correct, no delay is incurred.

Table 2.1: Turnaround 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 (37), 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.

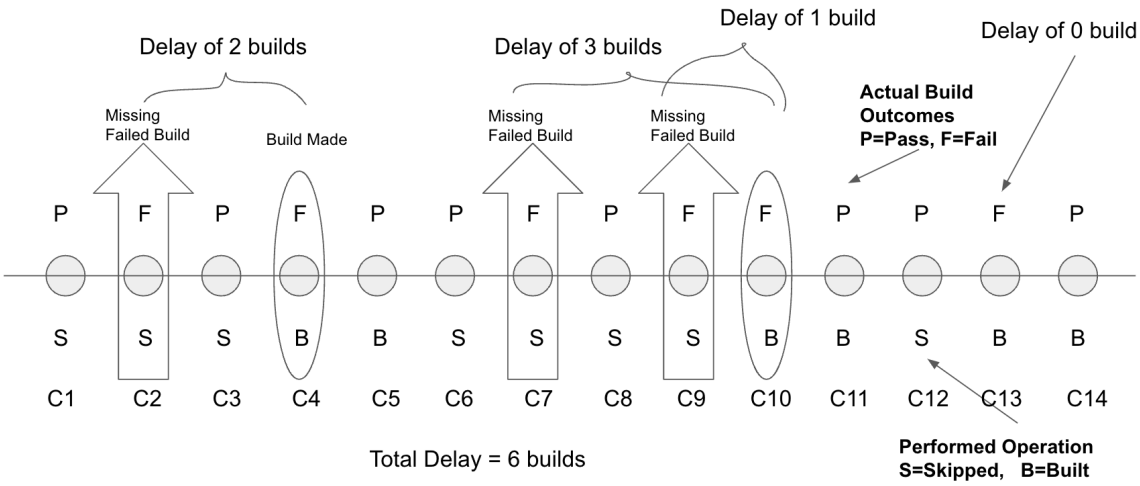


Figure 2.3: This figure illustrates the delays incurred during the SmartBuildSkip algorithm.

CI-Skip Rules

The concept of CI-Skip rules is based on the premise that not every commit needs to trigger the CI process (3). Examples of such commits include commits that only modify source code comments or commits that only modify the documentation of a project. Abdalkareem et al. introduced CI-Skip rules after examining 1,813 Java commits that developers explicitly marked to be skipped from the CI processes. These commits are hence named CI-Skip commits. The rule-based technique has 5 rules to detect CI-Skip commits (3):

- Changes that touch only documentation and non-source code files
- Changes related to preparing releases (version preparation) such as commits modifying version numbers in build configuration files
- Changes that only modify source code comments

- Changes that touch meta files such as .ignore files or image/media files like .png, .mp3
- Changes that format source code

Abdalkareem et al. (1) later presented a machine learning approach to detect CI-Skip commits, which remains outside the scope of this study as we are dedicated to the evaluation of rule-based techniques.

CHAPTER 3

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

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 38.21% 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.

3.1 Introduction

Continuous Integration (CI) is the practice of periodically integrating code changes from multiple contributors into a single code base (Rehkopf"), 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 (17). 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 (23). On the one hand, a large number of commits are produced daily (43). 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; 18; 60).

On the other hand, previous studies (29; 31; 32) 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 (31) and combining (44) 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 (7), 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; 29) 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 thesis, 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 (7) – i.e., Batch-Bisect, Batch4, and BatchStop4 – and one skipping heuristic, i.e., SmartBuildSkip (29). 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 commit grouping metrics 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:

- With respect to the grouping heuristics, we found that they can increase the number of builds required in the CI system by a median of 52.94%, in comparison to skipping heuristics.
- Regarding the skipping heuristics, our results suggest that they can induce a median delay of 53.5 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%.
- Not only can hybrid heuristics maintain the low turn-around-time of grouping-based heuristics, the Timeout Rule also reduces the number of builds required by a median of 36.47%.
- Our study's replication package is available online (33).

We organized the remainder of this chapter as follows. Section 3.2 introduces our hybrid heuristics. Section 3.3 introduces our study methodology, including the study goal, research questions, and study steps. Sections 3.4.1, 3.4.2, and 3.4.3 present our study results per research question. Section 3.5 discusses threats to the study validity. Finally, Section 3.6 concludes this chapter and suggests future work.

3.2 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 turnaround time. 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. (29)).

3.2.1 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 turnaround time.

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 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 7 ML-CI Model

```
predictor ← RandomForestModel
build_queue ← []
num_passes ← 0
Grouping ← False
while True do
    commit ← incoming_commit
    if Grouping is True then
        build_queue.append(commit)
        if len(build_queue) is batch_size then
            BatchStop4(build_queue)                    ▷ See Algorithm 2
            Clear(build_queue)
            Grouping ← False
            num_passes ← 1
        end if
    else
        outcome ← predict(commit ∪ num_passes)
        if outcome is True then
            num_passes ← num_passes + 1
            continue
        else
            num_passes ← 0
            build_queue.append(commit)
            Grouping ← True
        end if
    end if
end while
```

Table 3.1: Commit-level Metrics

Metric	**Description**
Patch Size	The 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.
#Lines changed in test files	The number of lines modified in test files. We use this metric since test files are less likely to contain bugs than source code
#Times code file is changed	The number of times a source code file is modified. If a file is changed often, its possibility of being error-prone is higher
#Files added in PR	The 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 deleted in PR	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
#Commits since last build scheduled (Timeout)	The 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
#Commits in the PR	The number of commits in the PR. The more commits in a PR, the more need for testing

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.

3.2.2 Timeout Rule

In the same vein as Jin et al. (29), 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. Instead of using the timeout metric to predict the build outcome, it is used external to the SmartBuildSkip 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.

Algorithm 8 shows the Timeout Rule, and is illustrated in Figure 3.1, 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

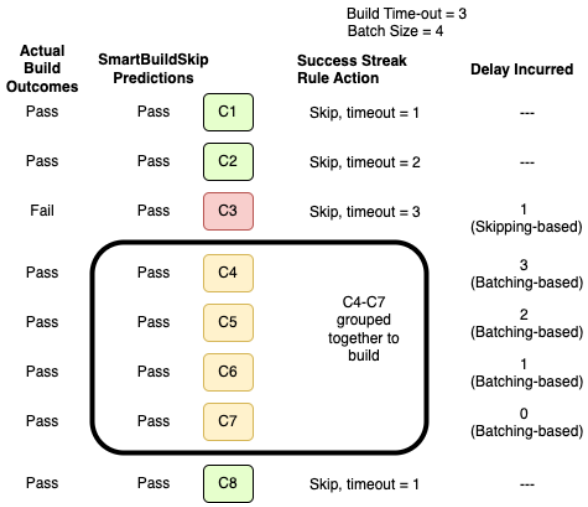


Figure 3.1: Illustration of the Timeout Rule meta-heuristic in a worst case scenario for SmartBuildSkip (no failure predictions). In the figure, red boxes signify failing builds, green boxes signify successful builds and yellow boxes signify the grouped builds.

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.

By introducing the Timeout Rule, we aim to reap the benefits of SmartBuildSkip 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

Algorithm 8 Timeout Rule

```
predictor ← SmartBuildSkip
build_queue ← []
timeout ← 0
Grouping ← False
while True do
    commit ← incoming_commit
    if timeout > threshold then
        build_queue.append(commit)
        if len(build_queue) is batch_size then
            batch_outcome ← Build(build_queue)
            Grouping ← (batch_outcome) ? False : True
            timeout ← 1
        end if
    else
        if Grouping is True then
            build_queue.append(commit)
            if len(build_queue) is batch_size then
                (batch_outcome) ← Build(build_queue)
                Grouping ← (batch_outcome) ? False : True
                timeout ← 1
            end if
        else
            outcome ← predict(commit)
            if outcome is True then
                timeout ← timeout + 1
                continue
            else
                Clear(build_queue)
                build_queue.append(commit)
                Grouping ← True
            end if
        end if
    end if
end while
```

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.

3.3 Data collection

3.3.1 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 (5) 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 (29).

In particular, we evaluate the two hybrid heuristics introduced in Section 3.2, i.e., ML-CI and Timeout Rule, and compare them to the Grouping and Skipping Heuristics

in terms of the following research questions:

RQ1: *At the commit level, how do the baseline batching and skipping heuristics perform compared to each other?* – We use this question to empirically identify areas of improvement of individual heuristics in terms of the number of builds skipped and the delay until a failure is observed. To our knowledge, this is the first study where these two lines of work are compared to each other.

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.

3.3.2 Data Set

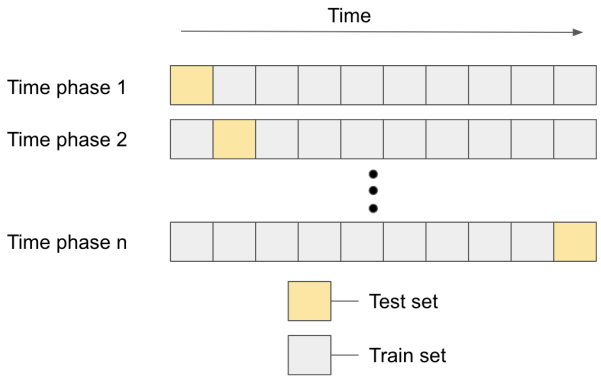
The dataset for this chapter is extracted from the replication package of the SmartBuildSkip algorithm (29). In their study, Jin et al. (29), selected 359 projects out of the 1,359 projects having CI data available on Travis Torrent. They only selected projects that are more than one year old, with at least 200 builds and at least 1,000 lines of source

code. Beheshtian et al. (7) 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%.

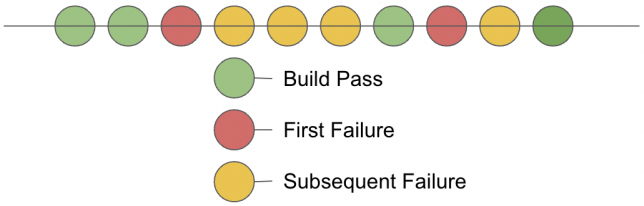
In our study, from the 1,277 projects in the replication package of SmartBuildSkip (29), we select those projects that consist of at least 2,000 commits, in order to analyse the trends of the four models. We do not limit our selection based on the failure rate of the project in order to identify the ramifications of our approaches on all kinds of projects. Eventually, our study resulted in a dataset of 20 projects.

3.3.3 Evaluation Methodology

Our choice for at least 2,000 commits allows our study to adequately simulate the behaviour of each heuristic. We perform stratified cross-validation by dividing each project's dataset into 10 equal time frames. We run 10 iterations of training by choosing one time frame as the test dataset named TE, and the rest of the data as the training dataset named TR. The dataset segregation procedure is depicted in Figure 3.2a. Following Tantithamthavorn et al. (51), who identified that out-of-sample bootstrap is the least biased model validation technique, we use bootstrap during the model training phase in each of our 10 iterations. With bootstrap, for each training dataset, we further run 100 iterations of model training, in each of which the model is trained using a training set TR' obtained by extracting a random sample from the training dataset TR. We use the remainder of the training dataset (TR - TR') as the validation dataset for each iteration. Once the 100 models are obtained, the model with the best F1-score is selected and then used for build outcome prediction on that iteration's test dataset TE.



(a) 10-fold cross-validation, with bootstrap performedon each iteration's training set..



(b) Segregation of build outcomes

Figure 3.2: Training process and data segregation.

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. (29) 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 3.2b.

For each heuristic, we developed a simulator to apply the rules of the algorithms discussed in Section 3.2 chronologically on the data in a given test set TE. 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.

Table 3.2: 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

3.3.4 RQ1 Analysis Procedure

This question compares the existing prediction-based algorithm SmartBuildSkip with the commit grouping algorithms proposed by Beheshtian et al. (7). 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 (7). We measure the percentage of builds required for all the heuristics by calculating the total number of builds made and skipped for each project in our simulator, across the 10 test sets of Figure 3.2.

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. (7) 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 thesis 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.

3.3.5 RQ2 Analysis Procedure

As discussed in Section 3.2, ML-CI is a machine learning Random Forest Model trained on the commit metrics defined in Table 3.1, 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

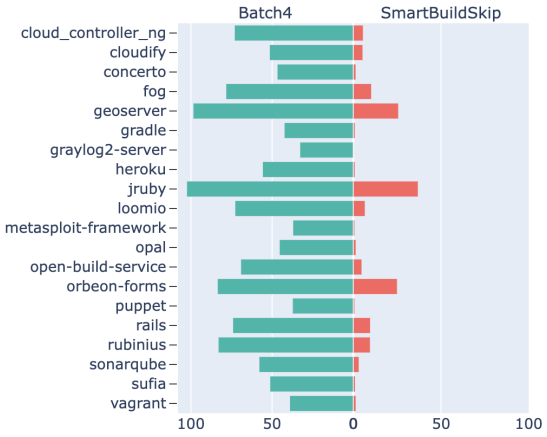
for percentage of builds required and for median delays. To do so, we use the Kruskal-Wallis test supported by the Pairwise Wilcoxon test as a post-hoc measure at a confidence interval of 95% (alpha=0.05). If two heuristics are found to be statistically different, we further evaluate the effect size to gauge the magnitude of their difference. We interpret effect size as negligible ($r < 0.147$), small ($0.147 \leq r < 0.33$), medium ($0.33 \leq r < 0.474$) or $large (r \geq 0.474)$ (24).

3.3.6 RQ3 Analysis Procedure

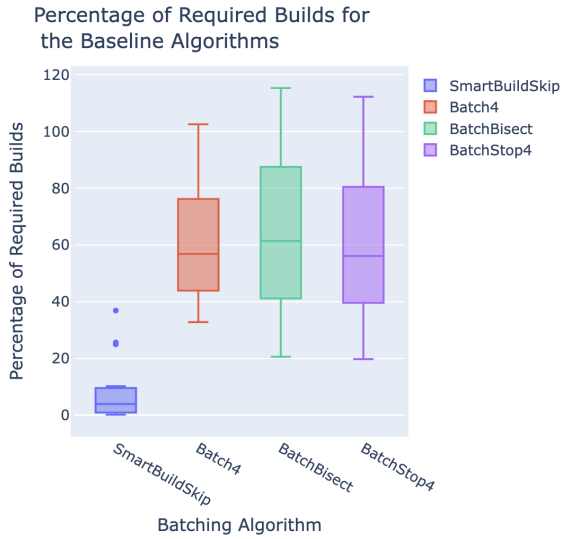
The Timeout rule discussed in Section 3.2 essentially uses the SmartBuildSkip model to predict build outcomes, yet (1) forces a build once a set number of builds has been skipped ("timeout") and (2) uses build grouping instead of direct execution of a build.

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.

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.



(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.

Figure 3.3: Builds Required by Baseline Algorithms.

3.4 Study Results

3.4.1 Performance Results of Baseline Heuristics (RQ1)

Table 3.3 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 from it.

Compared to the best batching approach (i.e., Batch4), SmartBuildSkip requires the lowest percentage of builds to be made with a median 3.91% compared to 56.85%, saving 52.94% builds. Figure 3.3a plots the percentage of builds required for the 20 projects by the Batch4 and SmartBuildSkip algorithms, while Figure 3.3b 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.

Table 3.3: Percentage of Builds Required by Individual Heuristic

Project	Algorithm					
	SmartBuildSkip	Batch4	BatchBisect		BatchStop4	
	% of Builds Required	% of Builds Required	% of Builds Required	Batch Size	% of Builds Required	Batch Size
cloud-controller-ng	5.507	73.091	79.099	4	77.597	8
cloudify	5.277	51.527	55.683	8	47.787	8
concerto	1.348	46.759	45.672	8	43.149	8
fog	10.205	78.333	89.184	2	83.004	8
geoserver	25.63	98.579	109.629	2	108.161	8
gradle	0.841	42.389	38.968	8	37.679	8
graylog2-server	0.142	32.78	20.588	16	19.734	16
heroku	0.817	55.79	59.539	8	54.349	8
jruby	36.879	102.563	115.351	2	112.281	8
loomio	6.605	72.793	83.676	4	76.181	8
metasploit-framework	0.658	37.105	28.671	16	27.803	16
opal	1.388	45.391	43.309	8	41.535	8
open-build-service	4.736	69.241	77.691	4	73.224	8
orbeon-forms	24.954	83.595	101.294	2	88.401	8
puppet	0.64	37.313	28.785	16	27.846	16
rails	9.603	74.142	85.861	4	77.921	8
rubinius	9.518	83.063	92.381	2	89.308	8
sonarqube	3.1	57.925	63.239	4	57.893	8
sufia	0.975	51.263	52.326	8	48.693	8
vagrant	1.384	39.041	32.592	16	30.319	16
Median	3.918	56.858	61.389	6.0	56.121	8.0

The low percentage for SmartBuildSkip in all 20 projects confirms that it can substantially reduce the resources consumed during CI. The BatchBisect algorithm still allows in at least 25% of the projects to reduce the number of builds to run by at least 59%, while the Batch(Stop)4 algorithms save at least 20% of builds for at least 75% of the projects. SmartBuildSkip requires less than 40% builds to be made for all projects.

The optimal batch size is 8 builds for 80% of the projects using BatchStop4 and for 30% 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, Puppet etc. For similar reasons, Nafaji et al. (44) and Beheshtian et al. (7) recommend using batching algorithms with projects having less than 25% failure rate.

The median delay of SmartBuildSkip varies from 9 to 797 commits. As shown in Figure 3.4, the minimum median delay is made by project JRuby (9 commits) while the maximum median delay (797 commits) is made by project Graylog2-server, for a median value of 53.5 builds across all projects. This means that if SmartBuildSkip misses to catch a build failure in JRuby, it can take a median of 9 commits before the build failure is caught. Similarly, it can take a median of 797 commits to catch the build failure

in Graylog2-server. This number varies for every project.

The drastic shift in performance from JRuby to Graylog2-server may be due to the sensitivity of the SmartBuildSkip model. Looking at Table 3.2, JRuby has a 55% failure rate, whereas Graylog2-server has a low failure rate of 3%. This means that the model trained on Graylog2-server 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.

A median of 40.25 commits are incorrectly skipped by the SmartBuildSkip algorithm. When a failing commit is not recognised immediately by SmartBuildSkip, each skipped commit will have a negative impact on CI until the failing commit is identified by a future build. This is referred to as “Bad Skips”. Figure 3.5 depicts the median number of bad skips made in SmartBuildSkip for the 20 projects. The median number of bad skips goes as high as 133.5 commits (project Sufia) and as low as 9 commits (project JRuby). 75% of the projects make median bad skips of more than 20.5 commits. This means that after skipping a failing build, at least 20 bad commits are skipped by

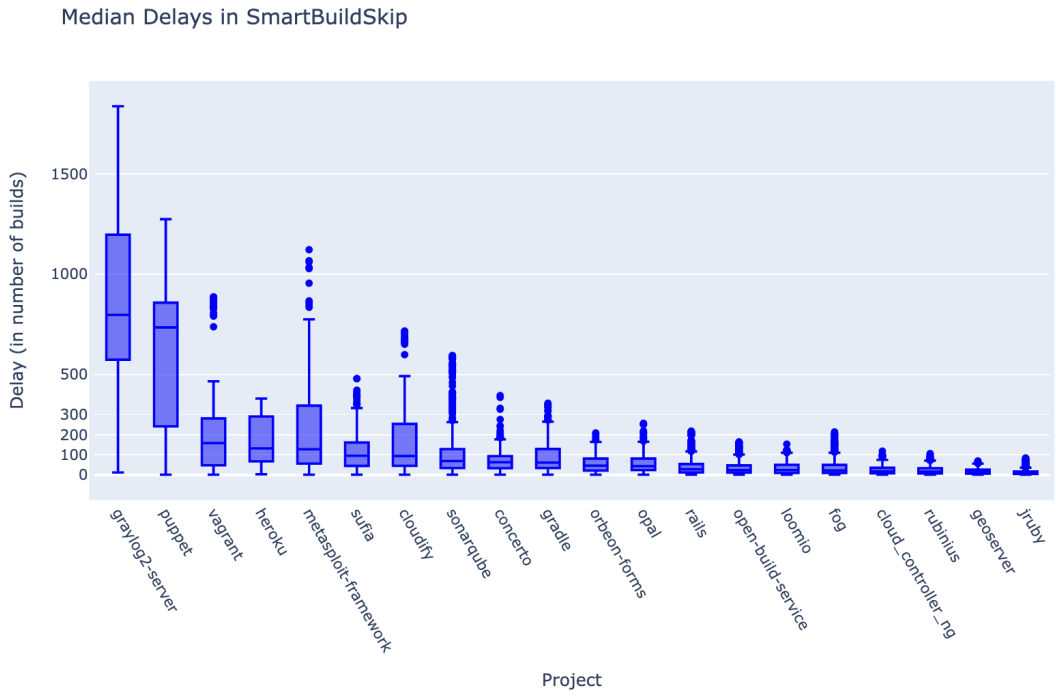


Figure 3.4: Boxplots of delays made per project by SmartBuildSkip algorithm.

SmartBuildSkip before the algorithm predicts a build failure. For such projects, since the probability of failing CI builds is low, it is difficult to predict whether a given commit in the CI line is buggy or non-buggy.

Summary of RQ1: *While SmartBuildSkip can save a median of 52.94% builds more than Commit Grouping, it introduces median delays in the identification of failed builds varying from 9 to 797 commits due to the low frequency of scheduled builds.*

3.4.2 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 3.4.2

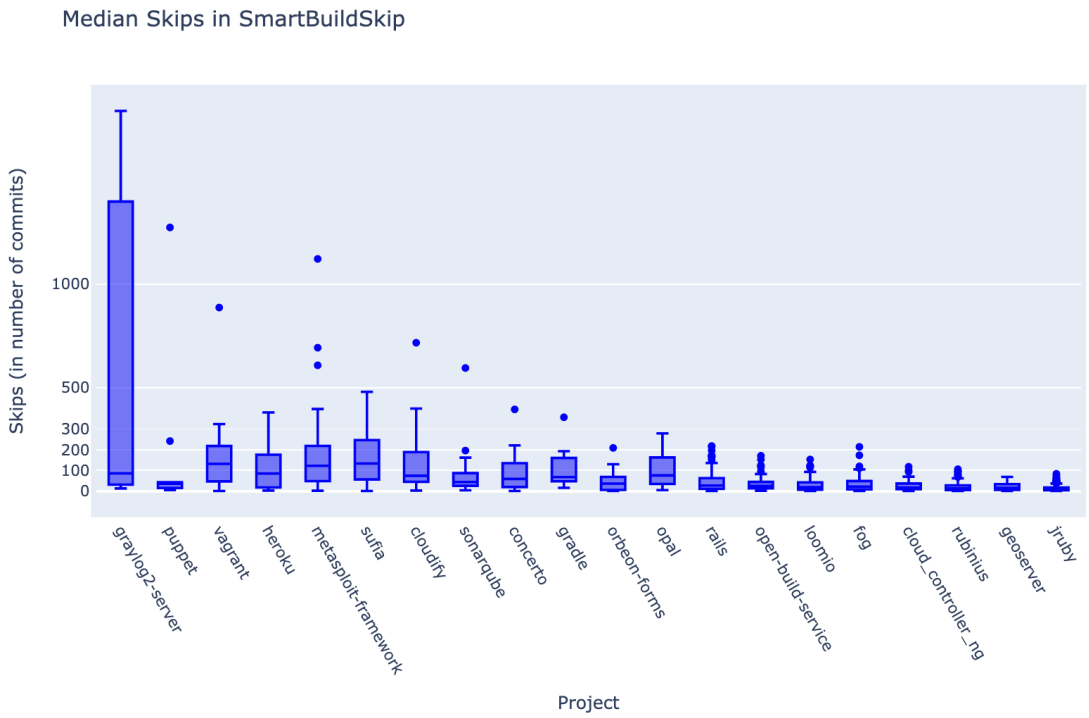
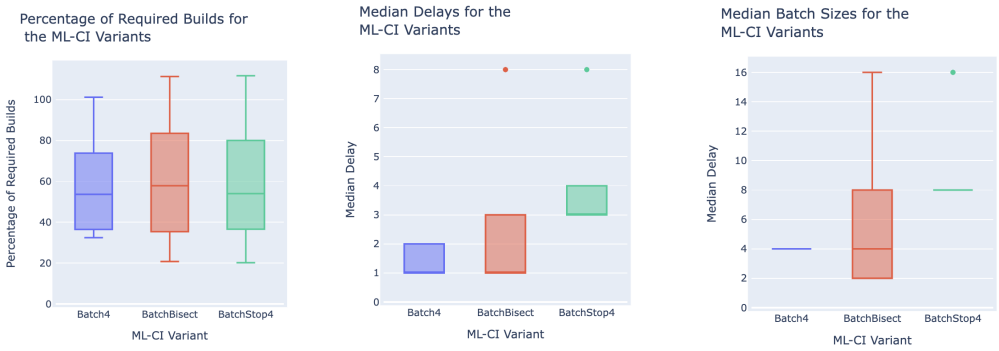


Figure 3.5: Boxplots of the number of bad skips made per project by SmartBuildSkip. Projects are shown in the same order as Figure 3.4.

evaluates the 3 variants of the ML-CI model. Sections 3.4.2 and 3.4.2 compare the results of ML-CI with SmartBuildSkip and Batch4, respectively.

Comparing the variants of ML-CI

Approach Table 3.4 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 3.6 depicts the box plots of Percentage of Builds required, median combined delays and best batch sizes



(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.

Figure 3.6: Boxplots demonstrating range of Percentage Build Required, Median Delays and the best performing batch sizes in ML-CI.

used by the ML-CI variants across 20 projects. While Figure 3.6c depicts the distribution of best performing batch sizes for each variant across 20 projects, each of the 20 data points in Figures 3.6a and 3.6b represent the percentage of builds required and median delay obtained with the best performing batch size for that ML-CI variant.

Results

For ML-CI, the median percentage of builds required is 53.66% for the Batch4 variant, 54.01% for the BatchStop4 variant, and 57.86% for the BatchBisect variant, although this performance is not consistent across projects. In order to determine statistical significance in the differences in terms of builds required by the 3 variants, we performed the Kruskal-Wallis test. At a confidence level $\alpha = 0.05$, the test revealed that there is no statistical difference ($H = 0.1242, p - value = 0.9398$) in the median 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, Metasploit-Framework, Puppet and Vagrant,

Table 3.4: Results of ML-CI

Project	ML-CI					
	Batch4 variant		BatchBisect variant		BatchStop4 variant	
	% Builds Required	Median Delay	% Builds Required	Median Delay	% Builds Required	Median Delay
cloud-controller-ng	70.129$^{\#}$	2.0$^{\#}$	71.381**	1.0**	74.969$^{\#\#}$	3.0$^{\#\#}$
cloudify	51.714$^{\#}$	2.0$^{\#}$	55.496$^{\wedge}$	8.0$^{\wedge}$	48.12$^{\#\#}$	4.0$^{\#\#}$
concerto	34.58$^{\#}$	2.0$^{\#}$	35.059**	1.0**	38.93$^{\#\#}$	3.0$^{\#\#}$
fog	73.554$^{\#}$	1.0$^{\#}$	79.59**	1.0**	82.824$^{\#\#}$	4.0$^{\#\#}$
geoserver	99.45$^{\#}$	1.0$^{\#}$	108.161**	1.0**	108.803$^{\#\#}$	3.5$^{\#\#}$
gradle	35.071$^{\#}$	2.0$^{\#}$	35.716$^{\#\#}$	3.0$^{\#\#}$	34.258$^{\#\#}$	3.0$^{\#\#}$
graylog2-server	32.448$^{\#}$	2.0$^{\#}$	20.778$^{\wedge}$	8.0$^{\wedge}$	20.209$^{\wedge}$	8.0$^{\wedge}$
heroku	48.39$^{\#}$	1.0$^{\#}$	53.484$^{\#}$	1.0$^{\#}$	51.177$^{\#\#}$	3.0$^{\#\#}$
jruby	101.221$^{\#}$	1.0$^{\#}$	111.338**	1.0**	111.725$^{\#\#}$	3.0$^{\#\#}$
loomio	71.971$^{\#}$	1.0$^{\#}$	82.512$^{\#}$	1.0$^{\#}$	75.77$^{\#\#}$	4.0$^{\#\#}$
metasploit-framework	35.961$^{\#}$	1.0$^{\#}$	28.829$^{\wedge}$	8.0$^{\wedge}$	28.066$^{\wedge}$	8.0$^{\wedge}$
opal	39.028$^{\#}$	2.0$^{\#}$	40.417$^{\#}$	2.0$^{\#}$	38.912$^{\#\#}$	3.0$^{\#\#}$
open-build-service	68.407$^{\#}$	1.0$^{\#}$	75.323**	1.0**	73.197$^{\#\#}$	3.0$^{\#\#}$
orbeon-forms	80.499$^{\#}$	2.0$^{\#}$	91.543**	1.0**	86.969$^{\#\#}$	3.0$^{\#\#}$
puppet	36.972$^{\#}$	1.5$^{\#}$	28.742$^{\wedge}$	8.0$^{\wedge}$	27.719$^{\wedge}$	8.0$^{\wedge}$
rails	74.121$^{\#}$	1.0$^{\#}$	85.068**	1.0**	77.263$^{\#\#}$	3.0$^{\#\#}$
rubinius	80.39$^{\#}$	1.0$^{\#}$	84.512**	1.0**	89.058$^{\#\#}$	3.0$^{\#\#}$
sonarqube	55.615$^{\#}$	2.0$^{\#}$	60.234$^{\#}$	2.0$^{\#}$	56.849$^{\#\#}$	3.0$^{\#\#}$
sufia	50.288$^{\#}$	1.0$^{\#}$	51.883$^{\#\#}$	3.0$^{\#\#}$	48.25$^{\#\#}$	3.0$^{\#\#}$
vagrant	35.582$^{\#}$	1.0$^{\#}$	32.468$^{\#\#}$	3.0$^{\#\#}$	30.096$^{\#\#}$	3.0$^{\#\#}$
Median	53.5	56.858	4.0	61.389	6.0	56.121

Batch Size legend: ** - 2, $^{\#}$ - 4, $^{\#\#}$ - 8, $^{\wedge}$ - 16

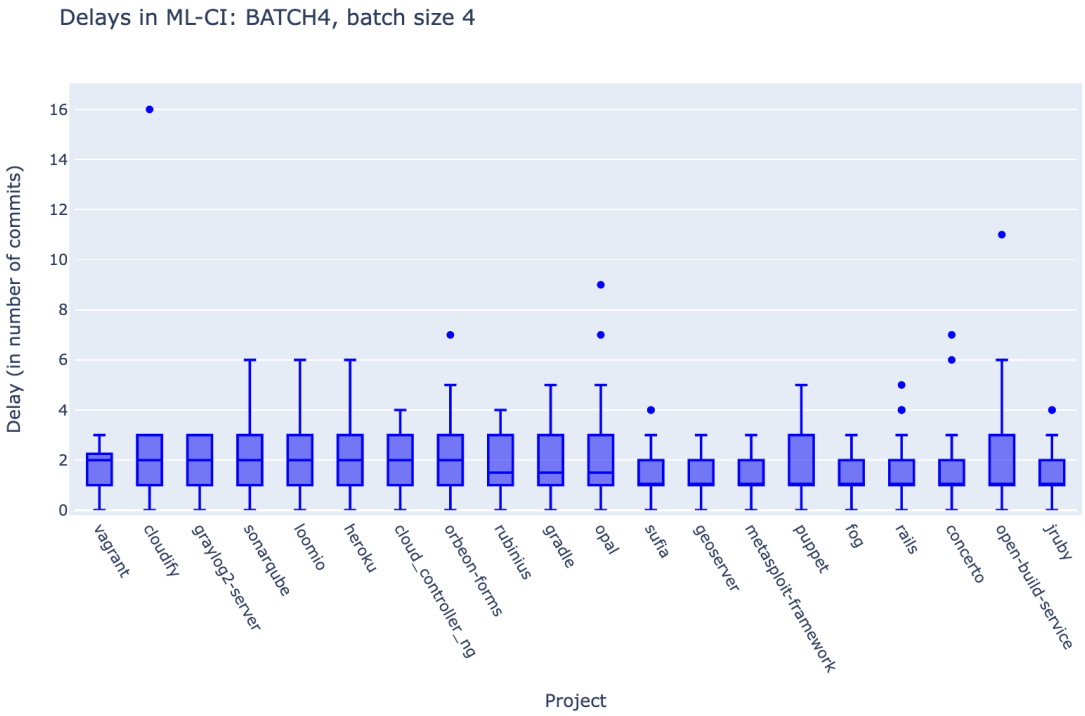


Figure 3.7: Boxplots of delays made per project by the ML-CI Batch4 variant.

while they demonstrate their least effective performances, in projects like Geoserver, JRuby, Rubinius and Orbeon-forms.

A myriad of factors can explain these findings of ML-CI. Based on the metadata of the dataset projects, we can hypothesize that dataset distribution 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 3.2, 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 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 BatchBisect variant, 2 builds for Batch4 variant and 4 builds for BatchStop4 variant. From Table 3.4, 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 3.7 that the median skipping based delays in ML-CI are less than 2 builds for most projects. This means that if a failing build is skipped by ML-CI, it is identified after a median of 2 builds. We performed the Kruskal-Wallis Test to determine any statistical differences between the performance of the 3 variants of ML-CI. We calculated ($H = 27.181, p-value = 1.252e-06$) in the median delay for the 3 variants, which indicated that there is a large statistical difference with an effect size of $r = 0.44$.

To further determine which variant differs from the rest, we used the Pairwise Wilcoxon Test at a confidence interval $\alpha = 0.05$ as a post-hoc testing measure. We measured $p-value = 7.4e-08$, and $p-value = 0.004$, between the median delays of BatchStop4 against Batch4 and BatchBisect respectively. With an effect size of $r = 0.85$ and $r = 0.45$ a strong statistical difference is noticed between the two variants. We can hypothesize that this difference arises due to the difference in BatchSize used by the three algorithms. The Batch4 uses the batch size of 4 and BatchBisect variant is also seen to perform best with a median batch size of 4 builds. However, the BatchStop4 variant perform best with the median batch size of 8 builds. Table 3.4 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.

Comparison to Baselines

Approach

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. For simplicity, we focus on the comparison of ML-CI to Batch4 because it operates only with the batch size of 4 and has a performance comparable or better than other batching heuristics.The scatterplots in Figure 3.9 and 3.11 plot the Percentage of Builds Required, whereas, Figure 3.8 and 3.10 plot the median delay, each of which compares ML-CI with the SmartBuildSkip and Batch4 baselines, respectively.

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.

Comparing to SmartBuildSkip

ML-CI incurs a median skipping delay of 1 to 3 builds across projects, significantly lower than that of SmartBuildSkip's 9 to 797 builds delay. From Figure 3.8, we see that all the data points lie above the $y = x$ reference line, 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 23 builds for 15 projects out of the studied 20 projects. On the

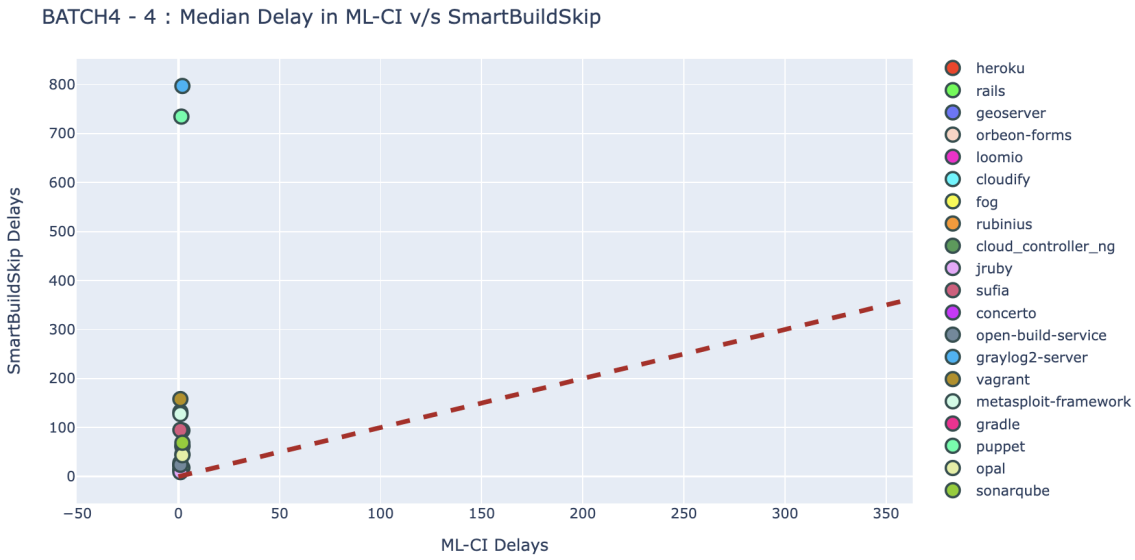


Figure 3.8: Median Combined Delay in ML-CI (using Batch4) vs SmartBuildSkip algorithm.

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 3 builds, with the maximum delay of 16 builds, while it ranges from a median 9 to 797 builds, with maximum delay of 1,837 builds, in SmartBuildSkip. This massive reduction in delay corresponds to a decrease percentage of 94.39% to 98.13% from SmartBuildSkip to ML-CI.

A Kruskal Wallis test confirmed the significant difference between the median delay induced by the different ML-CI variants and SmartBuildSkip with $p-value = 4.474e-13$ at $\alpha = 0.05$. The magnitude of this difference, given by effect size was calculated to be $r = 0.75$, indicating a large effect. The Wilcoxon rank sum post-hoc test further 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 $(7.3e-08, 7.3e-08, 7.3e-08)$ and a large effect size of $(0.60, 0.60, 0.60)$.

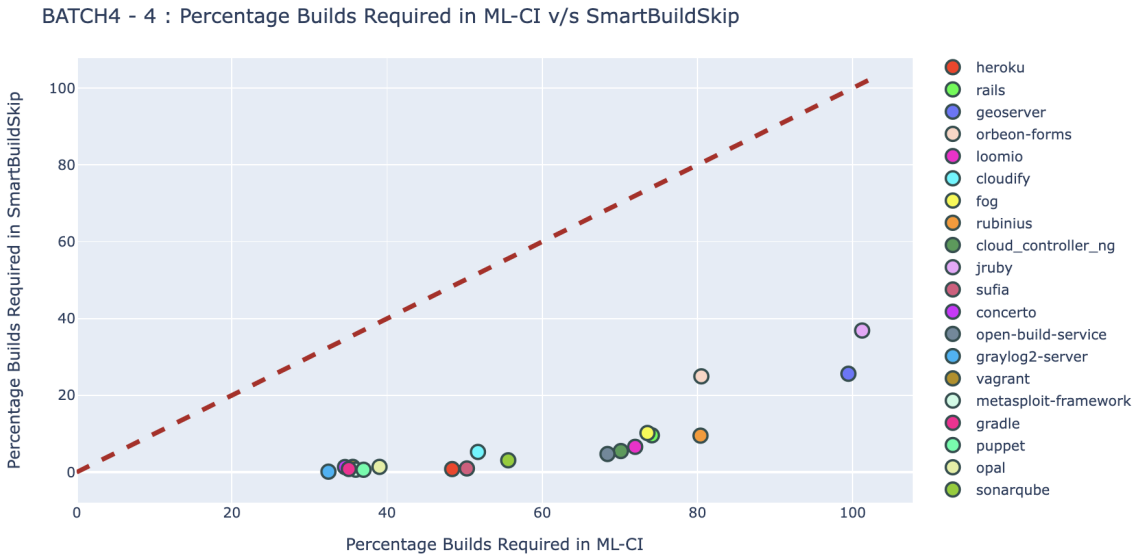


Figure 3.9: Percentage of Required builds in ML-CI (using Batch4) vs SmartBuildSkip algorithm.

ML-CI requires 49.74% more builds required compared to SmartBuildSkip. On the other hand, Figure 3.9 shows that a median percentage of 53.66% of builds across the 20 projects are run with ML-CI. All projects need to make more than 30% of builds (minimum of 32.44% in Graylog2-server). In contrast, with SmartBuildSkip, the projects require a median of 3.91% of builds to schedule, with all but 3 projects requiring less than 15% of builds. The maximum percentage of builds required occurs is about 36.89% for JRuby, while the same project requires 101.22% builds with ML-CI.

To analyze the statistical difference between SmartBuildSkip and the three variants of ML-CI, we performed a Kruskal Wallis test. On comparing the percentage of builds required, the test revealed a $p-value = 4.898e-09$ at $\alpha = 0.05$. The pairwise Wilcoxon rank sum post-hoc test further calculated a $p-value = 1.7e-09$ between SmartBuildSkip and each ML-CI variant, with a large effect size of 0.67.

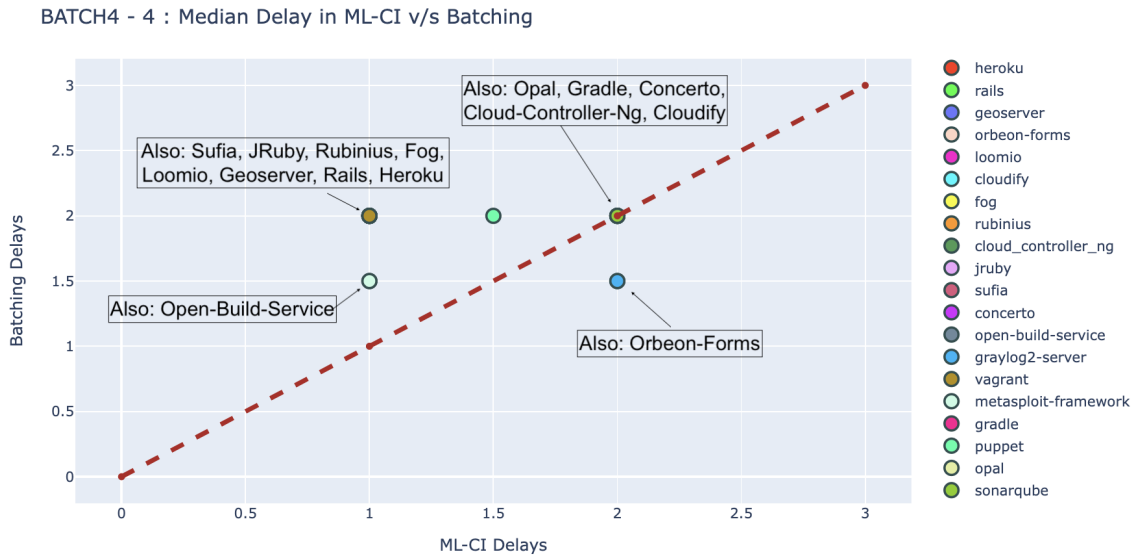


Figure 3.10: Median Combined Delay in ML-CI (using Batch4) vs. Baseline Batch4 Batching algorithm.

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 3.10 and 3.11, 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 1-2 builds for the Batch 4 ML-CI variant. Nine projects (e.g., Vagrant and Sufia) encounter slightly higher delays for Batching 4 compared to ML-CI.

In Figure 3.11, 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 in both heuristics is needed

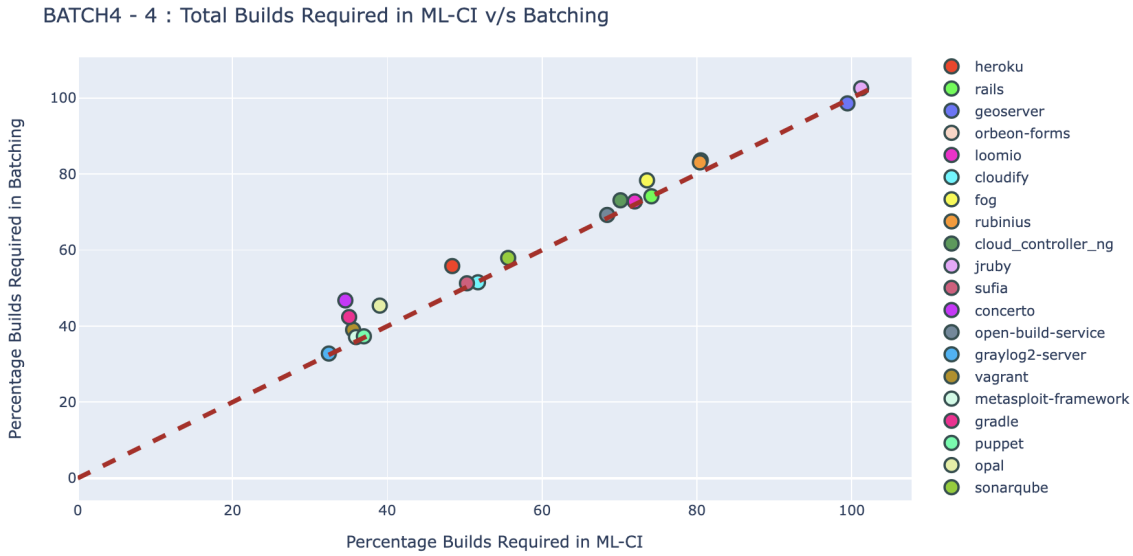


Figure 3.11: Percentage of Required builds in ML-CI (using Batch4) vs Baseline Batch4 Batching algorithm.

by JRuby, corresponding to 101.22% and 102.56% of builds for the ML-CI Batch4 variant and baseline Batch4, respectively. Both heuristics require the lowest percentage of builds for project Graylog2-server, which requires 32.44% of builds for ML-CI Batch4, and 32.77% builds for the baseline Batch4.

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., the Batch4 variant of ML-CI compared to the baseline Batch4 approach). The test calculated p-values= $(0.4777, 0.6783, 0.8201)$ for the Batch4, BatchBisect and BatchStop4 ML-CI variants, respectively, revealing no significant differences in percentage of builds required from their corresponding baseline algorithms.

Similarly, to measure the statistical difference of the median combined delays, we obtained a $p-value = 0.0016$ for the Batch4 variant, $p-value = 0.1934$ for the BatchBisect variant and $p-value = 0.0004$ for the BatchStop4 variant, indicating that

there are some differences between the median delays of the ML-CI Batch4 and Batch-Stop4 variants and their baseline algorithms at a confidence interval of $\alpha = 0.05$. That said, we computed effect sizes $r = 0.11$ and $r = 0.03$, which denote a small effect.

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 (Learn"). Table 3.5 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_touc* 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 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.

Table 3.5: 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

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 3.8. 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 3.2 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 49.74% more builds than SmartBuildSkip, albeit with a median skipping delay of 52.5 builds less.*

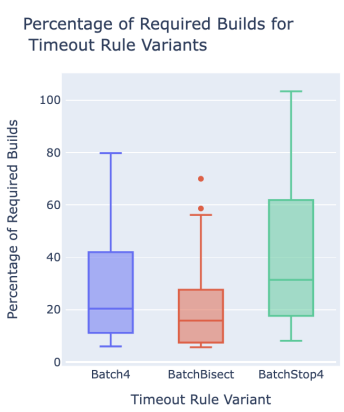
3.4.3 Performance Results for Timeout Rule (RQ3)

This section compares the performance of the Timeout Rule against the baseline SmartBuildSkip and Batching algorithms. Section 3.4.3 reports the results comparing the variants of the Timeout Rule. Sections 3.4.3 and 3.4.3 compare the results of the Timeout Rule with SmartBuildSkip and Batch4, respectively.

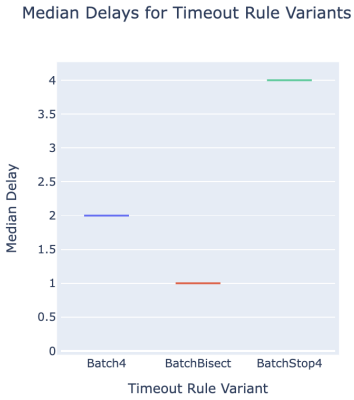
Comparing the variants of the Timeout Rule

Approach

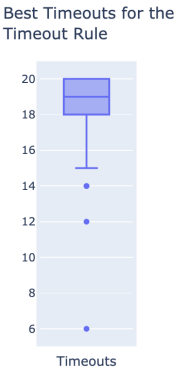
As mentioned in Section 3.3.6, here we evaluate the performance of the 3 variants of the Timeout Rule. The percentage of builds required and the median delay observed for the 20 studied projects, for each variant of the Timeout Rule is listed in Table 3.6.



(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.

Figure 3.12: Boxplots demonstrating range of Percentage Build Required, Median Delays, and best timeouts to use in Timeout Rule.

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 3.12 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 3.12c depicts the distribution of best performing timeout values across 20 projects, each of the 20 data points in Figures 3.12a and 3.12b represent the percentage of builds required and median delay obtained with the best performing batch size for that Timeout Rule variant.

Results

The Timeout Rule requires a median percentage of builds of 15.81% for the BatchBisect variant, 20.38% builds for the Batch4 variant, and 31.40% for the BatchStop4 variant, demonstrating a significant reduction in the percentage of builds required

Table 3.6: Results of Timeout Rule

Project	Timeout Rule					
	Batch4 variant		BatchBisect variant		BatchStop4 variant	
	Builds Required	Median Delay	Builds Required	Median Delay	Builds Required	Median Delay
cloud-controller-ng	38.912$^{\#}$	2.0$^{\#}$	23.032**	1.0**	57.084$^{\#\#}$	4.0$^{\#\#}$
cloudify	11.483$^{\#}$	2.0$^{\#}$	6.655**	1.0**	22.097$^{\#\#}$	4.0$^{\#\#}$
concerto	17.748$^{\#}$	2.0$^{\#}$	15.153**	1.0**	32.49$^{\#\#}$	4.0$^{\#\#}$
fog	10.773$^{\#}$	2.0$^{\#}$	8.451**	1.0**	15.295$^{\#\#}$	4.0$^{\#\#}$
geoserver	72.398$^{\#}$	2.0$^{\#}$	56.167**	1.0**	96.424$^{\#\#}$	4.0$^{\#\#}$
gradle	7.42$^{\#}$	2.0$^{\#}$	5.842**	1.0**	13.561$^{\#\#}$	4.0$^{\#\#}$
graylog2-server	13.735$^{\#}$	2.0$^{\#}$	9.171**	1.0**	22.818$^{\#\#}$	4.0$^{\#\#}$
heroku	25.673$^{\#}$	2.0$^{\#}$	16.604**	1.0**	49.569$^{\#\#}$	4.0$^{\#\#}$
jruby	17.636$^{\#}$	2.0$^{\#}$	15.089**	1.0**	24.796$^{\#\#}$	4.0$^{\#\#}$
loomio	79.79$^{\#}$	2.0$^{\#}$	69.999**	1.0**	103.397$^{\#\#}$	4.0$^{\#\#}$
metasploit-framework	56.793$^{\#}$	2.0$^{\#}$	58.641**	1.0**	70.887$^{\#\#}$	4.0$^{\#\#}$
opal	52.96$^{\#}$	2.0$^{\#}$	30.602**	1.0**	76.168$^{\#\#}$	4.0$^{\#\#}$
open-build-service	10.345$^{\#}$	2.0$^{\#}$	7.205**	1.0**	15.475$^{\#\#}$	4.0$^{\#\#}$
orbeon-forms	35.857$^{\#}$	2.0$^{\#}$	27.656**	1.0**	56.203$^{\#\#}$	4.0$^{\#\#}$
puppet	7.421$^{\#}$	2.0$^{\#}$	6.855**	1.0**	12.132$^{\#\#}$	4.0$^{\#\#}$
rails	45.059$^{\#}$	2.0$^{\#}$	27.56**	1.0**	66.691$^{\#\#}$	4.0$^{\#\#}$
rubinius	28.744$^{\#}$	2.0$^{\#}$	16.479**	1.0**	50.688$^{\#\#}$	4.0$^{\#\#}$
sonarqube	12.495$^{\#}$	2.0$^{\#}$	7.675**	1.0**	19.784$^{\#\#}$	4.0$^{\#\#}$
sufia	23.021$^{\#}$	2.0$^{\#}$	21.089**	1.0**	30.314$^{\#\#}$	4.0$^{\#\#}$
vagrant	5.977$^{\#}$	2.0$^{\#}$	5.645**	1.0**	8.112$^{\#\#}$	4.0$^{\#\#}$
Median	20.384	2.0	15.816	1.0	31.402	4.0

Batch Size legend: ** - 2, $^{\#}$ - 4, $^{\#\#}$ - 8, ^ - 16

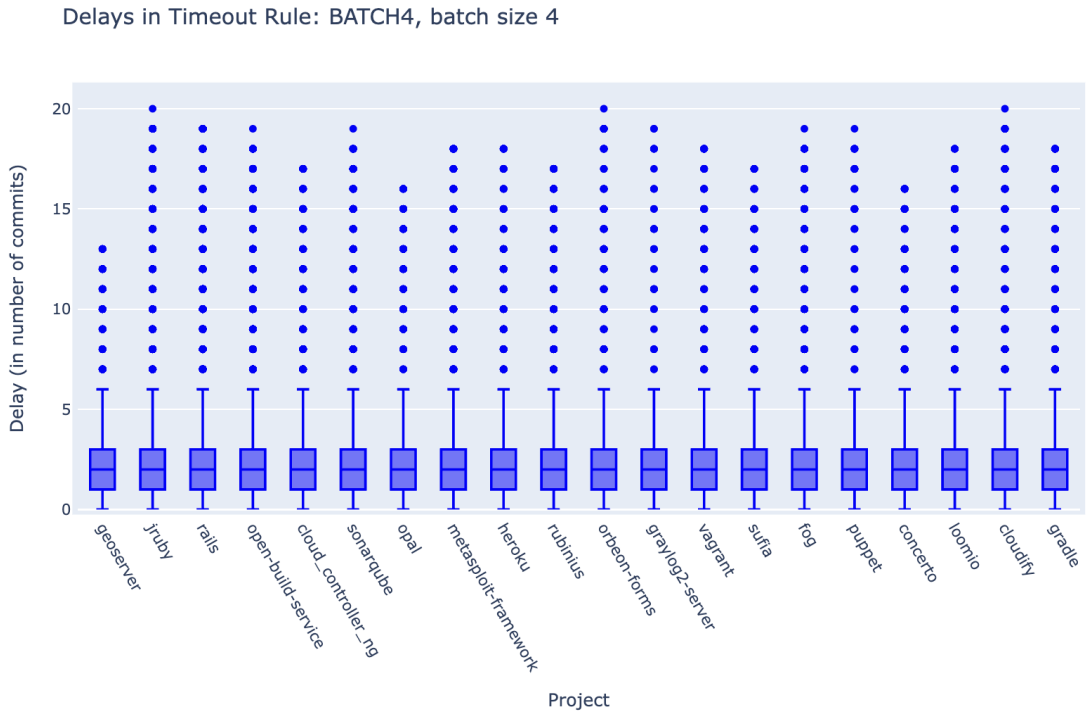


Figure 3.13: Median combined delays made per project by SSR-Batch4 variant.

during CI. Three quarters of projects require less than 61.88% builds for the Batch-Stop4 variant, less than 27.60% for the Batch4 variant, and less than 41.98% for the BatchBisect variant. Project Graylog2-server requires the least builds, i.e., 5.64% using BatchBisect with batch size of 2, while the maximum percentage of builds required occurs for JRuby, i.e., 141.84% using BatchBisect, with batch size 16.

In order to determine any statistical differences between the three variants of the Timeout Rule, we used the Kruskal Wallis test. By performing the test at a confidence level of $\alpha = 0.05$, we computed $p - value = 0.0254$, which signifies a statistical difference between the builds required for the 3 variants of this rule with an effect size of $r = 0.09$. The Pairwise Wilcoxon rank sum post-hoc test confirmed that while Batch4 has similar performance to others, the BatchBisect and BatchStop4 variants differ from

one another with a $p-value = 0.026$ and effect size of $r = 0.24$, connoting a large effect.

The median combined delays in a project for the Timeout Rule range from 1 to 4 builds, with 1 build for BatchBisect variant, 2 builds in Batch4 variant and 4 builds in BatchStop4 variant. Figure 3.13 shows the distribution of delays in the Timeout Rule (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.

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 ($r = 1$) between the median delays of the 3 variants. To identify which variants stand out, we performed a Pairwise Wilcoxon post-hoc test between each pair of the 3 Timeout Rule variants at a confidence interval of $\alpha = 0.05$. We noticed that statistically significant differences arise between the BatchBisect and BatchStop4 variant.

We find that smaller batch sizes are more effective with the Timeout Rule. Table 3.6 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 and increases the turn around time for a build.

Comparison to Baselines

Approach

In the following sections of this chapter, 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 because it operates only with the batch size of 4 and has a performance comparable or better than other batching heuristics. Figure 3.15 and 3.17 represent the scatter plots that plot the Percentage of Builds Required, whereas the scatter plots in Figures 3.14 and 3.16 plot the median delay, each of which compares the Timeout Rule with the SmartBuildSkip and Batch4 baselines, respectively.

Comparing to SmartBuildSkip

The Timeout Rule induces a median skipping delay of 51.5 commits less than SmartBuildSkip, but requires a median of 16.46% more builds. In Figure 3.14, we see the median delay incurred for SmartBuildSkip and the Timeout Rule. Each data point (X, Y) on the plot represents one project that incurs X commits of skipping delay for the Timeout Rule and Y commits of skipping delay for SmartBuildSkip. We see that all the data points are above the $y = x$ reference line, meaning that the delays are lower for the Timeout Rule than for SmartBuildSkip. All the data points have a median skipping delay of 2 builds for the Timeout Rule, whereas more than half of the projects have a median delay of more than 53.5 commits for SmartBuildSkip. This indicates that the Timeout Rule makes a 96.26% decrease in the median delays in comparison to SmartBuildSkip. The maximum median delay made by SmartBuildSkip corresponds

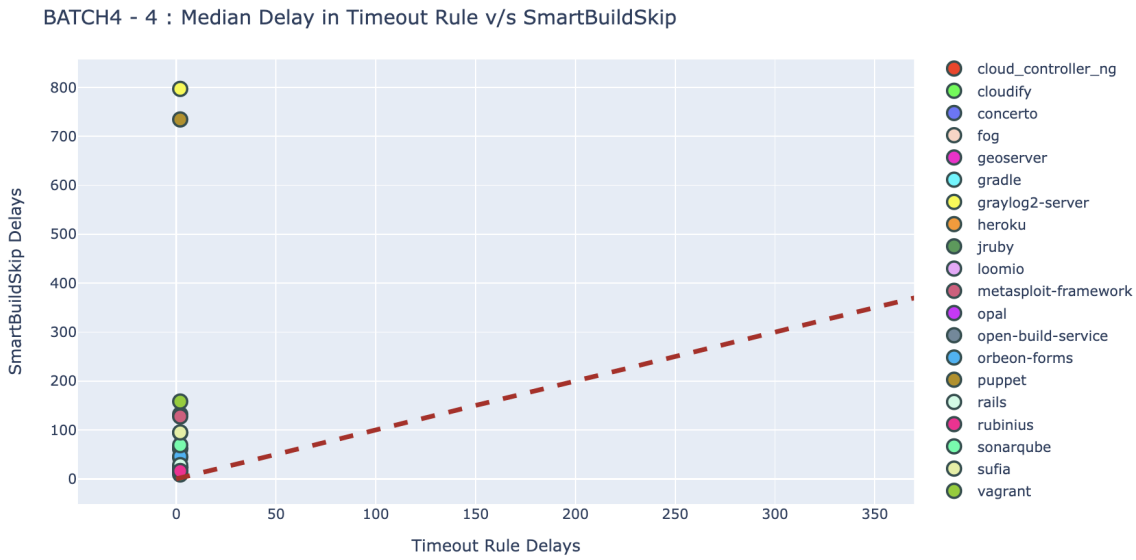


Figure 3.14: Median Combined Delay for the Timeout Rule (using Batch4 and best performing timeout threshold for that project) vs SmartBuildSkip algorithm.

to 797 builds with Graylog2-server, compared to only 20 builds of median delay for the Timeout Rule.

On the other hand, SmartBuildSkip requires 16.46% less builds compared to the Timeout Rule in most projects. In fact, the highest percentage of builds required by SmartBuildSkip is 36.87% in project JRuby, compared to 79.78% of builds for the Batch4 Timeout Rule variant. However, with a few exceptions like Orbeon-forms, Rubinius, Fog, JRuby and Geoserver, all other projects require less than 40% of total builds using the Timeout Rule, with 9 of them requiring less than 20% builds. In other words, Figures 3.14 and 3.15 clearly show a trade-off between percentage of builds required and median skipping delay for the Timeout 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.

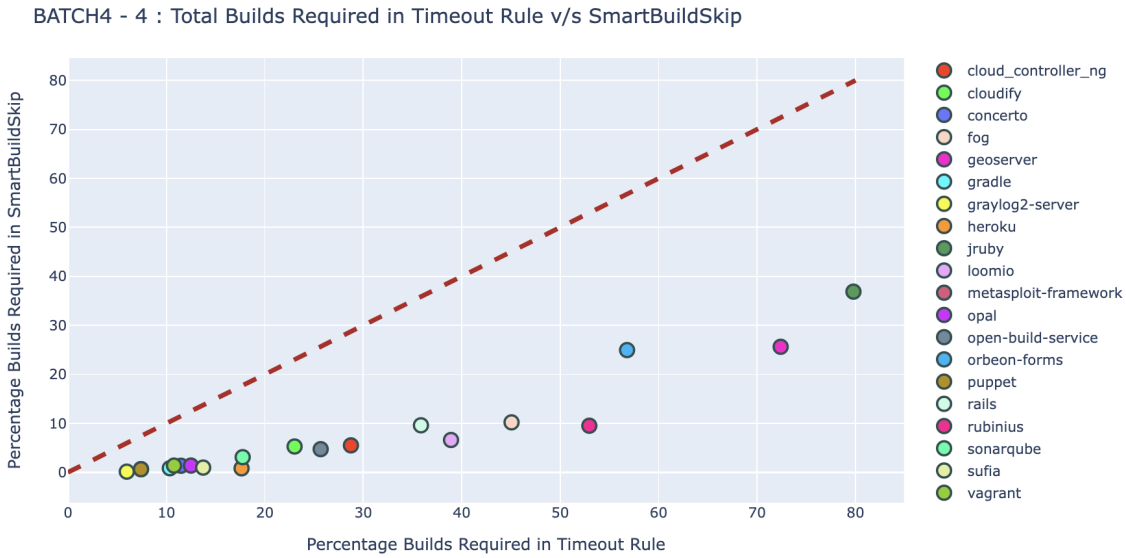


Figure 3.15: Percentage of Required builds for the Timeout Rule (using Batch4 and best performing timeout threshold for that project) vs SmartBuildSkip algorithm.

Compared to the median delay of 53.5 builds required by SmartBuildSkip, the delays are substantially lower in the Timeout Rule. The minimum median delay of 9 commits is made by JRuby in SmartBuildSkip, which in contrast makes 2 commits of delay with the Timeout Rule. One the other hand, the maximum median delay made by SmartBuildSkip is 797 commits with Graylog2-server, which also requires only 2 commits of delay with Timeout Rule.

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 revealed $p-value = 1.142e-06$ for percentage of builds required and $p-value < 2.2e-16$ 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 requires significantly higher builds compared to SmartBuildSkip with $p-value = (2.5e-05, 0.0005, 3.7e-06)$ and effect sizes $r = (0.47, 0.38, 0.51)$ connoting a significant effect.

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 3.16 and 3.17, we 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 turn around time 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 73.09% to 28.47% in the case of Cloud Controller, 57.92% to 17.74% in Sonarqube and so on.

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,

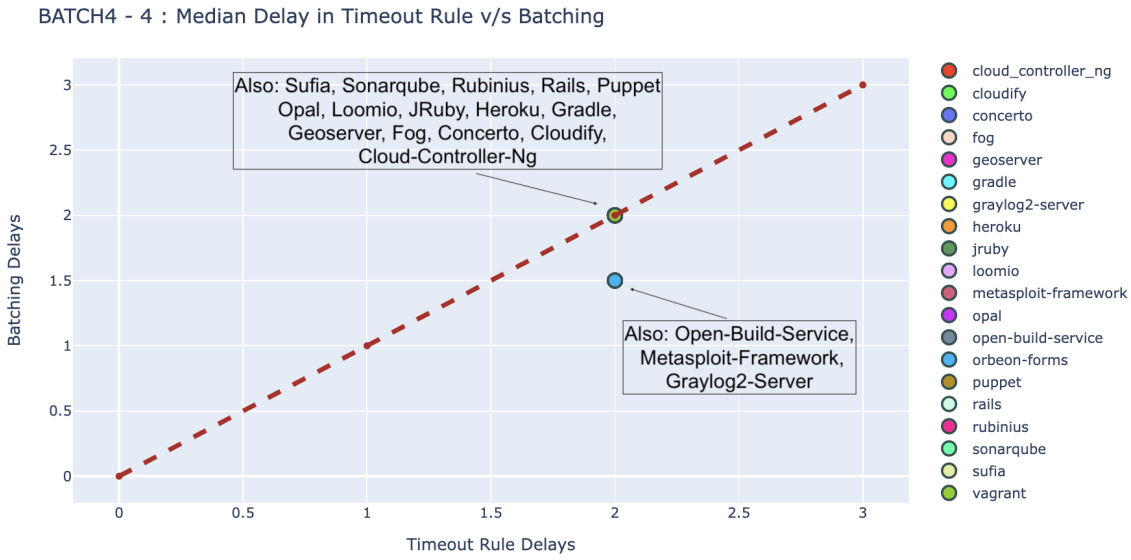


Figure 3.16: Median Combined Delay in the Timeout Rule (using Batch4 and best performing timeout threshold for that project) vs Baseline Batch4 Batching algorithm.

we computed $p-values = (3.899e-05, 1.367e-06, 0.02831)$ for percentage of builds required, and $p-values = (0.0398, 1.923e-05, 0.0400)$ for median delays. This indicates that each variant of the Timeout Rule has significant performance differences from its baseline counterpart. The computed effect sizes signified large statistical difference between each pair of heuristics compared.

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 3.18, we analyse the JRuby project to inspect the effects of parameters of the Timeout Rule on its performance. 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

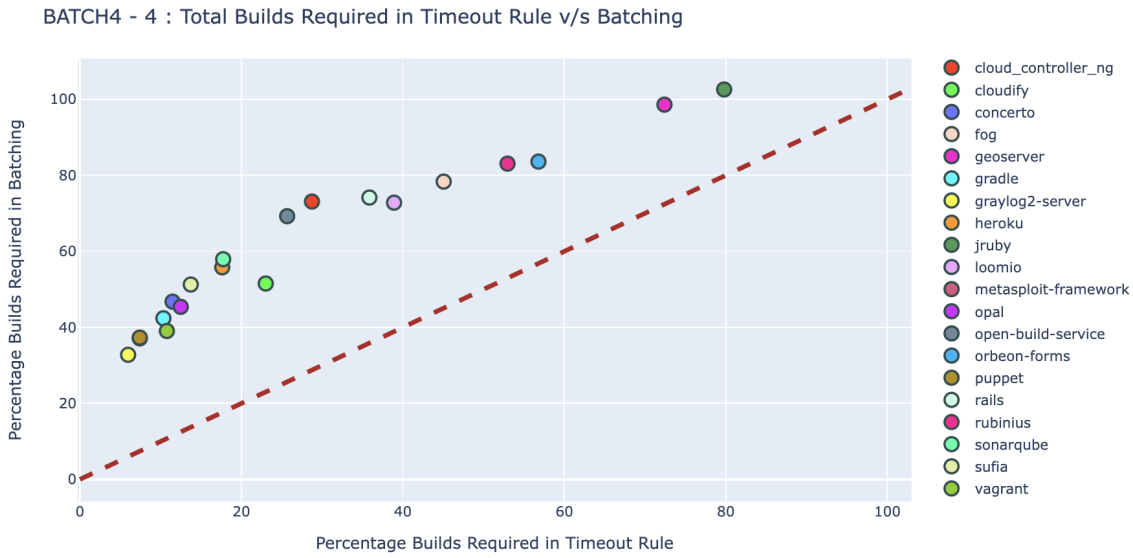


Figure 3.17: Percentage of Required builds in he Timeout Rule (using Batch4 and best performing timeout threshold for that project) vs Baseline Batch4 Batching algorithm.

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 40% builds, it induces a high median skipping delay close to 9 builds. From Figures 3.4 and 3.13, we also learnt that the median percentage of builds required for the Batch4-Timeout variant on JRuby Geoserver 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

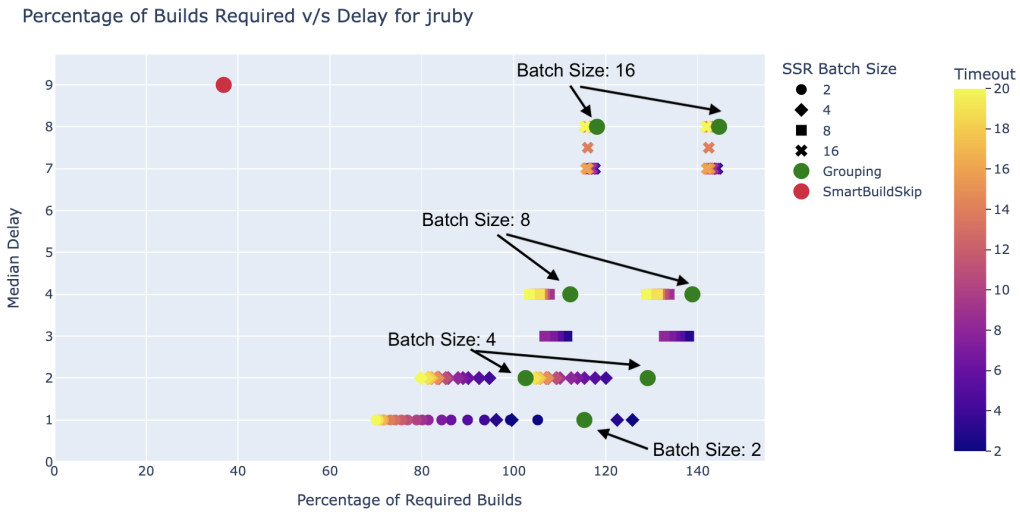


Figure 3.18: Percentage of Builds Required v/s Median Delay for each timeout threshold of the Timeout Rule for JRuby

configurations in comparison to other heuristics for JRuby. 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.

Summary of RQ3: *The percentage of builds required by the Timeout Rule is 36.47% lower than for Batch4, but 16.46% greater than SmartBuildSkip. The median delay induced by the Timeout Rule is, however, 51.5 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.*

3.5 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 simulate real-world CI scenarios. To mitigate this threat, we perform bootstrapping (51) of 200 iterations during each model training phase. We use the bootstrap model that provides the best F1 score in the training set of a given iteration of our 10-fold cross-validation for the testing phase and build outcome prediction of that iteration.

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 (6; 7; 29; 44).

Internal Validity: The features for predicting the build outcome have been chosen based on prior literature (21; 29). 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 3.4.3, 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.

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.

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.

3.6 Chapter Summary

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 36.47% 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 build turnaround time of CI systems.

CHAPTER 4

Lightweight Dynamic Batching Algorithms for Continuous Integration

CONTINUOUS Integration (CI) is a widely adopted process in software engineering that virtually merges developers' pull requests with the code base to perform builds and tests essential for quality assurance. CI, however, is also an expensive process, due to the large number of pull requests that are pushed by developers on a daily basis. To reduce the cost of CI, many companies adopt batching algorithms that combine several commits into a single build, either with a fixed or a dynamically evolving batch size. While a successful, dynamic batching technique has been proposed by earlier work, we propose a simpler, more flexible dynamic batching technique that can update batch sizes based on the outcome of the previous batch build. Empirical evaluation on 286,848 commits from 50 open-source projects using TravisCI shows that our

lightweight batching technique can perform equally well to the more complex state-of-the-art batching techniques, outperforming static batching techniques by a median of 7.73%.

4.1 Introduction

Continuous Integration (CI) (12) is a popular quality assurance process used by countless software organizations according to which a pull request made by a developer is integrated virtually with the current snapshot of a code base in order to build (i.e., compile and test) the changed code. This is typically done before or during the code review process, when any issues identified during CI can still easily be fixed. To support this process, countless CI automation servers (e.g., GitHub Actions, Azure DevOps Server) have been built and are widely available to software organizations.

However, CI is a costly process in terms of time, energy and computing resources. An individual build can take hours to complete (59), thereby affecting developer productivity. Furthermore, for a given code change one often needs to run multiple builds in parallel, for instance to support different platforms or to evaluate the quality of different configurations of a given product. Even large companies like Google and Mozilla report their CI costs in millions of dollars annually (29).

Hence, attempts have been made to reduce the cost of CI builds, both in terms of scheduling less builds (3; 7; 29) and in terms of making scheduled builds faster (13). We focus on the former line of research, where researchers have worked on (1) AI-based approaches that decide whether to schedule a build by predicting whether it will fail, and on (2) less complex rule-based approaches. While the former AI-based techniques have shown to be effective, they also bring additional costs and disadvantages to the

table, such as the cost of gathering data and training a model, coupled with the cost of retraining the model due to data drift and the potential cost of prediction errors.

Hence, in this study, we focus particularly on rule-based heuristics, i.e., Batching and CI-Skip. Batching is the processing of building multiple incoming commits together in a single build. Although it was first studied empirically by Najafi et al. (44), they indicate that the technique was in use long before by a variety of companies and open-source organizations. Initial batching techniques used a fixed number of commits as batch size. If the batch build was unsuccessful, diverse fallback algorithms like BatchBisect, Batch4, BatchDivide4 and BatchStop4 were used to identify the individual commits responsible for the batch failure.

While effective and lightweight, the inability of adapting the batch size to a given project's build context meant that static batching is not recommended for projects with historical failure rate greater than 25% (44). Bavand et al. presented dynamic batching techniques able to adapt the batch size using a mathematical model based on weighted smoothing and failure rate. The same fallback algorithms as before are then used to identify failing commits.

CI-Skip is a different kind of rule-based technique that defines the characteristics of commits that can be CI-Skipped, i.e., can be integrated without scheduling any build. This technique focuses on eliminating builds for simple commits that would trigger the CI process unnecessarily. Those include commits that touch documentation, source code and meta files, commits that only modify source code comments or format the source code by adding newlines and/or spaces, and commits that are related to version preparation. CI-Skip has never been evaluated empirically in the context of batching.

This thesis presents a Lightweight dynamic batching (LWD) algorithm based on

simple, light (including random) techniques run after every build to decide the batch size of the next build, aiming to simplify the more complex mathematical models of the state-of-the-art dynamic batching techniques. We also perform the first empirical study that combines batching with CI-Skip rules, comparing the performance with the individual batching techniques. Our large-scale empirical study on 286,848 commits from 50 open-source projects collected from the TravisTorrent data repository measures the percentage of builds saved by applying each of the existing and new heuristics (static batching, state-of-the-art dynamic batching, LWD and CI-Skip) to address the following three research questions:

- *RQ1: What is the best LWD approach to dynamically update the batch size?* – We evaluate all our lightweight batch updating LWD techniques against each other to identify the most effective techniques amongst them.
- *RQ2: How does LWD perform in comparison to state-of-the-art static and dynamic batching techniques?* – We compare the performance of our new heuristic against existing batching techniques to identify the most effective heuristic for build scheduling reduction.
- *RQ3: Does the addition of CI-Skip rules make batching more effective?* – We combine the studied batching techniques with CI-Skip to evaluate the advantages of combining the two rule-based heuristics.

We summarize below our major study contributions:

- Our LWD algorithms can save a median of 7.75% more builds than state-of-the-art static batching algorithms.

- The performance of the LWD algorithms is equally high as that of state-of-the-art dynamic batching.
- Although CI-Skip is effective in pruning out harmless commits from CI builds, it does not significantly improve on the performance of dynamic batching algorithms.
- Our study's replication package (34) is available online.

We organized the remainder of this chapter as follows. Section 4.2 introduces our Lightweight Dynamic Batching Algorithm. Section 4.3 introduces our study methodology, including the study goal, research questions, and study steps. Sections 4.4.1, 4.4.2, and 4.4.3 present our study results per research question. Section 4.5 discusses threats to the validity of our empirical study. Finally, Section 4.6 concludes this chapter and suggests future work.

4.2 Lightweight Dynamic Batching Algorithm and Variants (LWD)

Lightweight Dynamic Batching (LWD) is a simple rule-based batching heuristic that can dynamically update batch sizes of traditional batching techniques during CI operations. It only uses information about the failure rate, i.e., number of failing builds in the most recent batch that required bisection activity, to decide the batch size for building upcoming commits. If the majority of commits in the previous batch are failing, then a smaller batch size is used for the current batch, given the high risk of future build failures (47). Conversely, if the majority of commits in the previous batch are

successful, then a larger batch size is used, as the risk of future build failures has been reduced.

The main challenge of an LWD technique is to try to increase or decrease batch sizes at the right moment, such that the batch size can be as large as possible (i.e., minimal number of builds to be run) in between build failures, while as small as possible during a build failure streak (i.e., minimizing the bisection overhead of traditional batching). As such, a variety of LWD variants can be designed differing mainly in the way in which the new batch size is estimated.

This paper presents 5 main LWD variants covering a wide range of techniques for incrementing/decrementing batch sizes. All of these share a few basic principles:

- *Fallback Rule: If the failure rate of the most recent batch is greater than a given X%, the next batch size defaults to 1.* – This rule helps to account for a continuous stream of failing incoming commits. In order to avoid unnecessary bisection builds to identify the failing commits in a batch, we build one commit at a time, influenced by SmartBuildSkip (29), until a successful build is encountered. Once a successful build is encountered, the batch size is increased again based on the LWD variant's batch size updating rules.

- *Retention Rule: Retain batch size if the most recent batch has a successful final commit and failure rate less than Y%.* – This rule helps to account for scattered failing incoming commits. If there were very few failing commits in a batch, eventually succeeded by successful commits, we do not anticipate more failing commits in the next batch.

- *Customizability Rule: Every batch size updating technique can be customized according to the needs of a project.* – There exist customizable elements in every

technique to allow flexibility to the build engineers to control batching in CI according to their current requirements.

- *Factor Rule: Every batch size update occurs with respect to a factor.* – Fundamentally, a new batch size is obtained by adding, subtracting, multiplying, or dividing the old batch size with a factor to obtain a next batch size. With the exception of Random batch updating, where the next batch size is chosen randomly, all our batch updating techniques have non-zero factors.

4.2.1 Lightweight Batch Size Updating Variants

This subsection details our five LWD techniques incorporating the 4 principles presented previously, yielding Linear, Exponential, Random, Mixed and Most Frequently Used (MFU) variants. Of these five, the Linear, Exponential and Random techniques are atomic, i.e., they comprise only one condition to decide the next batch size. The Mixed and MFU variants are composite techniques that go through a series of if-else conditions to decide the next batch size.

Similar to the state-of-the-art dynamic batching technique, the LWD technique also relies on the three fallback algorithms BatchBisect, BatchStop4 and BatchDivide4 to identify culprit commits in a failing batch. Once a batch is formed and build is executed, any culprits of the batch are identified using one of the 3 fallback algorithms. Once culprits are identified (if any), we update the batch size using our five LWD batch updating variants.

Algorithm 9 calls the Failure Case and Success Case procedures depending on the outcome of the previous batch build. If the previous batch consists of all successful commits and the batch were compiled with a single build, the Success Case procedure

is called. Otherwise, the algorithm calls the Failure Case procedure.

Algorithm 9 Batching Function

```
1:  procedure FILLBATCH: (incomingCommit)
2:      i ← 0
3:      batch ← []
4:      fails ← []
5:      batchSize ← 16                          ▷ Default starting batch size
6:      while i < batchSize do
7:          batch.append(incomingCommit)
8:          i ← i + 1
9:      end while
10:     BatchBisect(batch, fails)               ▷ or BatchStop4 or BatchDivide4
11:     FR ← Count(0 in fails) ÷ batchSize      ▷ Measuring Failure Rate
12:     if FR > 0 then
13:         batchSize ← Failure_Case(FR)
14:     else
15:         batchSize ← Success_Case()
16:     end if
17: end procedure
```

In this study, we identify 39 sub-variants of the LWD batching technique. I.e., the five batch updating techniques form 13 sub-variants, each of which can be used in combination with one of three fallback algorithms. These sub-variants are listed in Table 4.1.

Atomic Batch Size Updating: Linear, Exponential, Random

Algorithm 10 details the steps of the atomic batch size updating techniques. In these algorithms, the elements fallback_limit (implements the Fallback Rule), retention_limit (implements the Retention Rule), factor (implements the Factor Rule), minimum and maximum can be customized according to the needs of the project. As depicted in lines 2, 5, 8 and 11, these techniques use the failure rate of the previous batch to decide

whether to retain the previous batch size, fall back to size 1 or update the batch size. To update the batch size in lines 12 and 20, they add, subtract, multiply or divide the current batch size by 'factor' (which is either constant or randomly chosen) according to the update rules in Table 4.1. We choose reasonable values for 'factor' to avoid making too large batch sizes (e.g., batch sizes exceeding 50 builds). To do so, we limit the factors of Exponential variant to (2,3). Similarly, factors of the Random-Exponential sub-variant is limited to [2...4].

Algorithm 10 Linear, Exponential, Random Batch size updating techniques

```
1:  procedure FAILURE CASE : (FR = failure_rate)
2:      if (FR > fallback_limit) then                          ▷ Fallback Rule
3:          return minimum
4:      end if
5:      if (FR < retention_limit) && (last_commit = Success) then   ▷ Retention Rule
6:          return (current_size)
7:      end if
8:      if current_size < factor then
9:          current_size = minimum
10:         return (current_size)
11:     else
12:         update current_size according to Table 4.1          ▷ Factor Rule
13:     end if
14: end procedure

15: procedure SUCCESS CASE
16:     if current_size > maximum then
17:         return (current_size)
18:     else
19:         update current_size according to Table 4.1
20:     end if
21: end procedure
```

Table 4.1: Batch size update rules for 3 atomic and 2 composite LWD variants

Variant		Factors	New Batch Size	
Base	Sub-Variant		Success Case	Failure Case
Linear		1,2,3,4	current+factor	current-factor
Exponential		2,3	current*factor	current ÷ factor
Random	Linear		current + F F = random(current, maximum)	current - F F = random(minimum, current)
	Exponential		current * F F = random(2, 4)	current ÷ F F = random(minimum, current)
	Random		random(current, maximum)	random(minimum, current)
Mixed		2,3	current*factor	current-factor or current÷factor
MFU		2,3	current*factor	current-factor or current÷factor

Composite Batch Size Updating: Mixed, MFU

The composite batch size updating techniques use an if-else-if branch to decide the next batch size to use in CI. These approaches help to form more conservative batch updating techniques that can decide how much a batch size needs to be reduced. The difference between our two composite LWD variants boils down to their ability to handle large failure rates. While the Mixed LWD variant immediately falls back to the minimum batch size, the MFU algorithm will default to the batch size that was most frequently used thus far for this project.

Algorithm 11 presents the Mixed and MFU variants. In this algorithm, elements *fallback_limit, min_limit, med_limit, max_limit, factor, minimum* and *maximum* are customizable according to the needs of the project. Each of these limits are tested individually as well as along with the criterion that the previous batch ended with a successful commit (to ensure that we are not in the middle of a series of failing commits). If the *min_limit* is not yet reached, we retain the current batch size in the case of the Mixed batch updating technique. Consequently, for Mixed and MFU techniques we update the batch size cautiously by subtracting $factor$ if the *med_limit* is not reached, but more aggressively divide by $factor$ if the *med_limit* is exceeded. To avoid making large batch sizes, we limit the $factor$ used in the Composite batch updating variants to (2, 3) only.

Algorithm 11 Mixed and MFU Batch Size Updating Techniques

```
1:  procedure FAILURE CASE : (FR = failure_rate)
2:      if (FR > fallback_limit) then                          ▷ Fallback Rule
3:          return minimum
4:      end if
5:      if current_size < factor  then
6:          return (minimum)
7:      else if (FR < min_limit) && (last commit = Success) then   ▷ For Mixed Only
8:          return (current_size)                              ▷ For Mixed Only
9:      else if (FR < med_limit) && (last commit = Success) then
10:         return (current_size - factor)
11:     else if (FR < max_limit) && (last commit = Success) then
12:         return (current_size ÷ factor)
13:     else
14:         return minimum                                     ▷ For Mixed Only
15:         return MFU(batch_size)                             ▷ For MFU Only
16:     end if
17: end procedure

18: procedure SUCCESS CASE
19:     if current_size > maximum  then
20:         return (maximum)
21:     else
22:         return min(current_size*factor, 16)
23:     end if
24: end procedure
```

4.3 Data Collection

4.3.1 Study Goal and Research Questions

In order to validate the performance of our LWD variants compared to the baseline batching techniques of Chapter 2, we perform an empirical study. Using the Goal/Question/Metric template (5), our study goal is as follows: **analyze** the performance of the lightweight dynamic batching variants during Continuous Integration (CI) testing; **for the purpose of** (i) identifying simpler heuristics for build savings and (ii) identifying ideal batch sizes for batching techniques; **with respect to** the percentage of builds saved in comparison to building all commits; **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.

In particular, we address the following research questions:

RQ1: *What is the best LWD approach to dynamically update the batch size?* – We use this question to compare the proposed linear, exponential, random, mixed and most frequently used (MFU) batch updating variants to identify the technique that saves most builds in CI.

RQ2: *How does LWD perform in comparison to state-of-the-art static and dynamic batching techniques?* – We aim to identify the technique that is not only simple to use but can also save the most builds in CI.

RQ3: *Does the addition of CI-Skip rules make batching more effective?* – We empirically evaluate the existing rule-based CI-Skip technique both in combination with

our LWD batching variants and state-of-the-art batching techniques. To our knowledge, this is the first study where CI-Skip rules are used in combination with commit batching techniques.

4.3.2 Data Set

To perform an appropriate comparison of our study against those by Bavand et al. (6) and Abdalkareem et al. (3), our data selection approach and data processing operations are inspired by both of these studies.

Following a prior study by Beheshtian et al. (7), Bavand et al. extracted the top 9 projects from TravisTorrent(unknown release date) that have a failure rate less than 20%. They also added to their study 3 additional top projects with a failure rate between 20% and 40% to understand the impact of high failure rate projects on batching. In summary, they studied 12 projects (5 Java and 7 Ruby) with a failure rate between 6.57% and 40.30% of the commits made to their "master" branch. For each project, they also used the first 100 builds to train their Weighted Smoothing algorithm (refer to Chapter 2), leaving all the remaining commits for testing purposes.

Abdalkareem et al. also used TravisTorrent for their study (release 6/12/2016). They focused their study on projects written in Java, filtering their dataset of 1,283 open source projects down to the 393 projects written in Java. By the inherent nature of the TravisTorrent dataset, the projects they examined contained at least 50 builds and at least 10 watchers. After cloning all 393 projects, they evaluated all the commits made in a project after TravisCI was introduced (i.e., all commits made after the .travis.yml file was first added to the project). They were unable to identify the origin of one such project; hence, it was eliminated from the study. In conclusion, they studied 392

projects, written in Java, that contain at least 50 builds.

To perform our study, we investigated the most recently available TravisTorrent data dump (released 25/1/2017), which contained 3,881,992 builds from 948 projects. As much as possible, we combine the data collection processes of both studied projects to perform our empirical study. First, we filtered the data to retain Java-based projects, yielding 401,767 builds from 241 Java projects. The build data of these 241 projects varies from 4 to 46,889 builds. In order to evaluate the effect of our heuristics on large projects with varying failure rates, we then only retained projects with over 2,000 commits, leaving us with a dataset of 53 projects. Following Bavand et al., we further limited ourselves to builds made only on the "master" branch for each project. Similarly, following their study, we also filtered out the first 100 builds from each project while testing across all heuristics. After doing this, 3 of the 53 projects had to be eliminated since they had less than 100 builds in the master branch to begin with, leaving us with a final dataset of 50 projects with failure rates varying from 5.84% to 98.32%.

4.3.3 Variants of Studied Heuristics

We study all LWD batching and the state-of-the-art static and dynamic batching with respect to the same three fallback algorithms, namely BatchBisect, BatchStop4 and BatchDivide4. Hence, both LWD and the state-of-the-art dynamic batching techniques have three variants each – one corresponding to each batching algorithm. Furthermore, each variant of LWD has its sub-variants according to the batch updating technique and factor used (see Table 4.1). We name our LWD batch updating sub-variant with respect to their factor. For example, Exponential-3 is an exponential batch updating variant used with a factor 3.

Table 4.2: Variants of State-of-the-art Heuristics

Heuristic	Batching Algorithm	Batch Sizes	Variants
Static Batching	BatchBisect	2, 4, 8, 16	*BatchBisect-2, BatchBisect-4, BatchBisect-8, BatchBisect-16*
	BatchStop4	4, 8, 16	*BatchStop4-4, BatchStop4-8, BatchStop4-16*
	BatchDivide4	4, 8, 16	*BatchDivide4-4, BatchDivide4-8, BatchDivide4-16*
Dynamic Batching	All algorithms		*BatchBisect, BatchStop4, BatchDivide4*

The variants of static batching, however, are obtained by choosing a fixed batch size. For BatchBisect, batch sizes of 2, 4, 8 and 16 are used in our study, whereas BatchStop4 and BatchDivide4 use batch sizes ≥ 4 (i.e., 4, 8, 16). Each variant of static batching is also named with respect to the batch size used. For example, BatchStop4-8 is used to represent the BatchStop4 algorithm using a batch size of 8 commits.

In RQ3, we also apply CI-Skip rules on incoming commits to remove CI-Skip commits from batches. Algorithm 12 details the detection of CI-Skip commits. We then measure the performance of each studied heuristic - LWD and state-of-the-art static and dynamic batching before and after applying CI-Skip.

4.3.4 Evaluation Methodology

To address our research questions, we constructed a simulator able to chronologically group incoming commits into batches and execute batch builds. Upon a failing batch build, we then apply one of the bisection-based fallback algorithms (i.e., BatchStop4,

Algorithm 12 CI-Skip (Abdalkareem et al. (3))

```
1:  procedure FILLBATCH: (incomingCommit)
2:      i ← 0
3:      batch ← []
4:      fails ← []
5:      batchSize ← 16                          ▷ Default starting batch size
6:      while i < batchSize do
7:          if is_CISkip(incomingCommit) then
8:              merge incomingCommit without CI build
9:          else
10:             batch.append(incomingCommit)
11:             i ← i + 1
12:         end if
13:     end while
14:     BatchBisect(batch, fails)               ▷ or BatchStop4 or BatchDivide4
15: end procedure
```

BatchBisect and BatchDivide4) to these batches and dynamically update the batch sizes based on the rules of either an LWD variant or CI-Skip rules. For the baselines, we relied on the batching simulator in the replication packages from Beheshtian et al. (7) (static baseline) and Bavand et al. (6) (dynamic baseline) to replicate the results of their techniques on our dataset of 50 projects.

For each fallback algorithm, we ran our 5 LWD variants (and their sub-variants) and all variants of the state-of-the-art static and dynamic baseline batching techniques, then recorded the percentage of builds saved, i.e., the percentage of builds avoided in comparison to building each incoming commit individually (TestAll baseline). To further corroborate our findings, we perform statistical tests with 95% confidence ($\alpha = 0.05$) to identify whether there is a significant difference between the percentage of builds saved by different heuristics. We rely on Wilcoxon signed-rank tests for 2 paired distributions and Friedman Test (with post-hoc Conover tests) for >2 paired distributions to compute statistical differences, then evaluate the magnitude of the differences

using the Cliff's Delta effect size and Kendall's W coefficient of concordance, respectively. Following Vargha et al. (53), we interpret Cliff's Delta effect size as: small if $\delta \geq 0.11$; medium if $\delta \geq 0.28$ and large if $\delta \geq 0.43$. Kendall's W coefficient can be interpreted using Cohen's guidelines where $0.1 \leq w < 0.3$ indicates a small effect size, $0.3 \leq w < 0.5$ indicates a moderate effect size and $w \geq 0.5$ indicates a large effect.

To measure and compare the performance of our studied heuristics, we use the performance metric of the median percentage of builds saved by each batching variant across the 50 studied projects. Due to the requirement to fill an entire existing batch before executing a build and bisecting to identify failing commits, batching techniques have a default delay equal to the batch size before the notification of build outcomes to the developer. For this reason, related work on batching techniques (6; 7) does not consider notification delay in their performance analysis, in contrast to the work on build outcome prediction.

4.4 Study Results

4.4.1 Performance Results of LWD Variants (RQ1)

In this section, we explore the performance of our LWD batching sub-variants and determine the batch updating technique that has the best performance when used with LWD. Overall, LWD techniques can save between a median of 26.87% and 49.90% builds (across the 50 studied projects), as shown in Figure 4.1. The highest median percentage of builds is saved using our Linear-4 BatchStop4 batch updating technique, while the lowest median performance is obtained when using the MFU-2 BatchBisect technique.

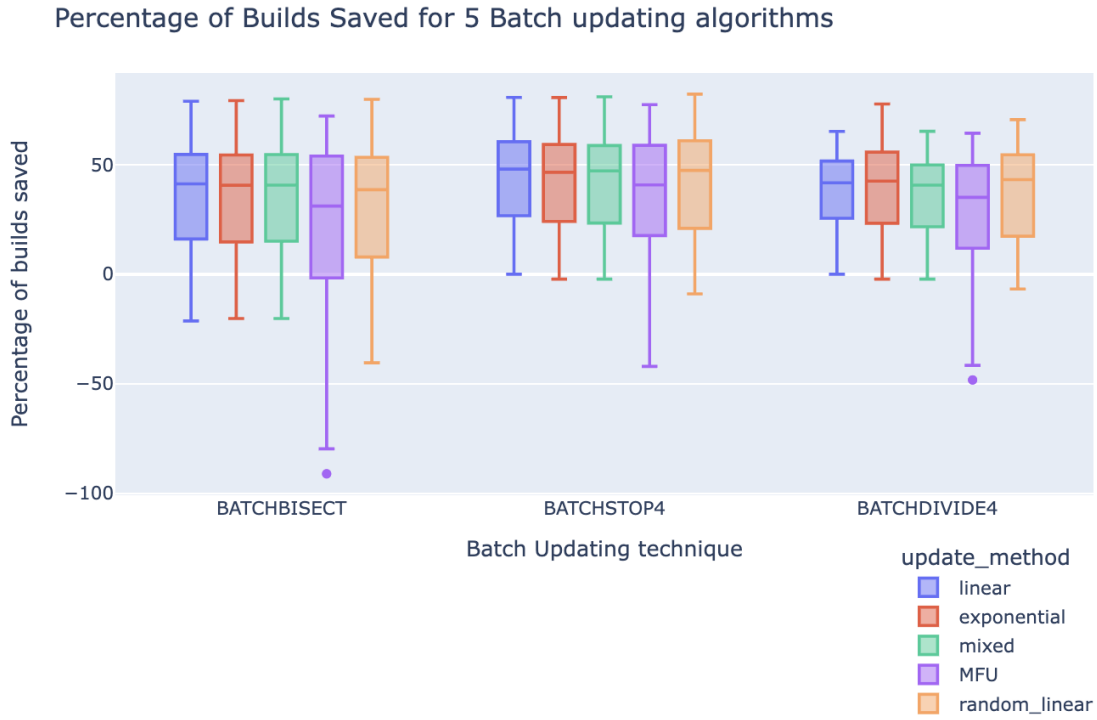


Figure 4.1: Comparing the performance of the LWD variants, with each boxplot covering the distribution of percentage of builds saved across all factor values and projects.

Picking the right factor value is important for BatchDivide4 Linear and BatchStop4 Random LWD variants, while it matters less or not at all for the other variants.

First, for the Linear Batch Updating technique, we performed a Friedman test with a post-hoc Conover test to determine any statistically significant performance differences in the percentage of builds saved for the variants using factor values 1 to 4. While, for each of the 3 fallback mechanisms, a significantly better factor value was identified (see p-values in Table 4.3, and the winning factor values in Table 4.4), the winning value was not consistent. With the BatchDivide4 fallback mechanism, factor 1 was the

best, with a Kendall's W effect size statistic of $w = 0.34$, indicating a moderate effect size. However, with the BatchBisect and BatchStop4 no factor value performed significantly better. Based on the median percentage of builds saved, factor values 2 and 4, respectively, performed slightly better.

Similarly, we found that the Random-Exponential variant outperforms all other Random variants, based on a Friedman test with post-hoc Conover tests comparing the random, linear and exponential subvariants of the Random batch updating technique. Only with BatchStop4 a statistically significant improvement was observed for the Random-Exponential sub-variant, while with BatchBisect and BatchDivide4, Random-Exponential sub-variant performed only slightly better based on its higher median performance.

Finally, to identify any statistical differences in the performance between the factor values 2 and 3 for the Exponential, Mixed and MFU techniques, we used Pairwise Wilcoxon tests (see Table 4.3). Only for the Exponential BatchDivide4 technique a clear statistical difference with small effect size $r = -0.24$ was observed. However, none of the other sub-variants of the MFU, Mixed and Exponential techniques showed any statistical difference, i.e., picking the right factor value does not matter that much for them.

Hence, to determine a factor with better performance for these sub-variants (for use in the remainder of this chapter), we inspect the boxplots of the performance distributions (Figure 4.2) to find factor values with higher median values. We found that Factor 3 is better in this regard than value 2 for Exponential, Mixed and MFU techniques, and select the former value for the remainder of the chapter.

Table 4.3: p-values for for the LWD Batching Updating Variants (Table 4.4 shows the corresponding best factor value). Bold indicates statistical significance for $\alpha = 0.05$

	Linear	Exponential	Mixed	MFU	Random
Test Used	Friedman	Pairwise Wilcoxon	Pairwise Wilcoxon	Pairwise Wilcoxon	Friedman
Post-Hoc Test	Conover	n/a	n/a	n/a	Conover
BatchBisect	**0.0145**	0.88	1	0.759	**1.103e-06**
BatchStop4	**7.5e-08**	0.45	0.74	0.72	**0.005063**
BatchDivide4	**3.004e-11**	**0.02**	0.72	0.72	**2.3e-13**

Table 4.4: Best factor values for each combination of LWD Batching Updating Variant and fallback algorithm, as well as best LWD variant per fallback mechanism. Bold indicates the factor values that were statistically significantly better than other factor values according to Table 4.3, while other factor values were determined based on higher median performance values (no significant improvement).

	Linear	Exponential	Mixed	MFU	Random	Best LWD subvariant	
						p-value	Variant
BatchBisect	**2**	3	3	3	**Exponential**	3.66e-16	Linear-2
BatchStop4	**4**	3	3	3	**Exponential**	< 2.2e-16	Linear-4
BatchDivide4	**1**	**3**	3	3	**Exponential**	< 2.2e-16	Exponential-3

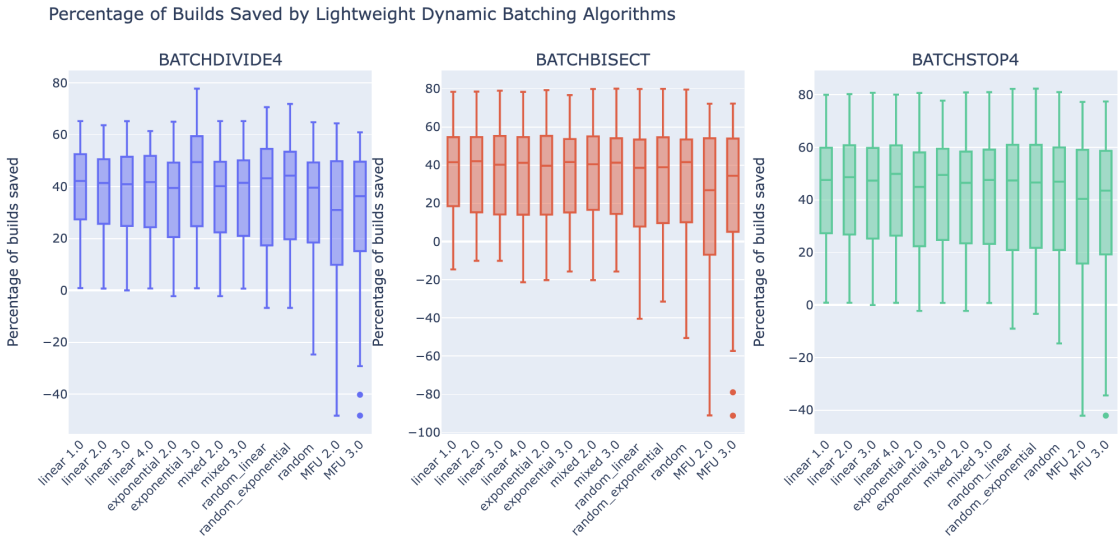


Figure 4.2: Performance of all the lightweight batch updating techniques.

The best LWD sub-variants to work with the BatchBisect and BatchStop4 fallback algorithms are Mixed, Linear and Exponential, while the best one for BatchDivide4 is Exponential.

As reported in Table 4.4, we picked the sub-variants with the best performance identified previously, then perform one Friedman test per fallback algorithm to identify the most successful LWD sub-variant. Based on post-hoc Conover tests (with Bonferroni correction of $\frac{\alpha}{2}$) and box plot analysis, the Random Exponential and MFU-3 subvariants for both the BatchBisect and BatchStop4 algorithms could be ruled out quickly, as they had a significantly lower performance than other sub-variants. Yet, the other three sub-variants, i.e., Mixed, Linear and Exponential performed equally well, even though, based on the box plots of Figure 4.2 the Linear sub-variants perform better (and are used in the remainder of this chapter).

In contrast, in the case of the BatchDivide4 fallback algorithm, the post-hoc Conover

tests were able to distinguisd the Exponential technique as being statistically significantly better than the other sub-variants.

The Linear-4 BatchStop4 and Exponential-3 BatchDivide4 batch updating variants have the highest percentage of saved builds. We perform a final statistical analysis to compare the best techniques of each fallback algorithm, as reported in Table 4.4, to each other. We find that the Linear-2 BatchBisect sub-variant has a significantly lower percentage of saved builds than the Linear-4 BatchStop4 and Exponential-3 BatchDivide4 subvariants. Since the Linear-4 sub-variant has higher median (49.90%), maximum (80.06%) and minimum (0.83%) build savings along with better skewness, we will use this sub-variant in the remainder of this chapter.

Summary of RQ1: The Linear-4 BatchStop4 and Exponential-3 BatchDivide4 LWD variants perform the best of all LWD variants, either statistically significantly or in terms of higher median percentages of saved builds.

4.4.2 Comparing LWD variants with baseline algorithms (RQ2)

This section compares our LWD variants against the state-of-the-art static and dynamic batching techniques.

Our LWD variants save a median of 7.75% more builds than the static baseline technique.

Figure 4.3 compares the performance of the best incarnation of our LWD variants (as listed in Table 4.4) against all variants of the static batching technique. We found that static batching techniques can save a median of 20.73% up to 38.71% builds across

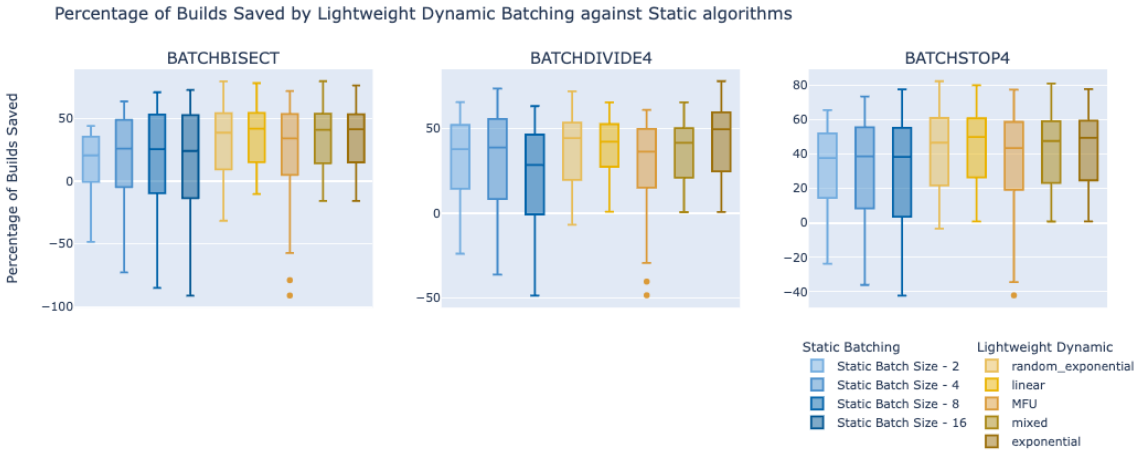


Figure 4.3: Comparison of the performance of the best incarnation of our LWD variants against all variants of Static Batching techniques

the 50 studied projects, while lightweight dynamic batching can save a median of 26.87% to 49.90% of builds.

By analyzing the median of the performance differences between each static batching variant against its corresponding LWD variants (e.g., Static BatchBisect versus Linear-X BatchBisect, Exponential-X BatchBisect etc.), we found that LWD consistently outperforms the static batching techniques for each fallback algorithm. The lowest performance difference between the two families of heuristics is 4.34% with the baseline BatchDivide4 variants, 7.75% with the BatchStop4 variants and the highest performance difference (11.94%) with the baseline BatchBisect static batching variant.

Corresponding Friedman tests (Table 4.5) between each baseline variant and corresponding LWD variants (using the optimal factor values established in Table 4.4), found statistically significant differences (in favour of LWD) for $\frac{39}{50}$ of the studied pairs of heuristics, with effect sizes ranging from 0.1744 to 0.492 (i.e., small to large). However, we did not find any statistical difference between the BatchBisect MFU-3 LWD sub-variant and $\frac{3}{4}$ of the variants of Static BatchBisect technique. Similarly, we did not

Table 4.5: p-values comparing Static Batching Variants(columns) against LWD Batching Variants(rows) (Table 4.4 shows the corresponding best factor value). Bold indicates statistical significance for $\alpha = 0.05$

	BatchBisect				BatchStop4			BatchDivide4		
	2	4	8	16	4	8	16	4	8	16
Linear	**< 2e-16**	**1.8e-13**	**4.6e-09**	**1.2e-12**	**< 2e-16**	**< 2e-16**	**< 2e-16**	**7.2e-03**	0.16	**< 2e-16**
Exponential	**1.0e-14**	**4.2e-11**	**4.7e-07**	**2.4e-10**	**8.4e-16**	**1.2e-13**	**< 2e-16**	**1.7e-10**	**4.2e-08**	**< 2e-16**
Mixed	**< 2e-16**	**3.3e-14**	**1.1e-09**	**2.3e-13**	**4.4e-09**	**2.3e-07**	**1.5e-09**	1.000	1.000	**9.2e-09**
Random	**2.3e-07**	**1e-04**	0.0545	**3e-04**	**3.5e-09**	**1.9e-07**	**1.1e-09**	0.091	1.000	**2.1e-15**
MFU	**0.003**	0.207	1.000	0.456	**0.0024**	**0.0311**	**0.0011**	0.125	**0.0053**	0.0910

find any statistical difference between the baseline static BatchDivide4-8 variant and $\frac{3}{5}$ of our batch updating BatchDivide4-8 sub-variants, including Linear-3, Mixed-3 and Random-Exponential.

On the other hand, we calculated a significant difference in $\frac{7}{10}$ pairs having a variant of random batch updates compared to static batching. The magnitude of this difference was found to range from 0.0416 (negligible) to 0.4064 (medium). This observation indicates that a predetermined batch size is not essential for the best performance of batching techniques, because using random batch sizes can be equally effective.

The LWD variants perform equally well as the baseline dynamic batching technique.

Our LWD variants are able to match the performance of the state-of-the-art dynamic batching technique, while leveraging less resources by avoiding offline simulation and repeated calculations of weighted failure rates, all the while adding the benefit of customizability (i.e., tweaking of factor values).

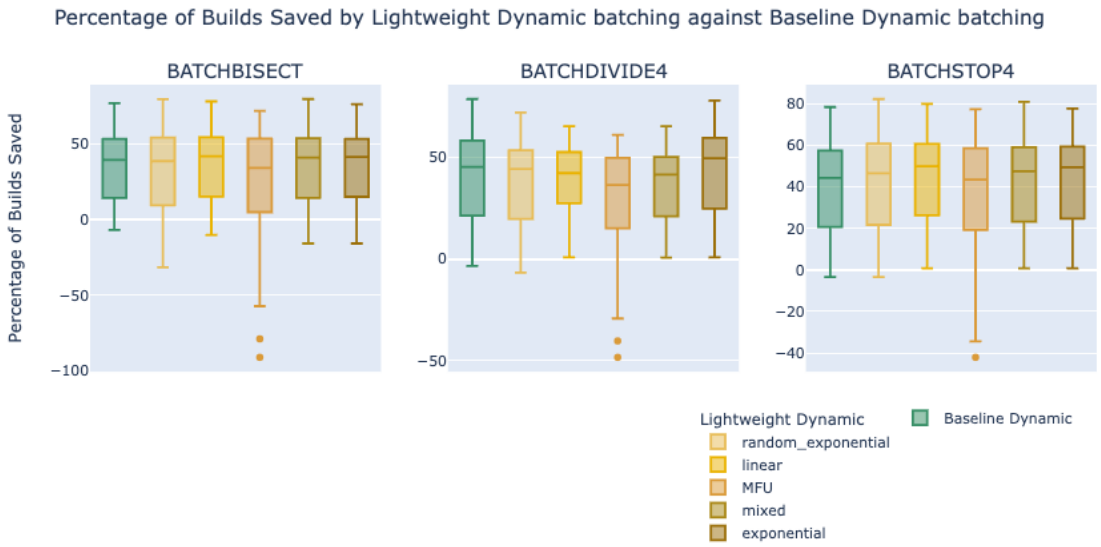


Figure 4.4: Comparing the performance of LWD variants against the baseline Dynamic Batching techniques

Figure 4.4 compares the LWD variants against the baseline dynamic technique for all three fallback algorithms. The performance differences between the two families of heuristics vary from 0 to 5% median builds saved across all algorithms. While using the BatchBisect and BatchStop4 fallback algorithms, at least 69.23% of the LWD variants have a higher median performance than their corresponding baseline dynamic technique. The opposite is true while using the BatchDivide4 algorithm, where the baseline dynamic technique performs better than 92% of our LWD variants in terms of the median percentage of builds saved by these heuristics.

When using a Pairwise Wilcoxon test to compare each corresponding pair of LWD variant and baseline dynamic technique in Table 4.6, we found that although there are some performance differences of up to 5%, they are negligible in the bigger picture. There is no real statistical difference between any of the LWD variants and the baseline technique for any batching algorithm. This indicates that the performance of the two

Table 4.6: p-values of Wilcoxon Tests comparing LWD Batching Variants (rows) against Baseline Dynamic Batching Variants (columns). (Table 4.4 shows the corresponding best factor value). Bold indicates statistical significance for $\alpha = 0.05$.

Algorithm	BatchBisect	BatchDivide4	BatchStop4
Linear	0.884	0.517	0.406
Exponential	0.862	0.543	0.423
Mixed	0.869	0.246	0.569
MFU	0.144	**0.019**	0.643
Random	0.786	0.511	0.595

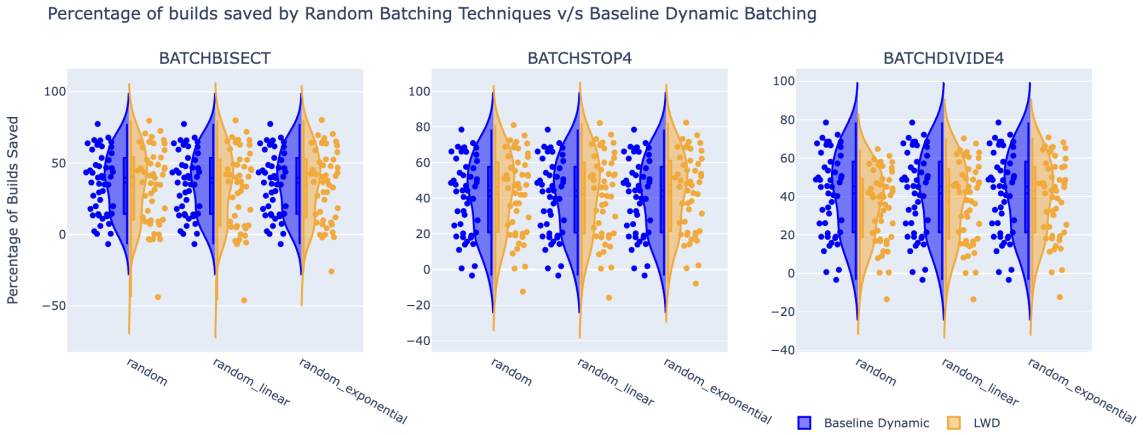


Figure 4.5: Comparing the performance of Random Batch Size Updating variants against the baseline Dynamic Batching techniques

heuristics are similar to each other.

These statistical similarities indicate that the use of simpler LWD variants, some even using random batch size updating techniques, is equally effective as using the state-of-the-art dynamic batching technique. Figure 4.5 compares the random batch updating variants (i.e., Random Linear, Random Exponential and Random) to the baseline dynamic technique, showing that, with the exception of some outliers, data points of the two techniques are distributed similarly across the nine plots.

Summary of RQ2: *The LWD variants save a median of 4.34% to 11.94% more builds than baseline static algorithms. Furthermore, the LWD variants (including the random sub-variants) perform equally well as more complex state-of-the-art baseline dynamic batching techniques.*

4.4.3 Performance results of CI-Skip rule integration (RQ3)

Motivated by the success of the lightweight dynamic batching (LWD) rules, we also evaluate the combination of LWD variants with another lightweight rule-based technique, i.e., CI-Skip rules (3).

Integrating CI-Skip rules with dynamic batching techniques does not correlate with significant build savings.

Figure 4.6 compares the performance of the LWD technique before and after applying CI-Skip rules. From the figure, we see that although the maximum percentage of builds saved using CI-Skip is a median of 0.67% higher than without CI-Skip, the median percentage of builds saved is reduced or remains unchanged. Similarly, in some updating techniques like Exponential and MFU, CI-Skip can also improve the minimum percentage of builds saved, yet in other techniques like Linear, Random and Mixed, the minimum is lower or remains unchanged.

Using Pairwise Wilcoxon tests to evaluate all pairs of CI-Skip vs. non-CI-Skip versions of each LWD variant, we only found statistical differences when combining CI-Skip rules with Linear, Exponential, and MFU. However, the effect size of these statistical differences ranged from -0.016 to -0.02, denoting a negligible difference, where the negative sign indicates that the performance of a variant without CI-Skip is better than that with CI-Skip.

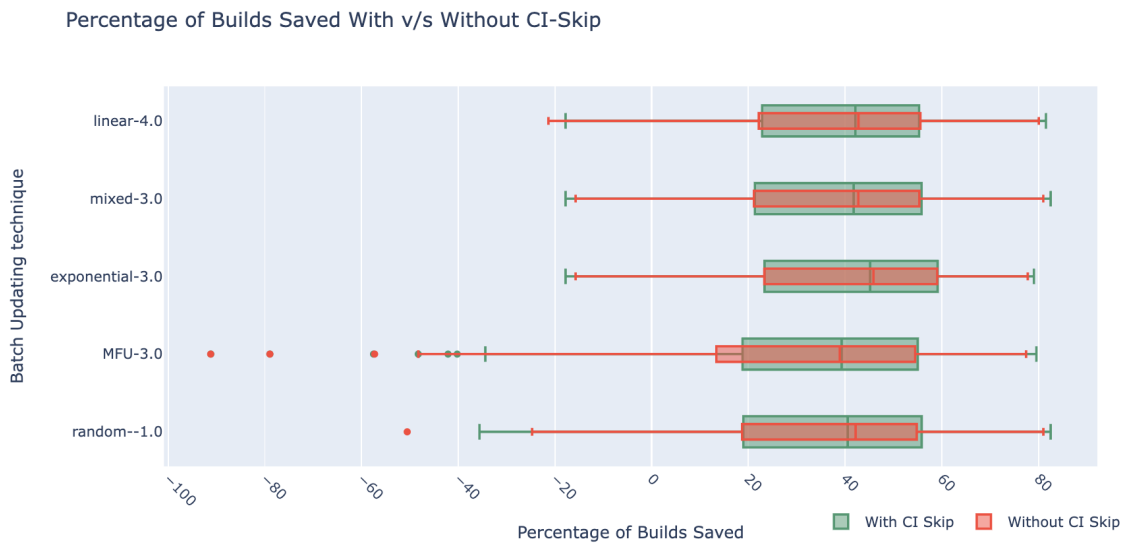


Figure 4.6: Comparing the performance of Lightweight Dynamic Batching technique with and without CI-Skip rules

Similarly, Figures 4.7 and 4.8 compare the performance of the Baseline Static (batch size=16) and Baseline Dynamic batching algorithms with and without CI-Skip rules, showing that in those cases as well CI-Skip rules only are able to save an additional 0.67% and 0.4% of builds compared to the non-CI-Skip variant. We could find no statistical differences between any two variants with and without CI-Skip for these heuristics.

While using CI-Skip rules by themselves helped to save a median of 5.51% builds across the 50 studied projects, the combination of batching with CI-Skip only saves a median of 0.40% builds in addition. Since CI-Skip rules help to filter out simple commits that are highly likely to yield successful builds, the batching heuristic instead has a higher impact on failing commits than on successful commits.

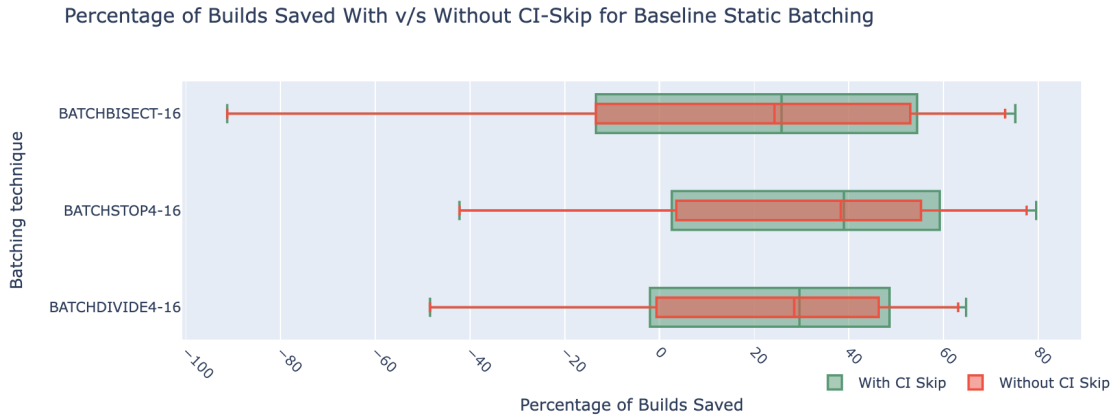


Figure 4.7: Comparing the performance of Baseline Static Batching (batch size 16) technique with and without CI-Skip rules

Summary of RQ3: *The combination of LWD and lightweight CI-Skip rules does not correlate with further build savings.*

4.5 Threats to Validity

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

Construct Validity: Following the studies by Najafi et al. (44), Bavand et al. (6), and Beheshtian et al. (7), we measure the performance of all studied heuristics in terms of the percentage of builds they save, in comparison to individually building 100% of incoming commits. Although useful, this metric does not give an accurate representation of the time, computing resources or energy conserved by the studied heuristics.

Internal Validity: In our study, all simulation scripts, data analysis and processing was conducted by a single author. This could pose a threat of human error and bias due to faulty scripts or missing data. To minimize this risk, we carefully use the replication

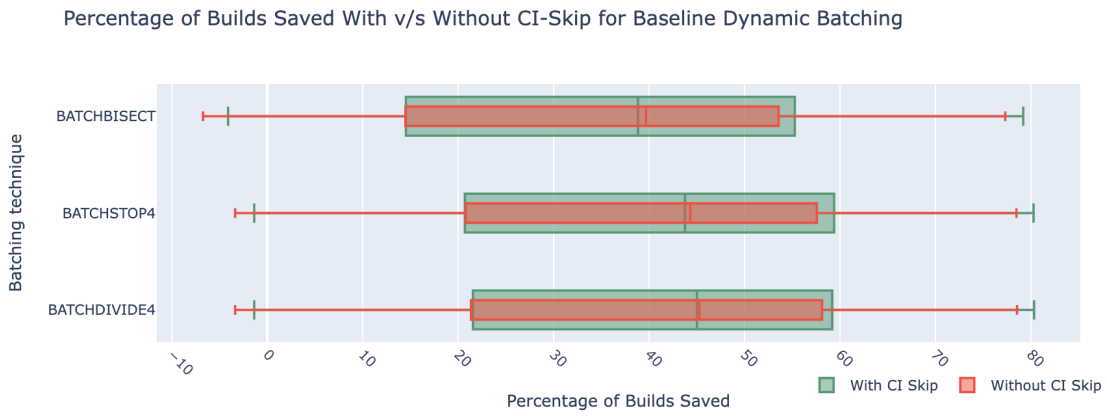


Figure 4.8: Comparing the performance of Baseline Dynamic Batching technique with and without CI-Skip rules

packages by Bavand et al. (6), Beheshtian et al. (7) and Abdalkareem et al. (3).

External Validity: In our work, we examine the impact of CI-Skip rules presented by Abdalkareem et al. (3) on commit batching techniques. These CI-Skip rules are designed for Java-based projects, and also led to us choosing Java projects for our study. Furthermore, we also limit ourselves to large, open-source projects containing at least 2,000 commits before 25-01-2017. However, we minimize the threat this poses by incorporating projects with varying failure rates in our study to generalize our results as much as possible. Further analysis will be needed to estimate the results of our proposed LWD heuristics on smaller projects as well as on projects using different programming languages, build and CI tools. The results may also vary for proprietary projects, which will have to be analysed separately.

Conclusion Validity: We study statistical differences between our studied heuristics (LWD, state-of-the-art static and dynamic batching techniques). We use Wilcoxon pairwise signed-rank test when two groups are compared to each other and use pairwise Friedman tests with the post-hoc Conover test in situations where more than 2

groups are involved. We also use Bonferroni correction wherever applicable to reduce the risk of Type-I errors.

4.6 Chapter Summary

Researchers continue to investigate techniques that help to combat the high cost of CI. Existing heuristics rely on build and test reduction and used techniques that involved machine learning or rule based models. In our study, we mainly investigate the impact of simple, rule-based heuristics and commit batching to reduce the number of builds required to run during CI.

Existing state-of-the-art static and dynamic commit batching techniques emphasize on either using the 'ideal' batch size or using more complex mathematical models to dynamically update batch sizes throughout the CI process. On the other hand, we examined the effect of 'lightweight techniques' that range from simple arithmetic operations to random and conditional operations to dynamically update the batch size used. We find that our proposed techniques perform better than state-of-the-art static batching algorithms and have equal performance to the existing, more complex dynamic batching algorithm.

Amidst the era and boom of AI, our study strives to show that not every question needs to be solved using intelligent methods. However, we understand that the problem of CI build outcome prediction or batch size estimation requires many factors to be considered, and is too random to predict. Any intelligent solution must either accept a reasonable degree of prediction errors or perform better than random techniques. In such CI problems, simple, pragmatic solutions can be equally if not more effective solutions.

CHAPTER 5

Conclusions and Future Work

THIS chapter summarizes our work and presents opportunities for future work.

Continuous Integration is an essential, but resource intensive, process that can simplify and smoothen the code integration and testing phases of software development at the expense of money, time and energy. To deal with this, several heuristics for build reduction have been introduced that provide a trade-off between software quality and cost reduction.

In this thesis, we empirically study the performance of existing build reduction heuristics and evaluate their strengths and weaknesses. We find that existing heuristics add complexities of slower turn-around-time, project-specific restrictions and cost of managing machine learning models. For that reason, we introduce 3 novel heuristics

that can help to reduce the cost of CI. All three heuristics are motivated to enhance existing heuristics by adding simple, history-based features.

We find that we can achieve acceptable savings in CI costs without sacrificing software quality by identifying the right balance between building and skipping commits. In particular, we find that our novel hybrid heuristics can drastically reduce turn-around-time by 96% and improve build savings by 38.21% in comparison to commit batching algorithms. While these savings are lower than for SmartBuildSkip, advantages in terms of lower turn-around-time and faster build failure identifications can help to compensate for this performance difference. We also find that our Lightweight Dynamic batching algorithms can improve build costs by saving a median of 7.73% builds in comparison to traditional batching techniques and can perform just as well as a more complex dynamic batching state-of-the-art approach.

5.1 Thesis Contributions

The main contributions of our thesis are listed below.

1. Two large-scale empirical studies of 79,482 commits from 20 open-source projects and 286,848 commits from 50 open-source projects to evaluate 2 new and 2 existing build reduction heuristics.

2. The ML-CI model, whose performance is on par with that of commit grouping algorithms and requires a median of 49.74% more builds than SmartBuildSkip, albeit with a median skipping delay of 52.5 builds less.

3. The Timeout Rule, which requires 36.47% less builds than Batch4, but 16.46% more than SmartBuildSkip. The median delay induced by the Timeout Rule is,

however, 51.5 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.

4. 13 LWD variants, which save a median of 4.34% to 11.94% more builds than baseline static algorithms. Furthermore, the LWD variants (including the random sub-variants) perform equally well as more complex state-of-the-art baseline dynamic batching techniques. The Linear-4 BatchStop4 and Exponential-3 Batch-Divide4 LWD variants perform the best of all LWD variants, either statistically significantly or in terms of higher median percentages of saved builds.

5. Empirical evidence about the combination of LWD and lightweight CI-Skip rules not correlating with further build savings.

6. Replication packages with scripts and data to simulate the ML-CI model, the Timeout Rule and the Lightweight Dynamic Batching algorithms.

5.2 Future Work

In this section, we discuss potential opportunities for extending our work.

One important direction of future work is on evaluating the real-time performance of our proposed build avoidance heuristics instead of simulating their behaviour on historical data. Currently, our studies, similar to the state-of-the-art, evaluates build reduction heuristics on incoming builds in a sequential manner in a single environment. Yet, in practice, developers have to schedule the builds for a new commit on different build environments. By tracking commits pushed into CI servers like Github

Actions and TravisCI, we can simulate the incoming commits along with their environments to evaluate our build reduction heuristics. Furthermore, the parallel nature of incoming commits (and their builds), with multiple commits undergoing CI in parallel (pre-merge) instead of sequentially, could be evaluated better when integrated in actual CI servers of open-source projects, monitoring incoming commits.

Finally, it is important to explore ways in which the lightweight dynamic batching techniques of Chapter 4 could be combined with the hybrid ML-CI and Timeout rule approaches of Chapter 3. By allowing the use of dynamic batch sizes with the hybrid models, we provide more flexibility to build engineers to optimize their CI processes.

Bibliography

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

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

[3] Abdalkareem, R., Mujahid, S., Shihab, E., and Rilling, J. (2019b). Which commits can be ci skipped? *IEEE Transactions on Software Engineering*, 47(3):448–463.

[4] Barrak, A., Eghan, E. E., Adams, B., and Khomh, F. (2021). Why do builds fail?—a conceptual replication study. *Journal of Systems and Software*, 177:110939.

[5] Basili, V. and Rombach, H. (1988). The TAME project: Towards improvement-oriented software environments. *IEEE Transactions on Software Engineering (TSE)*, 14(6):758–773.

[6] Bavand, A. H. and Rigby, P. C. (2021). Mining historical test failures to dynamically batch tests to save ci resources. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 217–226. IEEE.

[7] Beheshtian, M., Bavand, A., and Rigby, P. (2021). Software batch testing to save build test resources and to reduce feedback time. *IEEE Transactions on Software Engineering (TSE)*, pages 1–18.

[8] Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., and Marinov, D. (2018). Deflaker: Automatically detecting flaky tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 433–444. IEEE.

[9] Beller, M., Gousios, G., and Zaidman, A. (2017). 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)*, pages 356–367.

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

[11] Chen, B., Chen, L., Zhang, C., and Peng, X. (2020). Buildfast: History-aware build outcome prediction for fast feedback and reduced cost in continuous integration. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 42–53. IEEE.

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

[13] Elbaum, S., Rothermel, G., and Penix, J. (2014). Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE)*, pages 235–245.

[14] Elsner, D., Hauer, F., Pretschner, A., and Reimer, S. (2021). Empirically evaluating readily available information for regression test optimization in continuous integration. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 491–504.

[15] Esfahani, H., Fietz, J., Ke, Q., Kolomiets, A., Lan, E., Mavrinac, E., Schulte, W., Sanches, N., and Kandula, S. (2016). Cloudbuild: Microsoft's distributed and caching build service. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 11–20.

[16] Finlay, J., Pears, R., and Connor, A. M. (2014). Data stream mining for predicting software build outcomes using source code metrics. *Information and Software Technology*, 56(2):183–198.

[17] Fowler, M. and Foemmel, M. (2006). Continuous integration.

[18] Gallaba, K., Macho, C., Pinzger, M., and McIntosh, S. (2018). Noise and Heterogeneity in Historical Build Data: An Empirical Study of Travis CI. In *Proc. of the International Conference on Automated Software Engineering (ASE)*, page 87–97.

[19] Ghaleb, T. A., Da Costa, D. A., and Zou, Y. (2019). An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering*, 24(4):2102–2139.

[20] Hassan, A. and Zhang, K. (2006). Using decision trees to predict the certification result of a build. In *Proceedings of the 21st International Conference on Automated Software Engineering (ASE)*, pages 189–198.

[21] Hassan, F. and Wang, X. (2017). 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)*, pages 157–162.

[22] Herzig, K., Greiler, M., Czerwonka, J., and Murphy, B. (2015a). The art of testing less without sacrificing quality. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 483–493. IEEE.

[23] Herzig, K., Greiler, M., Czerwonka, J., and Murphy, B. (2015b). The art of testing less without sacrificing quality. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, page 483–493. IEEE Press.

[24] Hess, M. R. and Kromrey, J. D. (2004). Robust confidence intervals for effect sizes: A comparative study of cohen'sd and cliff's delta under non-normality and heterogeneous variances. In *annual meeting of the American Educational Research Association*, volume 1. Citeseer.

[25] Hilton, M., Tunnell, T., Huang, K., Marinov, D., and Dig, D. (2016). Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 426–437.

[26] Islam, M. and Zibran, M. (2017). Insights into continuous integration build failures. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*, pages 467–470.

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

[28] Jeffrey, D. and Gupta, N. (2006). Test case prioritization using relevant slices. In *Proceedings of the 30th International Computer Software and Applications Conference (COMPSAC)*, pages 411–420.

[29] Jin, X. and Servant, F. (2020). A cost-efficient approach to building in continuous integration. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, pages 13–25.

[30] Jin, X. and Servant, F. (2021a). 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)*, pages 166–167. IEEE.

[31] Jin, X. and Servant, F. (2021b). 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)*, pages 213–225. IEEE.

[32] Jin, X. and Servant, F. (2022). Which builds are really safe to skip? maximizing failure observation for build selection in continuous integration. *Journal of Systems and Software*, 188:111292.

[33] Kamath, D. (2023a). Replication package, lightweight dynamic batching algorithms.

[34] Kamath, D. (2023b). Replication package, timeout rule and ml-ci.

[35] Kazmi, R., Jawawi, D. N., Mohamad, R., and Ghani, I. (2017). Effective regression test case selection: A systematic literature review. *ACM Computing Surveys (CSUR)*, 50(2):1–32.

[36] Kerzazi, N., Khomh, F., and Adams, B. (2014). Why do automated builds break? an empirical study. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 41–50. IEEE.

[37] Kirinuki, H., Higo, Y., Hotta, K., and Kusumoto, S. (2016). Splitting commits via past code changes. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 129–136. IEEE.

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

[Learn"] Learn", S. "feature importances with a forest of trees".

[40] Liang, J., Elbaum, S., and Rothermel, G. (2018). Redefining prioritization: continuous prioritization for continuous integration. In *Proceedings of the 40th International Conference on Software Engineering*, pages 688–698.

[41] Marré, M. and Bertolino, A. (2003). Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering (TSE)*, 29(11):974–984.

[42] Maudoux, G. and Mens, K. (2018). Correct, efficient, and tailored: The future of build systems. *IEEE Software*, 35(2):32–37.

[43] Meyer, M. (2014). Continuous integration and its tools. *IEEE Software*, 31(3):14–16.

[44] Najafi, A., Rigby, P., and Shang, W. (2019). 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)*, pages 279–289.

[45] Ni, A. and Li, M. (2017). Cost-effective build outcome prediction using cascaded classifiers. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*, pages 455–458.

[46] O'Duinn, J. (2013). The financial cost of a checkin (part 2).

[47] Rausch, T., Hummer, W., Leitner, P., and Schulte, S. (2017). An empirical analysis of build failures in the continuous integration workflows of Java-based open-source software. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*, pages 345–355.

[Rehkopf"] Rehkopf", M. "what is continuous integration?".

[49] Seo, H., Sadowski, C., Elbaum, S., Aftandilian, E., and Bowdidge, R. (2014). Programmers' build errors: A case study (at Google). In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 724–734.

[50] Shi, A. W. (2020). *Improving regression testing efficiency and reliability via test-suite transformations*. PhD thesis, University of Illinois at Urbana-Champaign.

[51] Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and Matsumoto, K. (2016). An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1):1–18.

[52] Tufano, M., Sajnani, H., and Herzig, K. (2019). Towards predicting the impact of software changes on building activities. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 49–52. IEEE.

[53] Vargha, A. and Delaney, H. D. (2000). A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132.

[54] Vassallo, C., Schermann, G., Zampetti, F., Romano, D., Leitner, P., Zaidman, A., Di Penta, M., and Panichella, S. (2017). 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)*, pages 183–193.

[55] Wu, Y., Zhang, Y., Wang, T., and Wang, H. (2020). An empirical study of build failures in the Docker context. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, pages 76–80.

[56] Xia, X., Zhou, X., Lo, D., Zhao, X., and Wang, Y. (2014). An empirical study of bugs in software build system. *IEICE TRANSACTIONS on Information and Systems*, 97(7):1769–1780.

[57] Yoo, S. and Harman, M. (2012). Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120.

[58] Zhang, C., Chen, B., Chen, L., Peng, X., and Zhao, W. (2019). 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*, pages 176–187.

[59] Ziftci, C. and Reardon, J. (2017). 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)*, pages 113–122.

[60] Zolfagharinia, M., Adams, B., and Guéhénuc, Y.-G. (2017). Do not trust build results at face value-an empirical study of 30 million cpan builds. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 312–322. IEEE.