Software Batch Testing to Reduce Build Test Executions

Mohammad Javad Beheshtian

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Computer Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

September 2020

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Seyed Mohammad Javad Beheshtian Khabbaz**

Entitled: **Software Batch Testing to Reduce Build Test Executions**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Tse-Hsun Chen

_____ Examiner
Dr. Tse-Hsun Chen

_____ Examiner
Dr. Olga Ormandjieva

_____ Supervisor
Dr. Peter C. Rigby

Approved by _____
Dr. Lata Narayanan, Chair
Department of Computer Science and Software Engineering

September 2020

_____
Dr. Mourad Debbabi, Interim Dean
Gina Cody School of Engineering and Computer Science

Abstract

Software Batch Testing to Reduce Build Test Executions

Seyed Mohammad Javad Beheshtian Khabbaz

Testing is expensive and batching tests have the potential to reduce test costs. The continuous integration strategy of testing each commit or change individually helps to quickly identify faults but leads to a maximum number of test executions. Large companies that have a large number of commits, *e.g.,* Google and Facebook, or have expensive test infrastructure, *e.g.,* Ericsson, must batch changes together to reduce the number of total test runs. For example, if eight builds are batched together and there is no failure, then we have tested eight builds with one execution saving seven executions. However, when a failure occurs it is not immediately clear which build is the cause of the failure. A bisection is run to isolate the failing build, *i.e.* the culprit build. In our eight builds example, a failure will require an additional 6 executions, resulting in a saving of one execution.

The goal of this work is to improve the efficiency of the batch testing. We evaluate six approaches. The first is the baseline approach that tests each build individually. The second, is the existing bisection approach. The third uses a batch size of four, which we show mathematically reduces the number of execution without requiring bisection. The fourth combines the two prior techniques introducing a stopping condition to the bisection. The final two approaches use models of build change risk to isolate risky changes and test them in smaller batches.

We evaluate the approaches on nine open source projects that use Travis CI. Compared to the TestAll baseline, on average, the approaches reduce the number of build test executions across projects by 46%, 48%, 50%, 44%, and 49% for BatchBisect, Batch4, BatchStop4, RiskTopN, and RiskBatch, respectively. The greatest reduction is BatchStop4 at 50%. However, the simple approach of Batch4 does not require bisection and achieves a reduction of 48%. We recommend that all CI pipelines use a batch size of at least four. We release our scripts and data for replication [6].

Regardless of the approach, on average, we save around half the build test executions compared to testing each change individually. We release the `BatchBuilder` tool that automatically batches submitted changes on GitHub for testing on Travis CI [5]. Since the tool reports individual results for each pull-request or pushed commit, the batching happens in the background and the development process is unchanged.

Acknowledgments

I would like to take this opportunity to show my gratitude towards the people who have played an indispensable role in this memorable journey.

Foremost, I would like to express my sincere gratitude and respect towards my thesis supervisor, Dr. Peter Rigby. This work would not have been possible without his guidance, support and encouragement. His undying patience and guidance have helped me in all phases of this journey, from carrying out research to writing this thesis. Sincerely, I could not have asked for a better supervisor.

I would like to thank Concordia University for providing me with an opportunity to be a part of it.

Last but not the least, I would like to thank my parents for their love and constant support.

Contents

List of Figures viii

List of Tables ix

1 Introduction 1

2 Survey of the Literature 5

2.1 Software Testing and Continuous Integration 6

2.2 Test Selection and Prioritization 8

2.3 Statistical Bug Models 10

2.4 Batching 11

2.5 Culprit Isolation and Bisection 13

3 Background on Batching and Definitions 16

3.1 RA 1. TestAll 16

3.2 Batching 16

3.3 RA 2. BatchBisect 19

3.4 RA 3. Batch4 21

3.5 RA 4. BatchStop4 22

3.6 RA 5. RiskTopN 23

3.7 RA 6. RiskBatch 24

4 Data and Methodology 25

4.1 Travis Projects Under Study . . . 25
4.2 Statistical Risk Models . . . 27
4.3 Simulation and Evaluation . . . 28

5 Results **30**

5.1 Result: RA 1. BatchBisect . . . 30
5.2 Result: RA 2. Batch4 . . . 32
5.3 Result: BatchStop4 . . . 32
5.4 Result: Risk Model for RA 4 and RA 5 . . . 33
5.5 Result: RA 4. RiskTopN . . . 35
5.6 Result: RA 6. RiskBatch . . . 36
5.7 Tool implementation on GitHub: `BatchBuilder` . . . 37
5.8 Threats to validity . . . 38

6 Discussion and Future Work **40**

6.1 Related Work . . . 42
6.2 Risk Models . . . 44
6.3 Batching and Bisection . . . 45
6.4 Pooling Medical Tests . . . 46
6.5 Conclusion and Recommendations . . . 46

Appendix A Learning Hyper-Parameters **49**

Appendix B The `BatchBuilder` Tool **52**

B.1 Configuration . . . 52
B.2 Batching and Bisection . . . 53

Bibliography **53**

List of Figures

Figure 3.1 BatchBisect Example 17

Figure 3.2 Min and Max Executions per Technique 17

Figure 3.3 Batch4 Example 18

Figure 3.4 BatchStop4 Example 18

Figure 3.5 RiskTopN Example 19

Figure 3.6 RiskBatch Example 20

Figure 5.1 Build Test Execution Saving vs. Batch Size 31

Figure 5.2 Build Test Execution Saving vs. Risk Threshold 36

List of Tables

Table 4.1 Size of projects under study 25

Table 5.1 Percentage savings in build test executions relative to TestAll 31

Table 5.2 Proportion of total build test execution savings with a given batch size 33

Table 5.3 F-score of each model 35

Table 6.1 Summary of Batching Techniques 41

Table A.1 Optimum hyper parameters value for each project 49

Table A.2 F score and training time before and after hyper parameter tuning 50

Chapter 1

Introduction

Testing is critical but costly quality assurance practice [35]. Tests are run at multiple levels including unit, integration, and system tests [16]. The move to Continuous Integration and Delivery (CI/CD) emphasizes testing individual changes to ensure that problems are found immediately and before release [23]. Beller *et al.* [7] describe the costs involved in CI and finds that on most CI projects the time to run tests takes much longer and consumes more resources than the build and other aspects. This cost is amplified by the running of tests in different environments, *e.g.,* Python 2.7 and 3.7. Furthermore, the most common cause of CI build failure is a failed test which can further length the integration and release cycle.

In some development environments testing each change is infeasible and changes must be batched. For example, at Ericsson, expensive hardware simulation makes testing each change impossible [60]. At Google individual integration tests can run for more than 45 minutes requiring the batching of all recent changes [101]. Even when testing each change is feasible, there are many redundant test runs because changes often require the same tests, test suites, and test environments to be run.

In this work, we build upon the batch testing and bisection work of Najafi *et al.*'s [60] to re-evaluate their existing approaches: BatchBisect and RiskTopN. Based on the results, we introduce three novel approaches, Batch4, BatchStop4, and RiskBatch, to improve the efficiency of testing in CI. We conduct this evaluation on nine large open source projects that use Travis CI [8]. We release the `BatchBuilder` [5] tool that batches pull-request on GitHub for testing on Travis CI. Since

the batching happens in the background and results are reported for each individual pull-request, the development process is unchanged. We briefly introduce each of our research approaches (RA) below.

RA 1. TestAll: Running tests on a single build containing a single pushed change immediately isolates any failing test to the changed code. This approach is simple allowing developers to test each push as a single build in modern CI pipelines [22]. We use the TestAll as the baseline approach because it is in widespread use and does not required builds to be combined and the complexity of bisection on test failure.

RA 2. BatchBisect: TestAll can be prohibitively expensive for large companies with many tests or expensive test hardware. For example, Google [101], Ericsson [60], and Shopify [54] combine commits into a single batch to reduce the total number of test executions. If all the tests pass on a build of size n, then there will be $n - 1$ saved build test executions. In the case of test failure, a bisection is performed until a single build is isolated as the culprit. The execution savings are dependent on the number of test failures that result in bisection. We run simulations to determine the best batch size for a project.

Compared to TestAll, we see a BatchBisect saves between 22.35% and 57.55% of the total build test executions with an average across projects of 46.05%. The best batch size per project ranges from 4 to 8.

RA 3. Batch4: Tooling exists to batch commits and perform bisection on test failure , for example, SandCastle from Facebook [32]. However, bisection adds additional batching and complexity to the CI process. To avoid bisection, we note that batches of size four have the special property: on failure a bisection will cost at least 4 additional executions, which is the same as testing each build individually. We propose the novel Batch4 approach, which groups builds into batches of four saving $n - 1 = 3$ executions when all tests pass. On failure, we do not run a bisection, instead we revert to TestAll which costs 4 additional test executions or $n + 1 = 5$ executions in total.

This simple approach is also very effective at execution reduction. Compared to TestAll, we see that Batch4 saves between 29.51% and 55.84% with an average across projects of 47.63%. Compared to BatchBisect, Batch4 is not only simpler, requiring no bisections, but also outperforms BatchBisect with an average improvement of 1.58 percentage points.

RA 4. BatchStop4: On projects that have few failures, BatchBisect can still be more efficient than Batch4. For example, on the puppet project, the batch size is 8 and requires 5.09 percentage points fewer executions than Batch4. As a result, we introduce BatchStop4, which can make large batches and uses normal bisection until the batch size is four.

Compared to TestAll, we see that BatchStop4 saves between 29.51% and 60.83% with an average across projects of 50.31%. The majority of the savings are achieved with small batch sizes, batch 2, 4, and 8, realizing an average of 72%, 93%, and 99% of the total batch savings. Compared to BatchBisect and Batch4 the average improvement is 4.23 and 2.69 additional percentage points.

RA 5. RiskTopN: When a test fails on a batch, a bisection is required which costs additional execution. Not all builds are equally likely to fail, *i.e.* risky. Models of change risk have been widely used to identify bug introducing changes [25]. Recent work by Najafi *et al.* [60] used risk models to identify commits that had likely failing tests at Ericsson. We reproduce Najafi *et al.*'s [60], RiskTopN approach on nine OSS projects. When a batch fails, the n builds with the highest risk are isolated and tested by themselves. The remaining builds that have a lower modelled risk are tested together in a single batch. The process is repeated until all culprits are found.

Compared to TestAll, RiskTopN reduces executions between 23.23% and 54.80% with an average across projects of 44.17%. However, Batch4 and BatchStop4 both outperforms RiskTopN by 3 and 6 percentage points and do not require a statistical risk model.

RA 6. RiskBatch: In the previous approaches, the batch size is constant for all batches. We introduce the RiskBatch approach that uses a statistical model of risk to continue to add builds to a batch until a risk threshold is reached. Low risk builds will be put into larger batches than high risk builds, and a single high risk build that is above the threshold will be tested individually. In our simulations, we vary the risk threshold.

Compared to TestAll, RiskBatch reduces executions by between 25.93% and 57.43% with an average across projects of 48.50%. RiskBatch outperforms previous risk based approach, RiskTopN by 4.33 percentage points.

The remainder of this thesis is structured as follows. In Chapter 2, we survey the literature. In Chapter 3, we provide the background on batching and define each research approach. In Chapter 4, we discuss our data and outline our evaluation and simulation methodology. In Chapter 5, we

present the results for each research approach. In Chapter 5.7, we introduce our developer tool, `BatchBuilder`, that allows batch testing using GitHub with Travis CI with no visible change to the UI or development process. In Chapter 5.8, we discuss threats to validity. In Chapter 6, we compare our approaches, position the results in the related work, and discuss our contributions.

Chapter 2

Survey of the Literature

We survey the literature in the following areas beginning with general test and bug prediction work and ending with specific work on culprit isolation.

- Software Testing and Continuous Integration
- Test Selection and Prioritization
- Statistical Bug Models
- Batching
- Culprit Isolation and Bisection

2.1 Software Testing and Continuous Integration

Crispin *et al.* [16] categorize software tests into four types:

- unit and component tests (automated)
- functional tests (automated and manual)
- performance, load, and security tests (automated)
- user acceptance tests (manual)

The first category of tests are written by developers and are performed automatically during development and integration. Functional tests are higher level tests that verify a feature that can consist of several components and usually are closer to the business perspective rather than technical logic. These tests include manual and automated parts. The third group of tests verify the technical aspects of software but in contrast to unit tests, they check software as a whole. To perform this group of tests, external tools are often used which are usually automated. The last category is user tests including beta testing, A/B testing, and usability testing. These tests are often done manually because of the need for actual users, however, trends in A/B test have lead to a greater degree of automation [73].

Amannejad *et al.* [2] measure the cost of test automation in the software industry. They find that not all automated tests are cost effective because automation requires additional development effort, can lead to design change, and often increases the number of overall test setups and executions. They suggest a combination of automated and manual tests on software projects. Hoffman [39] also stated that software automation costs are often underestimated especially by management.

Automated tests are run at varying times including after each source code change [21, 91, 27], at a specific time daily [40], or a combination of two previous methods [91].

With the appearance of continuous integration systems, running automated tests has became one of the main parts of the pipeline. Continuous integration and delivery (CI/CD) systems have become popular in both industry and open-source projects because the deployment tasks are automated and developers receive feedback faster, tests are run automatically, and critical updates are delivered to customers more frequently [53, 81, 36, 69].

While CI/CD pipelines are designed to quickly integrate new changes and release them to users, multiple studies [99, 38, 37, 28] show that there are some limitations in automating the integration process including: test reusability, test coverage, resource costs, and developers expectations [72]. CI pipeline must be carefully designed to ensure that the costs do not outweigh the benefits [83]. We survey the papers that consider the pros and cons of CI pipelines below.

Pinto *et al.* [68] propose a survey on CI usages and challenges involving 158 developers. Although, automatic testing can increase software quality, developers say they might skip some tests in CI pipeline because the execution is slow and take a long time. Developers also tend to offload test execution from their system and use CI instead; however, running tests on CI is three times slower than local IDE.

Zhao *et al.* [99] study the impact of continuous integration on software development. They survey several teams on Github that use Travis CI to understand the impact of code change frequency, change size, pull requests duration, number of issues, and testing. Based on their survey, they report that software quality is improved by using test automation and CI. Developers also state that using Travis CI leads to fewer defects. They report that a CI system leads to more frequent commits and smaller bug fix changes. However, for new features developers still tend to commit larger changes.

Hilton *et al.* [38] study the CI benefits and challenges on open source projects. They analyze 34,544 projects and survey 442 developers to understand why and why not software teams use CI pipelines. They report that the computational cost of build test execution and waiting time to receive test results are two of the most important drawbacks of CI pipelines.

In another study, Hilton *et al.* [37] survey over 500 developers from different companies that are working on proprietary source code. They report 76% of respondents feel more productive using CI. 85% of the developers say having a CI pipeline increases the tendency of developers to put more efforts into automated testing.However, 50% of developers have issues with CI troubleshooting and configurations.

Ghaleb *et al.* [28] study the pros and cons of CI systems. Detecting errors in early stages with faster and more frequent releases is a key advantage. However, long CI builds can slow down the development since the developers have to wait for test results from many small changes. To deal with this problem, programmers spend more time on optimizing build and test configuration which

leads to additional work beyond normal development. Using data from 67 Github projects, they created a model to determine which factors lead to long running builds. They found that project size, team size, CI configuration, and test density had the greatest impact on the duration of builds.

Yu *et al.* [97] study CI testing with non-functional requirements (NFR). Testing non-functional requirements, such as latency or resource utilization, requires additional time and effort to design and write these complex tests. They identified 47 papers that introduced tools and methods for NFR testing and found that these tools did not easily integrate into CI pipelines. They proposed a tool and techniques to facilitate tool integration.

Beller *et al.* [7] study the CI usage in over 1,200 open source projects on GitHub. Most CI pipelines have three phases: 1) compilation, 2) static check, and 3) testing. They report that testing takes longer than the other phases and causes most of the build breaks in a CI pipeline. Most of the time, running tests takes less than one minute, although there are some cases that can take up to 30 minutes. Another parameter that increase the testing time is having multiple environments *e.g.,* Python 2 and Python 3. In this case a set of similar tests are run in all of the environments with projects testing five different environments on average drastically increasing test load.

Tomassi *et al.* [87] introduce a tool to extract failures and bug-fixes from Github and Travis CI. They mine fail-pass pairs in Travis builds and reproduce them in a container. Travis makes a branch on the fly (phantom branch) which contains the base branch and merges it with a pull request (or new changes through git push). Then, this new branch is tested against different environments which are defined in the Travis configuration file. Each of these environments creates a Travis job. They consider the final build verdict that reflects the result of all the jobs. In our work we also consider a Travis build as our unit.

2.2 Test Selection and Prioritization

Yoo *et al.* [95] perform a survey of test selection, minimization, and prioritization to deal with resource constraints in test execution. Test suite minimization is the process of removing duplicate test cases. A set of requirements that need to be tested are defined. For each requirement, one test case among multiple ones is selected. If a requirement needs more than one test case, it has to

be broken down into sub-requirements. Choosing requirements and level of abstraction influence the performance and number of failures that slip-through. Test case selection is also performed by choosing a subset of test cases, but in contrast to test minimization, test cases that verify risky or recent changes are chosen.

To increase the efficacy of testing tools and consume less computational power, a subset of tests which are the most relevant could be selected instead of running all the tests in each build [98, 29]. To give feedback to developers about their code as soon as possible, test prioritization is also suggested [61, 31].

Minimization and selection will have failures that slip-through because not all tests are run [35]. In contrast, prioritization approaches run all the tests but change the run order. With prioritization the assumption is that tests can be run in an arbitrary order. However, changing the test order can lead to new flaky failures. Lam *et al.* [52] found that the source of 50.5% of flaky failures is a violation of test order dependencies. In contrast, our approach to reducing test executions comes from grouping builds not from eliminating or selecting a subset of tests. As a result, we guarantee no slip-throughs because we run all the tests. We do not introduce order dependency flaky failures, because the entire test suite is run in its original order. Future work is necessary to directly compare the efficacy of batching with selection and prioritization on the same datasets.

Marijan *et al.* [56] study continuous software testing optimization. Regression tests are helpful in a CI pipeline to prevent degradation in new code changes. However, such tests could take a long time and consume significant resources, especially when the same functionality is tested multiple times and the same set of tests are run. They propose an approach to reduce this test redundancy by learning from testing history and code coverage metrics to predict future faults. They report that testing time can be reduced by 40% on average. They measure three parameters regarding test cases, time efficiency, size efficiency, and fault detection effectiveness. Their prediction models are accurate with an F-score between 88% and 93%.

Shi *et al.* [79] study regression testing and test reduction. They compare test-suite reduction and regression test selection and also evaluate the combination of these two approaches. Test reduction completely removes the test case from the flow whereas regression test selection tries to remove the tests when the test outcome remains the same in different runs. They experiment on 17 open source

projects. They find that test suite reduction can reduce the number of test executions by 40 percentage points. However, in some bugs this reduction comes at the cost of fault slip-throughs. When the approaches are combine, the savings is 5 percentage point but there are fewer slip-throughs.

Kumar *et al.* [50] study test selection and classification. They create a framework that uses fuzzy-ant colony optimization algorithm to optimize test execution. Their approach has three stages. In the first stage, unfit test cases are removed using fuzzy synthesis-based filtering. In the second stage, test selection and classification is done using fuzzy entropy-based filtering which reduces the categorized ambiguous test cases. In the final stage, ant colony optimization is used to search the space to find the optimum set of test cases. They found the third stage is the most important one because it can perform most of the optimization by itself.

Marijan *et al.* [57] propose a test selection approach to prioritize test cases based on configuration coverage analysis at Cisco. They compare this novel approach to the existing process and also a random test selection algorithm. Their results show that their approach is better than the other two methods and can increase fault detection while keeping test feedback delay the same. They found uniformity of configuration increases by 39% and failure detection becomes 15% better.

2.3 Statistical Bug Models

Predicting software defects using machine learning models is a research area which has been popular in recent years [30, 77, 62]. Memon *et al.* [59] propose a study on continuous testing at Google. They report regression testing every single change is not feasible due to the high frequency of changes and long duration of tests. They introduced a tool to mitigate this problem by limiting test workloads and also informing developers about their code influence on quality. They report only a few percentage of tests ever fail and these failing tests are closer to source code that is modified more frequently. Also, some users and tools are more likely to cause failures. This study shows certain attributes of code changes can affect the possibility of test failure. We use this finding to create a learning model using build features to predict future failures.

Chen *et al.* propose a method to predict software defects using a statistical model. They use source code metrics such as number of methods, average method complexity, and number of lines

of code to build their model. The learning model is created using k nearest neighbor (KNN). They evaluated their approach on ten projects and the result shows that it performs better compared to previous methods. This model chooses less features and costs less computational power while the prediction is improved.

Yang *et al.* [94] study software defect prediction using deep learning at change level. Their prediction has two stages: 1) feature selection which is done by extracting a set of features from a broader initial set of feature using Deep Belief Network. 2) building a classifier using the selected attributes. To evaluate, they use cost effectiveness and F-score. The first metric is important when testing resources are limited. There is a trade-off between the allocated testing resources to predict risky changes and the number of suggested lines of code to inspect manually. In our study, we achieve the same goal *i.e.* predicting risky changes, but we also propose another evaluation metric which is based on the number of build test execution.

Hassan [33] studies fault prediction based on complexity of change process which is calculated by history of code modifications. Pandey *et al.* [67] propose an approach to detect software defective modules using deep ensemble learning model. Their approach allocates more testing resources to modules that are more likely bug-prone based on model prediction. To solve the problem of learning from imbalanced data, they perform minority class oversampling. Many studies evaluate the learning algorithm to improve the accuracy of bug prediction models [80, 66, 90, 86]. In our study, we evaluate several machine learning techniques and tune hyper-parameters to increase the performance of our models.

2.4 Batching

Batching is an effective technique to deal with resource constraints, whether it is computational power, development costs, or time. Alexeevich *et al.* [1] propose a method for batching tests during execution of a test suite. Batching is not only useful in test cases but also in test data and test results. Using this approach data connections which are considered as a constraint during testing could be consumed wisely. They focus on optimizing tests for resource-limited mobile devices. Test suites can be lengthy and complex and these devices often have limited processing power which leads to

even longer build and test duration. They find that batching in different phases of testing can reduce the required resources and speed up the process.

Cho *et al.* [15] propose a batching scheme for data processing in Internet of Things (IoT). Due to the limitations in network capacity and processing power of these devices, computational power must be consumed very carefully. Specially when it comes to data transfer, batching can play a significant role to reduce the latency. They propose an adaptive approach to utilize the capacity of devices using a combination of batching and scheduling methods based on previous data.

In medical tests, pool testing *i.e.* batching is an effective way to reduce the number of required test kits and thus decreasing costs. Dorfman [20] proposed an approach to detect infected individuals in a large population during World War II. He suggests pooling test to reduce the cost and the time. If the test is negative, it means all members of that group are tested negative, otherwise each individual needs to be tested separately. Gajpal *et al.* propose an approach to partition people into groups and test each group with one kit. Only, groups with a positive result need to be divided into subgroups and tested further [26]. To improve the pooling process, placing one sample in multiple pools is suggested. Broder *et al.* propose a study on double pooling tests to reduce the cost of isolation process [10]. Viehweger *et al.* also suggest to replicate samples in multiple pools. If a pool tested positive, the samples that are common among other negative tested pools can be skipped from further testing [88]. Wolff *et al.* propose a study on different batching approaches for testing a large population against COVID-19 with a limited number of test kits in a short period of time. To calculate the best pool size, they experiment pool sizes of 1 to 30 and realize most of the saving is achieved with smaller pool sizes. They also found as the infection rate increases, the optimum pool size decreases because the overhead of bisection becomes larger than the saving from batching [19]. Aragón-Caqueo *et al.* study the effectiveness of batching in COVID-19 tests and report that batching gains more saving when the infection rate is lower [3]. In our work, we experiment batch sizes of 1 to 20 to realize the optimum batch size. We evaluate nine projects with various failure rate to express the relationship between failure rate and optimum batch size.

2.5 Culprit Isolation and Bisection

When multiple changes are tested together, in case of failure, we need a mechanism to find the root cause and isolate the culprit change. Git bisection [55] can be used when commits are ordered [82]. It does a binary search to find the failure [9, 101].

Ziftci *et al.* [101] propose a study on isolating failures in a group of changes. They suggest an algorithm to identify the code changes that cause degradation and regression. They experiment on 140 projects at Google and the result shows that this approach can detect faults 82% of the times by suggesting top 5 riskiest changes among thousands in every build. In this work they focus only on test case failures which means they ignore breakage in previous stages such as code compilation. The Google source code repository does not have multiple branch meaning every change goes directly into the main branch and creates a new version. Because of this architecture, it is critical to test each change before merging to the repository as it can break other parts of the code. They call these tests pre-submit tests which are performed before merging happen and in the case of failure the changes will be rejected. They also have post-submit tests that happen after merging on the code base. These tests are often longer and more complex and cannot be done on each change. The number of changes is also large and testing each single change without batching can overwhelm the resources. When batching is performed, if a test fails, it is critical to find the root cause as soon as possible, because it can affect the whole development process. To solve this problem and find the culprit change faster without manual intervention, they create a dependency graph, that shows the relation between test cases and file changes. This graph shows the distance between a file change and test case. Larger changes are more likely to cause the failure as well as changes that are closer to the failed test case. To evaluate the results they asked developers to investigate the suggested culprits and give feedback. In this study, flaky failures are considered as normal failures which means no special processing is done to detect flaky changes/tests. Our study is directly related to this work with some modifications. Instead of having a test selection process we run all the tests each time, but batch builds to reduce tests executions. We do not use file changes to predict failures, instead we use a statistical bug model based on historical build features gathered from GitHub and Travis CI. Another difference is in the evaluation process which we evaluate our

batching approaches by calculating the reduction in number of build test execution compared to test all changes without batching.

Saha *et al.* [75] propose an approach for selective bisection consisting of test selection and commit selection. They state that test selection can reduce the number of test executions while commit selection can reduce the number of compiler invocations. Finally, the combination of the two, can reduce the total time required for testing. To perform selective bisection, they ignore commits that are likely having no effect on failing tests. This prediction is based on the test coverage and done on file level.

Heger *et al.* [34] propose an automatic approach to find the root cause of software performance regressions. Their work has two stages. First, performance regressions are detected using the unit tests that are related to performance. The next stage is finding the root cause of the regression using bisection. The bisection process is similar to Git bisect. For choosing the optimum split point, a call tree is used. To correlate changes with performance regressions, they perform a static analysis that select the related class and methods. They only report the methods that are suspicious to the developers. In our work we perform bisection in three of our batching approaches. We also propose a novel approach that use batching without bisection and show that it can be effective to find the root cause of a failure.

Bendik *et al.* [9] propose an approach to find regression points in a software version control system. Git bisection can be used to find the culprit change in a group of commits. However, in each iteration, it is assumed that there is only one problem exist in a batch. Sometimes, fixing a bug can take several commits and also it is possible that an incomplete fix causes future test failures. To address these issues, they propose an algorithm to determine the regression points in a cost effective manner. They assume that each regression point can be predecessor of the leaves in the version control graph that has failure. They propagate the regression for all influenced leaves and remove the middle nodes from the inspection process. As a result they focus on the latest incident causes by the regression rather than middle changes. In our work, we reduce the cost of bisection that causes multiple test executions by stopping at batch size four. In other word, we do not bisect batches that are smaller than four and show mathematically that this can save build test executions.

Najafi *et al.* [60] study batching, bisection and using bug models to find the root cause of a test

failure. They experiment on three projects at Ericsson. To find the optimum batch size for a software project, they experiment with batch sizes from 1 to 20 and find that batches of size four generate the best result *i.e.* lowest number of test executions. They also consider flaky failures in their simulation and show that increasing batch size leads to more flakiness. They propose a risk-based approach to identify top n riskiest commit in a batch of commits. They only evaluate this approach with batches of size four. After isolating possible culprits, they are tested individually. We replicate this approach in our work with a slight modification. Instead of using fixed batch size of four, we evaluate batches of size 1 to 20. We also, propose a novel risk-base approach that uses dynamic batch size and outperform this top n batching techniques. To predict the risk of each commit, they evaluate two methods. The first one is based on test execution history and correlates the previous test cases' result with changes to predict culprits. The second method is using a logistic regression bug model based on 7 features for each change to determine the likelihood of failure. In our work, we use more sophisticated learning models to predict culprit changes. We experiment with several machine learning models and find that Random Forest outperforms other techniques.

Chapter 3

Background on Batching and Definitions

In this section, we introduce the background on batching, bisection, and statistical risk models. We mathematically show the minimum and maximum number of build test executions required to find the culprit build on failing tests as well as the savings when builds pass. These definitions are complimented by examples for each of our six research approaches.

3.1 RA 1. TestAll

TestAll is the simplest and most common form of running tests in a CI flow. Every change will be tested individually before being merged to the main repository or master branch. The number of build test executions is equal to the number of changes made to the system, n. The number of executions is constant regardless of a pass or fail in a build because the on failure there is only one possible culprit build. Formally, the number of executions for a pass, p, or fail, f, is defined below:

$$\text{TestAll}_p(n) = \text{TestAll}_f(n) = n \tag{1}$$

3.2 Batching

Instead of testing each build individually, we batch builds together and test them in groups. When the batches passes, we need only one test execution:

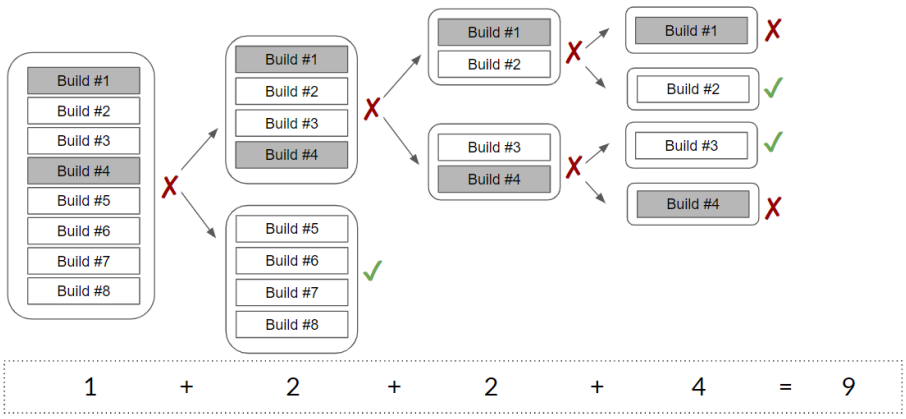

Figure 3.1: BatchBisect. In this example, a batch of size eight builds is tested. The batch fails. Bisection is used to isolate the culprits in two batches of size four. The bottom passes and can be integrated. The top contains two failures that are isolated. In total, we need nine executions to isolate the culprits.

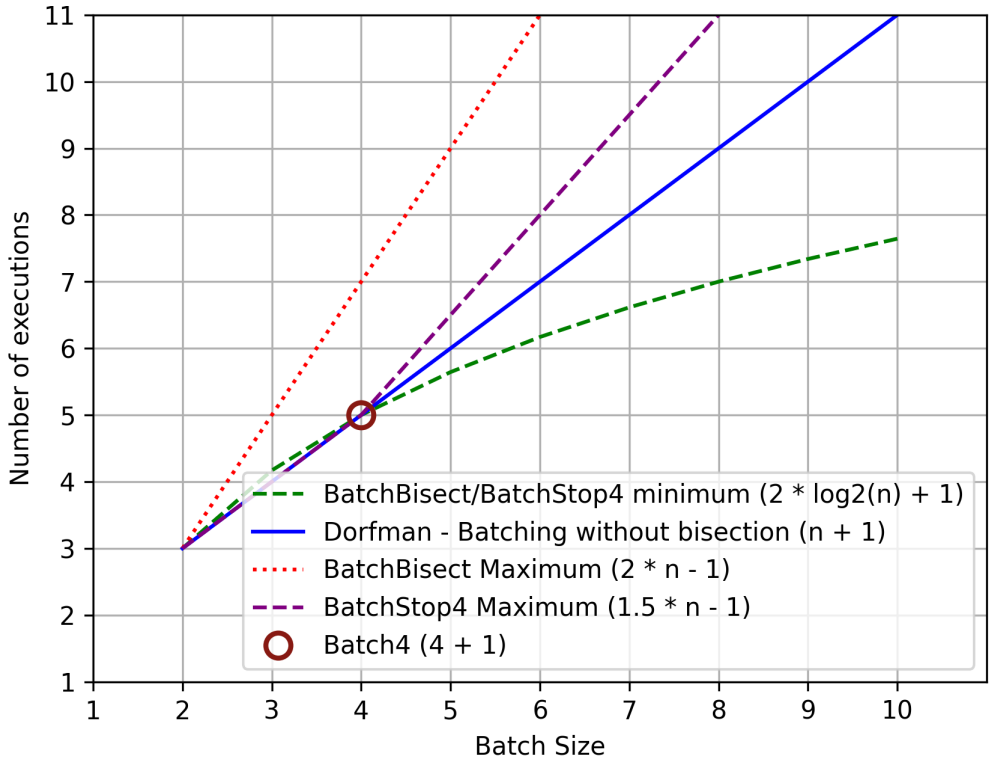

Figure 3.2: On batch failure, the minimum and maximum number of executions required to isolate the culprit build(s) for each approach.

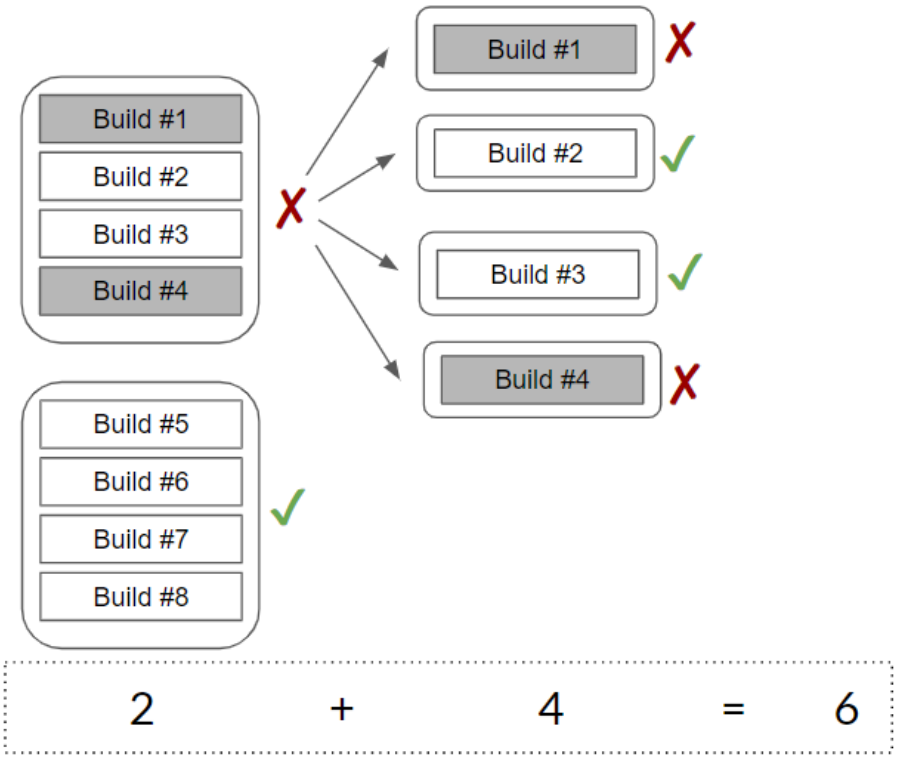

Figure 3.3: Batch4. If TestAll is run on failure the number of executions is constant and equal to the minimum executions for BatchBisect(4). In this example, the first four builds fail, and each is then tested individually for a total of five executions. The second batch passes requiring a single execution. We need a total of six executions, while the same builds required seven executions for BatchBisect in Figure 3.1

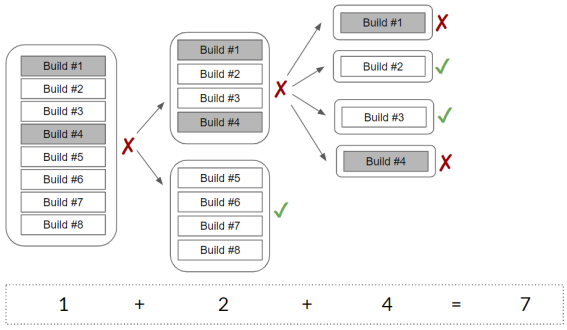

Figure 3.4: BatchStop4. We add a stopping condition for bisection when the batch is size four. For example, the first batch fails and a bisection is performed. In the second batch, Build 1 and 4 are culprits but batch size is four, so instead of bisection, all builds are tested individually. Build 5 to 8 have no failures and there is no need for further test executions. In total, we need 7 execution to find all culprits.

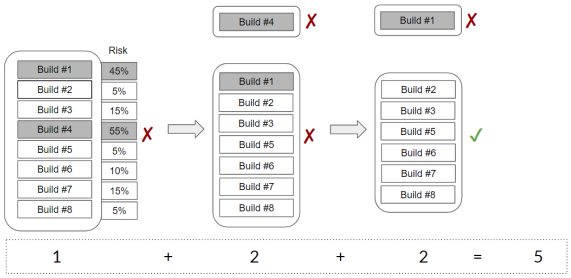

Figure 3.5: RiskTopN. A risk model is run to determine the likelihood of failure for each build [60]. The riskiest N builds are tested in isolation with the remainder tested as a batch. Using RiskTop1 in this example, Build 4 is the riskiest and is isolated for testing. When the remaining batch still fails, Build 1 is now the riskiest. The remaining batch passes. In total, we need five executions to isolate the culprits.

$$\text{Batch}_p(n) = 1 \tag{2}$$

This savings can be substantial. In an extreme example, imagine a project that does a nightly test run on 100 builds, if the build passes the savings in execution will be $1 - n = -99$ or 99 build test executions.

$$\text{BatchSaving}_p(n) = \text{Batch}_p(n) - n = 1 - n = 1 - \text{TestAll}_p(n) \tag{3}$$

However, on failure the culprit must be identified and the number of executions varies depending on the approach.

3.3 RA 2. BatchBisect

When a batch passes only one execution is required to merge the builds in the batch. However, if the batch fails, the build that has failing tests, *i.e.* the culprit(s), must be found using bisection. GitBisection uses a binary search and in so doing assumes ordered commits and that there is only one commit that introduces the failure (*i.e.* we search for the failing commit). However, if there are two commits that have failing tests, then GitBisection would only be able to find the oldest culprit commit. The remaining commits could not be integrated without further testing as a second culprit may also be present. Instead of a search for a single culprit, we need to traverse a binary

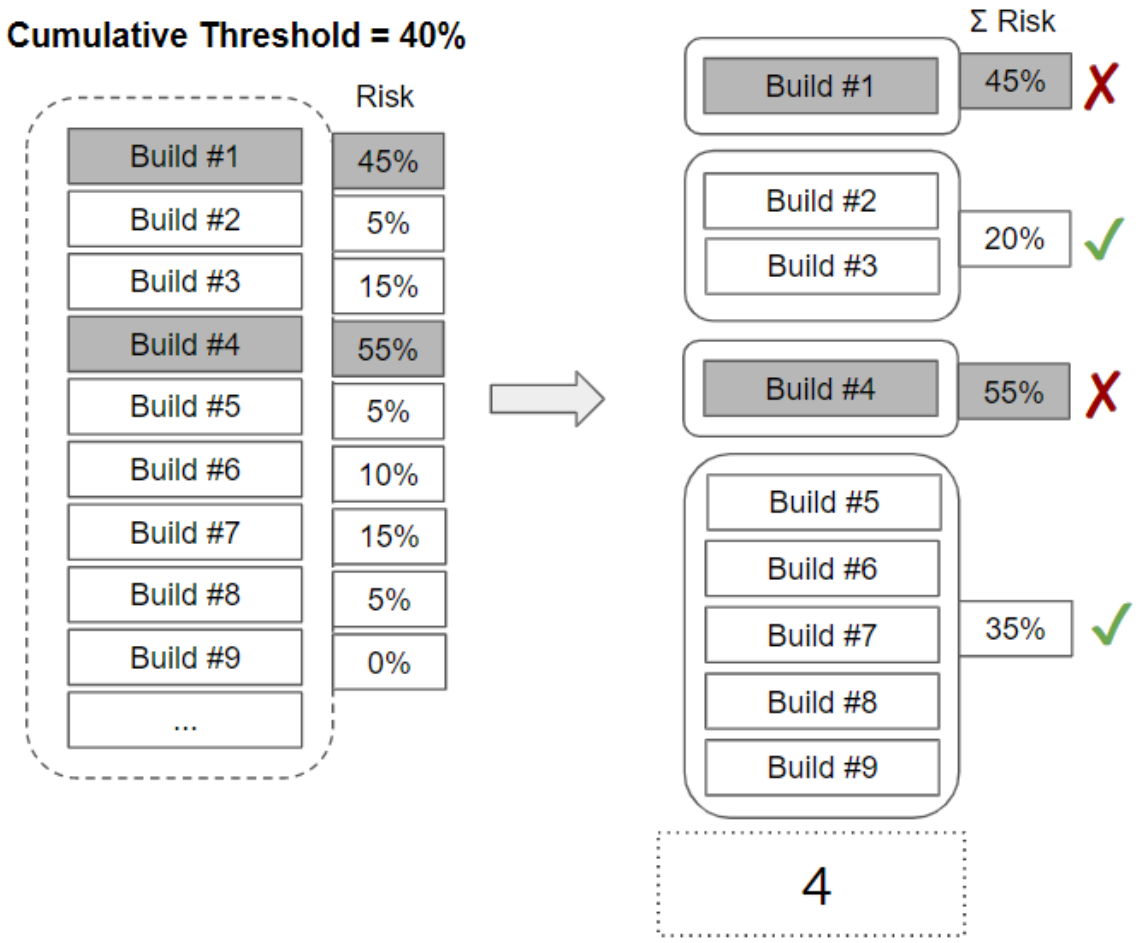

Figure 3.6: RiskBatch. Builds are added to the batch until a threshold is reached. With a threshold of 40%, Build 1 is isolated, while Builds 2 and 3 are tested together because their combined risk is 20%. Adding Build 4 would have a increased the cumulative risk to 75%, so Build 4 is tested individually. The remaining builds have a combined risk of 35%, so they are tested as a single batch. In total we need four executions to isolate the culprits.

tree identifying all the culprits. As a result, in this work we bisect by splitting the builds into equal batches for testing. Figure 3.1, illustrates the process.

Mathematically, we know that the number of test executions required to find a single culprit is the minimum cost on failure. The bottom line in Figure 3.2 shows the number of executions required to find a single culprit for batch sizes between 1 and 10.

$$min(\text{BatchBisect}_f(n)) = 2 * log_2(n) + 1 \tag{4}$$

If all builds in a batch contain a failing test, *i.e.* are culprits, then the number of required executions is equal to the number of nodes in a full binary tree, which is the maximum cost on batch failure. This maximum is shown as the top line in Figure 3.2.

$$max(\text{BatchBisect}_f(n)) = 2 * n - 1 \tag{5}$$

The greater the number of builds that have failing tests, the greater the number of test executions. For example, Figure 3.1, in a batch of 8 that contains two culprits we need 9 build test executions to find both culprits. This is actually larger than the TestAll scenario with one build test execution per build, *i.e.* eight. In Section 5, we run simulations to determine the optimal batch size and execution reduction attained by BatchBisect for the Travis projects.

3.4 RA 3. Batch4

When a batch fails, bisection requires test executions to find the culprits. Given that the bisection is performed using a binary tree, a batch of size four has a special properties that we will discuss. The build test execution reduction when a batch of size four passes is constant at 3 build test executions. For completeness:

$$\begin{aligned} \text{Batch4}_p(4) - n &= 1 - 4 = -3 \\ &= \text{BatchBisect}_p(4)) - n \end{aligned} \tag{6}$$

However, one failure with BatchBisect(4) requires between 5 and 7 executions to identify the culprits. In contrast, the Batch4 approach, that on failure runs the tests on each individual build, *i.e.* TestAll, resulting in a constant number of executions described below.

$$\begin{aligned} min(\text{Batch4}_f(4)) &= max(\text{Batch4}_f(4)) \\ &= n + 1 = 4 + 1 = 5 \\ &= min(\text{BatchBisect}_f(4)) \\ &< max(\text{BatchBisect}_f(4) = 7 \end{aligned} \tag{7}$$

Figure 3.3 provides an example of the Batch4 approach. The first batch has two culprits and BatchBisect would require 9 executions. In contrast, Batch4 requires 6 test executions. The second batch has no culprits so it requires one build test execution. When there is a single culprit, BatchBisect and Batch4 are the same (see Equation 7), however, when there are two or more culprits the Batch4 saves up to 2 executions.

Batch4 is a special case of the Dorfman [20] method introduced during World War II to batch medical tests of, for example, syphilis. The naive Dorfman algorithm combines n soldiers into a single batched test, on failure each individual solider is tested individually, *i.e.* TestAll. In Figure 3.2, we show that Dorfman requires additional executions beyond the minimum for BatchBisect after four builds. In our simulations in Section 3.4, we show that the simple Batch4 approach is highly effective.

3.5 RA 4. BatchStop4

In the previous section, we mathematically showed that bisection with four builds should be replaced by Batch4. We build upon this idea with BatchStop4, which runs normal bisection until the batch size is four, in which case it runs Batch4. For example, in Figure 3.4 the batch size is 8 and Builds 1 and 4 contain failures. After a bisection, the first batch contains the failures while the second batch passes. Since the batch size is four, bisection is no longer performed, instead each build is tested individually *i.e.* TestAll. Total number of execution is 7 for BatchStop4, while the

total for BatchBisect is 9.

With the stopping condition at 4, then the number of executions required to find one culprit is the modified version of Equation 4:

$$\begin{aligned} min(\text{BatchStop4}_f(n)) &= 2 * log_2(n) + 1 - 4 + 4 \\ &= 2 * log_2(n) + 1 \\ &= min(\text{BatchBisect}_f(n)) \end{aligned} \tag{8}$$

While the maximum number of executions on failure is

$$\begin{aligned} max(\text{BatchStop4}_f(n)) &= 2 * n - 1 - (n/2 + n) + n \\ &= 2 * n - 1 - n/2 \\ &< max(\text{BatchBisect}_f(n)) \\ &= 2 * n - 1 \end{aligned} \tag{9}$$

Since we stop bisection when a batch contains 4 builds, the height of the tree is reduced by two with an execution reduction of $n/2+n$. However, we still need to run TestAll on these batches of 4, so we need n additional executions. With one culprit BatchStop4 is equivalent to BatchBisect, however, with additional culprits we can save up to $n/2$ executions. Section 5.3 presents the simulation results and we find that BatchStop4 has the second highest savings of our approaches.

3.6 RA 5. RiskTopN

When a batch fails, bisection requires expensive additional executions. Commit risk models have been used to alert developers to bug-introducing changes that may need additional testing or review [4]. Najafi *et al.* [60] used risk models to isolate the top n riskiest commits to be tested individually while batching the remaining low risk commits.

For example RiskTop1 is illustrated in in Figure 3.5. We see the modeled risk probabilities for

each build with Build 4 has the highest risk, *i.e.* 55% chance of failure, so it is tested individually. The remaining builds are tested in a single batch. The process of testing risky builds in isolation is repeated until all failures are found and passing builds are integrated. Finding the two culprits in our examples take only five build test executions compared to the 9 and 7 required for BatchBisect and BatchStop4 respectively.

Najafi *et al.* [60] created a simple logistic regression model with seven features. In contrast, as we show in our data and methodology, Section 4, we create more sophisticated models, *e.g.,* Random Forest using 19 features. As we discuss in the result, the accuracy of the model dictates the degree of savings (see Section 5.5).

3.7 RA 6. RiskBatch

Najafi *et al.* [60] testing risky builds in isolation, in contrast, we introduce the RiskBatch approach that group builds into a batch up to a cumulative risk threshold. For example in Figure 3.6 we set the risk of failure threshold to 40%. Build 1, with a risk of 45%, is tested individually while Build 2 and build 3 are tested together because their combined risk is 20%. Build 4 could not be added to the previous batch because the combined risk of would be 75%, so build 4 is tested individually. The process is repeated for the remaining four builds that have a combined risk of 35%. In this example, we need four executions to isolate the culprits and integrate the passing builds, compared to the 9 and 5 for BatchBisect and RiskTop1, respectively. The savings is dependent on the accuracy of the risk model, and Section 5.6 presents our results and tuning with various thresholds.

Chapter 4

Data and Methodology

In this section, we describe the projects and data we study. We then describe our statistical risk models. Finally we describe our simulation method and define the outcome measures.

4.1 Travis Projects Under Study

Travis CI is a continuous integration system that is freely available for use by open source projects.[1] The data from the builds of 1,200 open source projects was made available by Travis Torrent [8]. We use the Travis Torrent dataset in this work. In the Travis Torrent dataset, a Travis build can have the following outcomes:

[1]Travis: https://travis-ci.com/

Table 4.1: Size of projects under study

Project	Failure Rate	Tested Builds	Years	Contributors
ruby	22.21%	15,382	5	192
metasploit	7.93%	8,836	4	703
graylog2	10.51%	5,194	4	98
owncloud	16.13%	4,452	2	71
vagrant	9.59%	4,402	4	914
gradle	8.96%	4,018	2	434
puppet	6.95%	3,223	4	532
opal	9.87%	2,980	4	99
rspec	19.36%	2,856	5	274

- “passed:” The code has been successfully tested and no failures have occurred.
- “failed:” The code has been successfully tested but some tests have failed.
- “errored:” There was an error while running the tests. For example, there is a bug in test code, an error in setup test environment, a timeout, or an error returned from git.
- “canceled:” The build has been canceled by the user.

We discard “canceled” builds because a developer manual stopped the test run and we cannot model the reason for this stoppage. We consider “errored” and “failed” builds as failures because environmental failures will also result in a bisection [60].

Using this data, we order projects by the number of builds and select the top nine active projects that have a failure rate below 25%: Ruby, Metasploit, Graylog2, OwnCloud-android, Vagrant, Gradle, Puppet, Opal, and Rspec. We do not consider projects with a failure rate above 25% as batching is not effective with high failure rates [60]. Table 4.1 provides additional descriptive statistics on the projects. The projects have between 2.8k and 15k test builds, there is a wide range of failure rates from 7% to 22%, multiple years of development, and between 67 and 914 contributors per project.

The projects are from diverse software domains and we briefly describe each project. The Ruby project is a popular object oriented programming language that is often used for web development. The Metasploit project is a testing framework used for penetration testing having about 900 exploits for different operating systems. The Graylog2 project is an open source logging system capable of collecting, storing and analyzing logs in production. The OwnCloud-android project is an Android app to access cloud storage provided by an OwnCloud Server. The Opal project is a source to source compiler for converting Ruby code to JavaScript. The Rspec project is a testing framework for Ruby projects focusing on test driven development. The Vagrant project helps to build and manage portable virtual machines and containers such as AWS or Docker containers. The Gradle project is a build automation and dependency management software that supports many languages including Java, C++, and Python. The Puppet project is management software that controls distributed operating systems with a centralized configuration and facilities administrative tasks such as updating software and managing users.

4.2 Statistical Risk Models

Two of our approaches require statistical models: RiskTopN (RA 5) and RiskBatch (RA 6). We develop risk models to identify the builds that are most likely to fail. We use scikit-learn[2] library for this purpose. Change risk modelling has been widely studied to identify faults [25] and bug-introducing changes [48]. Prior work by Najafi *et al.* [60] created a simple logistic regression using seven predictors. In this work, we use more sophisticated models and additional features. The dataset has 61 features for each Travis build. We exclude all features that are available only after the tests have been run, including number of failed tests, number of skipped tests, and test duration. We also exclude unique features including the commit hash, date, and project level features, such as the team size that would be constant across all project builds. In total we have 19 features in total, which we describe briefly for completeness.

(1) gh_is_pr: true if this build is started by a pull request otherwise false.

(2) gh_num_commits_in_push: Number of commits in the push that started the build.

(3) git_prev_commit_resolution_status: String, ”merge found” if this build is a merge otherwise ”build found”.

(4) git_num_all_built_commits: Integer, Number of all commits in this build.

(5) gh_num_commit_comments: Number of comments on all commits in this build on GitHub.

(6) git_diff_src_churn: Number of modified lines of source code.

(7) git_diff_test_churn: Number of modified lines of test code.

(8) gh_diff_files_added: Number of files added.

(9) gh_diff_files_deleted: Number of files deleted.

(10) gh_diff_files_modified: Number of files modified.

(11) gh_diff_tests_added: Number of test cases added.

[2]https://scikit-learn.org/

(12) gh_diff_tests_deleted: Number of test cases deleted.

(13) gh_diff_src_files: Number of source files changed.

(14) gh_diff_doc_files: Number of documentation files changed.

(15) gh_diff_other_files: Number of other files changed (other than source code and documentation).

(16) gh_num_commits_on_files_touched: Total number of commits on the files touched in this build in previous 3 months.

(17) gh_sloc: Total number of lines of source codes in the repository.

(18) gh_asserts_cases_per_kloc: Number of assertions per 1000 gh_sloc.

(19) gh_by_core_team_member: True if the triggering commit was by a core team member. (Someone who has committed code at least once in previous 3 months)[8]

The outcome of our risk model is the probability that a build will fail one or more tests. We evaluated five classifiers: random forest, Naive Bayes, MLP, logistic regression, and SGD.

4.3 Simulation and Evaluation

The Travis dataset provides the test verdict for each individual build. Failed builds must be investigated while passing builds are integrated. To simulate the impact of our batching approaches on the number of build test executions, we use the verdict of each build, and combine builds based on the the approaches described in Section 3.

Our simulated batches contain only the builds that have been flagged as ready for integration with Travis CI. We do not introduce any new conflicts when we create batches because any conflict would have been dealt with when the developer ensures that the code can be merged in the pull request prior to submission to Travis CI.

We only combine builds that have the same Travis CI configuration, *e.g.,* that request the same dependencies and environment. If two builds have different configurations, we cannot combine

them in a batch. For example, a build that requires postgres cannot be combined with one that requires MySQL. We only batch builds with identical configuration files.

For risk based approaches, we must train a risk model, and we use the first month of data for training. As we discuss in threats to validity, we experimented with larger training time periods, but found that one month was equal or better than longer time periods. To compare with the other approaches, we also ignore the first month in non-risk approaches.

The goal of this work is to identify failing builds and integrate passing builds with a minimal number of build test executions. We report the percentage decrease in build test executions for each research approach, A, relative to the total number of builds that must be tested, *i.e.* the TestAll approach, according to the following equation:

$$\begin{aligned} \text{ExecutionReduction(A)} &= 1 - \frac{\text{Executions(A)}}{\text{TotalBuilds}} \\ &= 1 - \frac{\text{Executions(A)}}{\text{Executions(TestAll)}} \end{aligned} \tag{10}$$

We also report the additional savings for each approach relative to the total number of builds. This is equivalent to calculating the differences in percentages, *i.e.* percentage point difference, for each approach. We use the equation below to calculate the the additional savings for approach, A2, given approach, A1, and the number of TotalBuilds:

$$\begin{aligned} &\text{AdditionalReduction(A2 — A1)} \\ &= (1 - \frac{\text{Executions}(A1)}{\text{TotalBuilds}}) - (1 - \frac{\text{Executions}(A2)}{\text{TotalBuilds}}) \\ &= \frac{\text{Executions}(A2) - \text{Executions}(A1)}{\text{TotalBuilds}} \\ &= \text{ExecutionReduction}(A1) - \text{ExecutionReduction}(A2) \\ &= \text{PercentagePointDifference}(A2, A1) \end{aligned} \tag{11}$$

Chapter 5

Results

In this section, we discuss the simulation results for each of our approaches relative to testing each batch, *i.e.* TestAll, in Table 5.1. The approaches are BatchBisect, Batch4, BatchStop4, RiskTopN, and RiskBatch. We also discuss the additional savings for between approaches relative to the total builds, *i.e.* the change in percentage points between approaches.

5.1 Result: RA 1. BatchBisect

Batching commits is widely used for integration testing and when the tests are long running or expensive [101]. Najafi *et al.* [60] empirically showed that batching commits and using a bisecting process to isolating the failing commit is effective at Ericsson with a savings in build test executions of 7%, 14%, and 41% depending on the project. We reproduce the result on nine large projects hosted on Travis CI. We run simulations with batch sizes between 1 and 20 builds and plot the saving in build test executions in Figure 5.1. From the execution saving curve in the figure, we note a logarithmic improvement with the majority of the savings coming from small batches sizes. At a batch size of 8, we see that at a minimum 97% of the total executions savings has been achieved. On the projects that Najafi *et al.* [60] studied, the improvements began to decrease with larger batch sizes. We see a similar trend on the rspec and ruby projects that have the highest failure rates. The remaining project plateau with larger batch sizes resulting in little to no improvement in execution savings. As a result, we report the saving results at or below 8 for the remainder of the thesis. The

Table 5.1: Percentage savings in build test executions relative to TestAll. We see on average the techniques save slightly less than half the build test executions. The best performing approach is RiskBatch. Batch4, which does not require bisection or a risk model, performs well and is simple to implement.

Project	Batch Bisect	Batch4	Batch Stop4	Risk Top2	Risk Batch
ruby	22.87%	32.97%	32.97%	29.96%	33.77%
metasploit	52.37%	51.46%	54.64%	50.60%	53.54%
graylog2	52.05%	52.17%	55.69%	49.39%	55.20%
owncloud	53.82%	53.42%	57.98%	54.80%	57.43%
vagrant	57.55%	55.84%	60.83%	50.35%	55.27%
gradle	48.49%	49.21%	50.92%	41.91%	49.29%
puppet	57.16%	54.86%	59.34%	50.85%	56.26%
opal	47.81%	49.19%	50.91%	46.40%	49.84%
rspec	22.35%	29.51%	29.51%	23.23%	25.93%
Minimum	22.35%	29.51%	29.51%	23.23%	25.93%
Average	46.05%	47.63%	50.31%	44.17%	48.50%
Maximum	57.55%	55.84%	60.83%	54.80%	57.43%

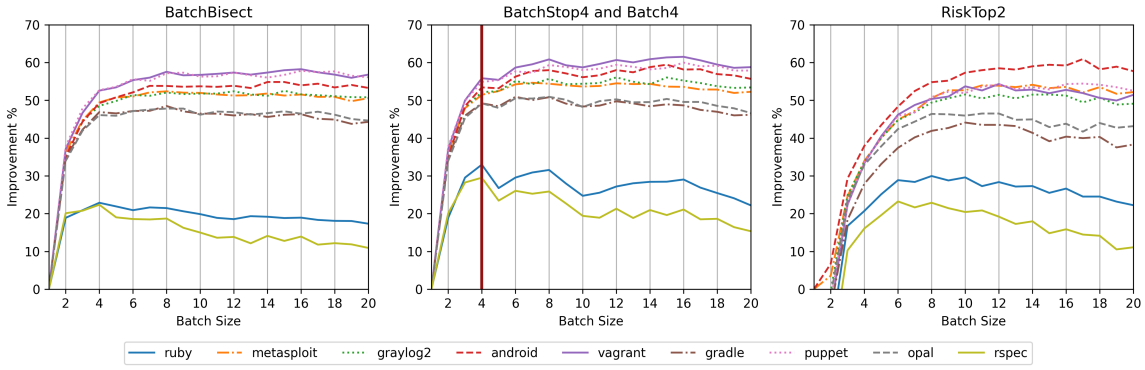

Figure 5.1: Savings in number of build test executions for each batch size. We see that much of the savings is achieved with small batch sizes. Batch4 is represented as a vertical line in the middle figure. Projects with higher failure rates see a decrease in savings with large batch sizes, while most projects plateau.

best batch sizes are 4, 8, 8, 7, 8, 8, 8, 8, and 4 for each project respectively and the corresponding execution savings are 22.87%, 52.37%, 52.05%, 53.82%, 57.55%, 48.49%, 57.16%, 47.81%, and 22.35%, respectively, with an average of 46.05%.

> Compared to TestAll, we see a BatchBisect saves between 22.35% and 57.55% of the total build test executions with an average across projects of 46.05%. The best batch size per project ranges from 4 to 8, with the majority of the savings realized with small batch sizes.

5.2 Result: RA 2. Batch4

In Section 3.4, we mathematically showed that batches of four builds save three executions when they pass, but can require between five and seven executions on failure. However, if we simply test all the builds individually on failure, we always need five executions which is the same as the minimum number of executions for BatchBisect. Furthermore, the Batch4 approach does not require bisection and the complexity of regrouping commits inherent in this process. In Figure 5.1, we see the vertical line represents the savings for Batch4 which are 32.97%, 51.46%, 52.17%, 53.42%, 55.84%, 49.21%, 54.86%, 49.19%, and 29.51% per project.

> Compared to TestAll, we see that Batch4 saves between 29.51% and 55.84% with an average across projects of 47.63%. Compared to BatchBisect, Batch4 is not only simpler, requiring no bisections, but also outperforms BatchBisect with an average improvement of 1.58 percentage points.

5.3 Result: BatchStop4

For BatchStop4, we use bisection until there are only four builds in a batch at which point we revert to Batch4 and TestAll on failure as discussed in Section 3.3. Figure 5.1, shows the simulation results. The execution saving compared to TestAll are 32.97%, 54.64%, 55.69%, 57.98%, 60.83%, 50.92%, 59.34%, 50.91%, and 29.51% for each project respectively, with an average of 50.31. These savings are achieved by choosing batch sizes: 4, 7, 8, 8, 8, 6, 8, 8, and 4, respectively.

Table 5.2: Proportion of total build test execution savings with a given batch size for BatchStop4. We also include the optimal batch size in the final column. Small batch sizes account for the vast majority of the savings, with at least 97% of the savings achieved with a batch size of 8.

Project	Size = 2	Size = 4	Size = 6	Size = 8	Best Size
ruby	0.83	1.00	1.00	1.00	4
metasploit	0.71	0.93	0.99	0.99	7
graylog2	0.68	0.92	0.98	0.99	12
owncloud	0.62	0.87	0.93	0.97	15
vagrant	0.61	0.87	0.93	0.98	16
gradle	0.75	0.97	1.00	1.00	6
puppet	0.64	0.89	0.94	0.98	16
opal	0.74	0.97	0.99	1.00	8
rspec	0.88	1.00	1.00	1.00	4
Minimum	0.61	0.87	0.93	0.97	4
Average	0.72	0.93	0.97	0.99	9.77
Maximum	0.88	1.00	1.00	1.00	16

Compared to BatchBisect and Batch4, we see a reduction of 4.26 and 2.69 percentage point in number of build test executions.

Again the figure shows a logarithmic improvement. In Table 5.2 we show the percentage of total savings for each batch size. With a batch size of 2 we have already realized an average of 72% of the total savings, by batch 4 we see an average of 93%, and by Batch size 8 the average savings is 99%. It is clear that the largest gain in savings comes with small batch sizes and that larger batch sizes provide little further advantage and in some cases require extra executions.

Compared to TestAll, we see that BatchStop4 saves between 29.51% and 60.83% with an average across projects of 50.31%. The majority of the savings are achieved with small batch sizes, batch 2, 4, and 8, realizing an average of 72%, 93%, and 99% of the total batch savings. Compared to BatchBisect and Batch4 the average improvement is 4.23 and 2.69 additional percentage points.

5.4 Result: Risk Model for RA 4 and RA 5

The RiskTopN and RiskBatch depend on a risk model of how likely a build is to fail. RiskTopN then tests the riskiest N builds in isolation, while RiskBatch groups builds into until a cumulative risk

threshold is reached. We described the 19 features that we included in our risk model in Section 4.2. Najafi *et al.* [60] used 7 features and a simple logistic regression. In contrast, we evaluate five classifiers: Naive Bayes, Random Forest, Multilayer Perceptron (MLP), logistic regression and Stochastic Gradient Decent (SGD). We did not use decision trees because they are not designed to provide a probability for the prediction and would not be able to create risk thresholds need to create batches [92, 17]. Table 5.3 shows the F-score for each model. We see that Random Forest out performs the other predictors on all projects except gradle where it is 1 percentage point worse than SGD. As a result, we use Random Forest in the remainder of this thesis. For completeness we report the precision and recall for Random Forest. The precision is 0.51, 0.29, 0.40, 0.46, 0.30, 0.14, 0.23, 0.23, and 0.30 for each project respectively. The recall is 0.55, 0.18, 0.33, 0.37, 0.25, 0.09, 0.14, 0.16, and 0.27, respectively.

We tuned the parameters for random forest. For *number of trees* we experimented the values of 10, 50, 100, 200, and 400 and found a differences in F score between 2 and 4 percentage point. For *maximum depth* of the trees we evaluated the model with the values of 10, 20, 50, 100, 200 and no limit. The difference in F score was between 0 and 4 percentage point. For the *criterion* parameter, we experimented gini and entropy and found a the default gini function was the best choice in all of the projects. For *minimum samples split* we experimented the values of 2, 5, 10, 20, 50, 100. The default value of 2 had the best result in 8 of the projects. One of the projects had the best result with the value of 10 although the difference was 1 percentage point in F score. For *minimum samples leaf* we evaluated the values of 1, 2, 5, 10, 20, 50, 100 and found the default value of 1 generates the best result in all of the projects.

An accurate risk model will reduce the number of executions, while an inaccurate model can even increase the number of executions to find culprits. However, unlike bug prediction that can result in a developer investigating a commit that does not introduce a bug, *i.e.* a false positive, our risk models are used to automatically batch builds. The failing build will always be found and an inaccurate risk model will simply require more executions but will never change the final outcome, *i.e.* it will never add a false positive or negative.

Table 5.3: Comparison of F-scores for each model and project. Regardless of F-score all failing builds are found. Low F-scores result in more build test executions. Precision and recall for Random Forest are in the text.

Project	Random Forest	Naive Bayes	MLP	Logistic Regression	SGD
ruby	**0.53**	0.31	0.35	0.31	0.35
metasploit	**0.23**	0.03	0.08	0.03	0.08
graylog2	**0.36**	0.31	0.20	0.33	0.28
owncloud	**0.41**	0.32	0.28	0.14	0.11
vagrant	**0.28**	0.17	0.11	0.22	0.17
gradle	0.11	0.05	0.11	0.06	**0.12**
puppet	**0.18**	0.05	0.14	0.05	0.06
opal	**0.19**	0.09	0.03	0.18	0.13
rspec	**0.29**	0.22	0.19	0.25	0.19

5.5 Result: RA 4. RiskTopN

The RiskTopN approach isolates the riskiest builds to be tested in isolation, while testing the less risky builds in a batch. We reproduce Najafi *et al.*'s [60] Ericsson study on Travis CI projects using more predictors and a random forests (Our models are discussed in Section 5.4.) Najafi *et al.*'s [60] RiskTopN used fixed batch size of four and $N = 2$. We evaluate $N = 1, ..., 10$. $N = 2$ produced the best results for all projects. We also experimented with batch sizes from 1 to 20.

Figure 5.1 shows the execution savings for each batch size. Like the other approaches, we see that the majority of the savings are at batch 8, so for comparison purposes we report the results at batch size 8 in the thesis. The improvement over TestAll is 29.96%, 50.60%, 49.39%, 54.80%, 50.35%, 41.91%, 50.85%, 46.40%, and 23.23% respectively. On all projects, the savings in executions is lower than Batch4 and BatchStop4 which do not require a risk prediction model. Despite the use of more advanced models and predictors than Najafi *et al.* [60], the results do not justify the addition of a risk prediction model in the CI pipeline.

Compared to TestAll, RiskTopN introduced by Najafi *et al.* [60] reduces executions between 23.23% and 54.80% with an average across projects of 44.17%. However, Batch4 and BatchStop4 both outperforms RiskTopN by 3 and 6 percentage points and do not require a statistical risk model.

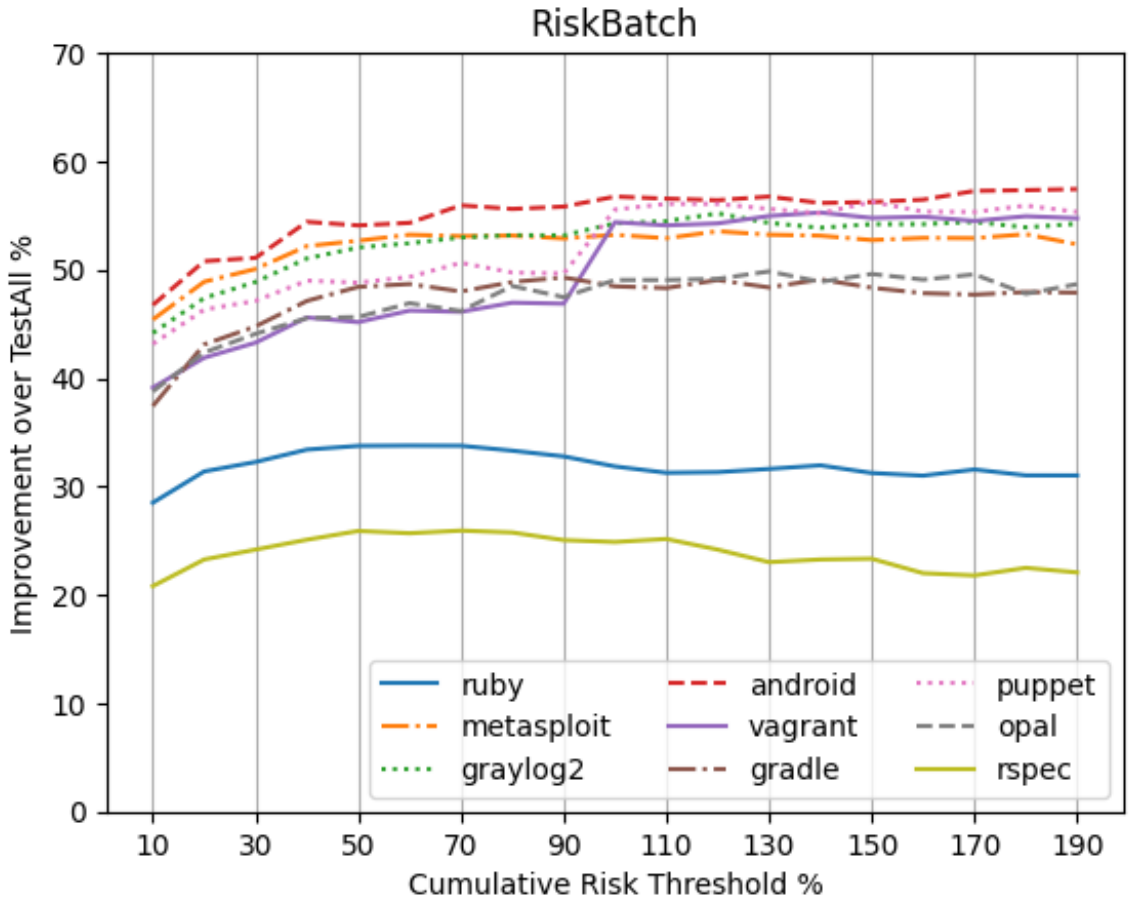

Figure 5.2: Experimenting with the RiskBatch cumulative risk threshold. We see that most projects have at or above 90%.

5.6 Result: RA 6. RiskBatch

Instead of isolating risky builds, our RiskBatch approach adds builds to a batch until the sum of the builds added to the batch reaches a threshold. Section 3.7 and Figure 3.6 illustrate the process. We varied the cumulative risk threshold of failure from 10% to 200% in steps of 10 percentage point increases. We find that the best thresholds are 90%, 120%, 120%, 170%, 140%, 90%, 110%, 130%, and 80% for each project respectively. The cumulative risk is often over 100% indicating that although the model predicts a high cumulative risk of failure, the strategy of making large batches appears to outweigh this risk. However, Figure 5.2 plots the execution improvement for each threshold and shows that low risk threshold are also reasonably effective.

Compared to TestAll, the reduction in number of executions are 33.77%, 53.54%, 55.20%,

57.43%, 55.27%, 49.29%, 56.26%, 49.84%, and 25.93% respectively. RiskBatch outperform previous risk based approach, RiskTopN by 4.33 percentage points.

> Compared to TestAll, RiskBatch reduces executions by between 25.93% and 57.43% with an average across projects of 48.50%. RiskBatch outperforms RiskTopN by 4.33 percentage points.

5.7 Tool implementation on GitHub: `BatchBuilder`

We created the `BatchBuilder` tool to implement the BatchStop4 approach for use by developers. The tool integrates with GitHub pull requests and runs Travis CI. We release the source code [5]. After configuring a batch size and a maximum waiting time, the developer will be unaware that submitted pull requests or pushed changes are now being tested in batches because each submitted change will still have its own test verdict. If the 'batch size' is set to four the approach will be the Batch4 strategy. The highlevel pseudocode for `BatchBuilder` show in Algorithm 1.

Algorithm 1: GitHub App: BatchBuilder

```
When there are 'batch size' changes or the 'wait' time has elapses create a batch
  branch to combine changes
Function TestBatch(batch):
    result = Travis(batch)
    if tests result is passed then
        set status of each changes to "successful" on GitHub;
    else
        if batch length is equal to 1 then
            set status of the change to "failed" on GitHub;
        else
            if batch length is smaller than or equal to 4 then
                foreach change ∈ batch do
                    TestBatch(change);
                end
            else
                TestBatch(first half of batch);
                TestBatch(second half of batch);
            end
        end
    end
```

Merge Conflicts. Our approach does *not* introduce any new merge conflicts. With pre-merge testing, if two or more changes are combined in a testing batch and have a conflict, this conflict will also exist when the changes are added to the master or main branch and would need to be resolved regardless of batch testing. BatchStop4 preserves the testing order of changes, so the conflict can be assigned to the change that occurred later and the changes without conflict can still be tested and integrated with master. With post-merge testing, any conflicts related to integration with master will already have been dealt with before batch testing begins.

Performance. In our implementation, on failure, each batch is implemented as a git branch containing the changes that need to be tested. In TestAll, each commit must be also be merged with master and tested. This same merge operation occurs with the branch. An additional branch operations must occur on failure. However, further optimizations could be performed because each branch has the same ancestor, *i.e.* the lastest commit on master, meaning that we already know the last common ancestor and the branch creations simply involves a simple diff operation. In practice we see that the new branch operation takes less than one second (about 700 milliseconds) and in contrast the testing time is on the order of minutes [7].

5.8 Threats to validity

We selected large open source projects with at least 100 contributors from the Travis torrent dataset [8]. The projects covered a variety of software development contexts, from programming languages to cloud computing. In reproducing, Najafi *et al.* [60] work at Ericsson on OSS projects we increase the generalizability of batching and bisection. Our novel approaches will need to be evaluated in other development contexts. To this end we release our scripts, data, and our `BatchBisect` developer tool [6].

On small projects, there may not always be multiple pull-request available for batching. Clearly, these projects require fewer resources and can either wait until there are enough changes, or run with a smaller batch size. As we show in Section 5.3, the majority of the savings happen with batch sizes of 4, *i.e.* 93%, and even the smallest batch size of two sees substantial savings, *i.e.* 72%. In our tool implementation, we provide a workaround that will test commits that have waited for longer than

the "wait time" specified in the configuration file.

It is only possible to batch builds that required the same test environment, *e.g.,* pull-requests that both requested python 2.7 can be combined while these builds could not be combined with a request for python 3.7. In this work, we use the Travis configuration file to ensure that the combined builds run the same tests in the same environment. On projects that select a subset of tests to be run, combining builds might increase the test scope and future studies of are necessary impact of test scope on batching.

We created build failure risk models using five classifiers: Random Forest, Naive Bayes, MLP, logistic regression, and SGD. Random Forest was the best classifier, so we tuned five hyper parameters for Random Forest leading to a total of 27 configurations for each project. We found an average of one percentage point difference and did not find a consistent configurations across projects, so we reported results with the default parameters. After tuning, the longest training time for the projects was reduced from 4.25 to 3.5 minutes (on a standard laptop).

We assessed the impact of training period by using builds from previous 30 days, two months, or six months of data. We found that the 30 day training period had the same or higher F-scores compared to the longer periods. As other researchers have reported, longer training periods tend to reduce the accuracy of the model by including stale data [42, 93].

Chapter 6

Discussion and Future Work

We contrast the approaches and discuss implications of our findings as well as future work. Table 6.1 shows the important variations for each approach. The first point of variation is the action to be taken on test failure. The original bisection algorithm continues recursively until the individual culprits have been identified [60]. In Section 3.4, we showed mathematically that it is more efficient to stop when the batch is of size four. On failure the Batch4 algorithm tests each commit individually, TestAll, for a constant of 5 executions on failure. The BatchStop4 algorithm uses bisection on failure, but stops bisection when the failing batch contains only four builds using the Batch4 approach. RiskTopN uses a risk model to test the riskiest N builds in isolation and the remaining builds as a batch [60]. If the batch fails, RiskTopN recursively continues with next N riskiest builds. RiskBatch uses a risk model to group builds until a cumulative risk threshold is reached. If the build fails RiskBatch, cannot be repeated because the batch already reaches to threshold. Instead BatchStop4 is used to isolate culprits.

Ranking of approaches. Compared to the standard practice of testing each change in an individually build, all approaches provide substantial improvements reducing the test executions by around half on average. The following is the ranking of approaches by average reduction in savings across projects from worst to best: 44.17% RiskTopN, 46.05% BatchBisect, 47.63% Batch4, 48.50% RiskBatch, and 50.31% BatchStop4.

Stop at 4 The worst two approaches do not use the stop at four condition. From the algorithmic analysis, BatchStop4 has requires the same number of executions as BatchBisect when there is

Table 6.1: Variations in Batching Technique. Stopping at batch size four and using a variable batch size with a risk model are the most promising techniques.

Technique	In Case of Failure	Average Improvement	Stop At 4	Use Bug Model	Dynamic Batch Size	Preserve Order
TestAll	-	-	×	×	×	✓
BatchBisect	Repeat	46.05%	×	×	×	✓
Batch4	TestAll	47.63%	✓	×	×	✓
BatchStop4	Bisect until Batch4	50.31%	✓	×	×	✓
RiskTopN	Repeat	44.17%	×	✓	×	×
RiskBatch	BatchStop4	48.50%	✓	✓	✓	✓

one culprit, but when there are more culprits BatchStop4 requires less, see plot in Figure 5.1. In the empirical evaluation, we see that BatchStop4 is on average 4 percentage points better than BatchBisect. As we later discuss, most savings occurs with small batch sizes resulting in the simple Batch4 algorithm performing only 3 percentage points lower than BatchStop4.

Risk Model. RiskTopN and RiskBatch use a risk model. The model is created using traditional features such as SLOC and number of tests as well as change features, such as the number of changed files or added lines. The accuracy of the model affects the performance of batching approaches, however, we guarantee that all culprits are found and isolated in contrast to, for example, test selection methods that may allow failing tests to ”slip-through” to other QA stages.

Algorithmically, RiskTopN is substantially different from the other algorithms and does *not preserve the test order* of builds providing results for the riskiest builds in isolation first. However, the approach appears to work poorly with the lowest reduction in executions of all techniques. The approach is highly dependent on the risk model and on projects with highly predictive risk the approach may be effective. In contrast, RiskBatch, also uses the risk model but allows for *variable batch sizes* and uses BatchStop4 on failure. This combination appears to allow for appropriate risk and batch sizes providing the second best average savings. For the ruby project, the F-score of 0.53 is the highest among the projects and RiskBatch outperform the other approaches. It is possible that a more accurate risk model may allow RiskBatch to the most effective approach.

Best Batch Size. In all batching approaches, most of the saving is found early with smaller batch sizes 5.1. For BatchStop4 the proportion of saving using different batch sizes is reported, see Figure 5.2. On average across projects, 93% of the saving is achieved with a batch size of 4. The saving achieved by batch size 8 is at least 97% and does not increase with batch size 10. As a result, we reported the savings with a maximum batch size of 8. However on projects graylog2, owncloud, vagrant, and puppet we see that the true best batch size is actually 12, 15, 16, and 16. Table 5.2 shows the projects' best batch size and the additional percentage of savings for those batch sizes, 1, 3, 2, and 2, respectively. While developers from these projects would need to experiment with batch sizes, we feel that it is unlikely that these minor improvements would be beneficial given the additional need for bisection of large batches on failure.

Failure rate vs Savings. Najafi *et al.* [60], concluded that the failure rate controls the batch size. On ruby and rspec that have the highest failure rates, 22.51% and 19.36%, we also see that the best batch size is the lowest at four and the savings are lowest at 32.97% and 29.51%.

However, we see exceptions to the failure rate controlling the savings and batch size. For example, owncloud has the third highest failure rate but the second highest, 53.42%, savings and the best batch size of 15. Examining the owncloud over time we see an uneven distribution of failures with some periods having multiple consecutive build failures followed by consecutive build passes.

Fixed Batch Size In our work, for BatchBisect, BatchStop4, and RiskTopN we have identified a single batch size for the entire period of study. Developers, will need to examine their project history to identify the best batch size. If the failure rate is not constant over time, then projects with an uneven distribution, would clearly benefit from a *variable batch size*. This uneven risk of failure, was the main motivation for introducing RiskBatch that dynamically adjust the batch size based on a risk model. We believe that dynamic batching strategies is the most promising direction for future work.

6.1 Related Work

Continuous integration and delivery (CI/CD) systems are beneficial in both industry and open-source projects because the deployment tasks are automated and developers receive feedback faster,

tests are run automatically, and critical updates are delivered to customers more frequently [53, 81, 36, 69]. However, the goal of CI/CD is to release changes as quickly as possible which increases the already high computational requirements involved in regression testing [35]. Running a subset of tests can reduce the cost of testing. Regression testing research has three streams of research [96]. The first, *minimization*, involves eliminating tests that are redundant or of low value. Early work reduced the problem to one of code coverage, for example, tests become redundant as the system evolves and more than one test covers the same control flow. As a result, much of the work in this area is algorithmic, such as transforming it into a spanning set problem [58], using divide-and-conquer strategies [11], and greedy algorithms [85]. More recent approaches include ant colony optimization in a search space to find the optimum set of test cases [50]. The use genetic algorithm to optimize selected tests and evaluate by total code coverage has also received substantial attention, *e.g.,* [45, 49].

The second, *selection*, uses the same static analysis techniques such as coverage [84] and slicing [41], but selects tests that cover source files that are at higher risk because they have been changed recently [74]. Using specifications such as requirements defined by customer is also used in test selection [13]. A Recent work have focused on using deep learning models to optimize test selection results. [64]

Test case selection is also performed by choosing a subset of test cases, but in contrast to test minimization, test cases that verify risky or recent changes are chosen. Noor *et al.* [65] predict failed test based on similarity to previous failed tests. Wang *et al.* [89] first detect fault-prone source code and then identify related test cases by coverage. Nguyen *et al.* [63] select test cases based on change-sensitivity to external services. Laali *et al.* [51] dynamically identify failed test based on the location of previous failed tests.

The third, *prioritization*, orders tests such that expensive, low-value, or long running tests are run after tests that find faults early. While early prioritization techniques continued to use coverage measures to gauge priority [31], more recent approaches incorporate the faults found in past test runs [46, 24, 61] and change relationships among files [78] to identify high value tests. Zhu *et al.* [100] examine the tests that historically fail together prioritize test runs. Just *et al.* [43] propose an approach based on mutation analysis. Qu *et al.* [70] suggest to prioritize risky configuration in testing.

Wang *et al.* [89] utilizes the quality of source code before finding the relationships between tests and code based on coverage.

Minimization and selection will have slip-throughs because not all tests are run [35]. With prioritization the assumption is that tests can be run in an arbitrary order. However, changing the test order can lead to new flaky failures. Lam *et al.* [52] found that flaky failures due to order dependencies account for 50.5% of flaky failures in the projects they examined. In contrast, our reduction in test executions comes from grouping builds not from eliminate/selecting a subset of tests. As a result, we guarantee no slip-throughs because we run all the tests. We do not introduce order dependency flaky failures, because the entire test suite is run in its original order. Future work is necessary to directly compare the efficacy of batching with selection and prioritization on the same datasets.

6.2 Risk Models

Predicting software defects using statistical models is a research area which has been popular in recent years [30, 77, 62, 18]. Different learning models are used and evaluated to perform bug prediction, such as Support Vector Model (SVM) [47], Logistic Regression [44, 60], KNN [12], and Deep Learning [94, 67]. Bug prediction can be made on varing units, with early studies focusing on file level prediction while recent studies perform change level prediction [47, 44]

Radjenović *et al.*'s [71] survey of bug models categorized the metrics into 1) traditional source code metrics, such as SLOC, 2) object-oriented metrics, such number of children in a class and depth of inheritance [14], and 3) development process metrics such as code change frequency which uses historical data to predict failures. In our work, we use the first and third types of metrics.

Recent works have identified risky changes. Early work focused on regression models [44]. Chen *et al.* [12] use source code metrics such as number of methods, average method complexity, and number of lines of code to build their model. Their learning model is created using k nearest neighbor (KNN). Yang *et al.* [94] study software defect prediction using deep learning at change level. Their prediction has two stages: 1) feature selection which is done by extracting a set of features from a broader initial set of feature using Deep Belief Network. 2) building a logistic

regression classifier using the selected attributes. Pandey *et al.* [67] introduce an approach to detect software defective modules using a deep ensemble learning model. Their approach allocates more testing resources to modules that are more likely bug-prone based on model prediction. In our work, we evaluated five classifiers and found that Random Forests performed the best.

A criticism of statistical bug prediction models is that they do not provide actionable outcomes [44], *e.g.,* what specific action can a developer take if a change is labeled 'risky' because it is in a recently changed file? A further problem is that predictions are often incorrect, which in practice reduces developer confidence [76]. In contrast, our work uses the risk to batch commits and requires no action from developers. If the prediction is inaccurate then additional build test executions are required. However, the saving achieved, even with relatively inaccurate models, is substantial compared to testing each change individually.

6.3 Batching and Bisection

Batching is an effective technique to deal with resource constraints, whether it is computational power, development costs, or time [1, 15]. When changes are batched together and there is a failure, bisection can be used to reduce the number of test execution. When commits are ordered, GitBisection [55] uses a binary search to identify the culprit in $O(log(n))$ time. The approach works well when finding a single regression, but is not designed to find multiple culprits in a batch of changes for integration. To ensure that all tests pass on all changes in a batch, GitBisect would need to run multiple searches, in the worst case n searches, $O(n * log(n))$. In contrast, the bisection approaches discussed in Section 3.5 are designed for integrating multiple commits in $O(log(n))$ time when there is a single culprit and in the worst case $0(n)$ time.

At Google, integration tests can run on the order of hours and can cover thousands of commits, making GitBisection too computationally expensive. Instead, Google developers use the static build dependencies to determine which tests must be run when a file is changed. When a group of changes fails during integration testing, Google developers can immediately eliminate all changes that do not individually relate to the failing test. Since there can be thousands of changes in an integration test, Google also scores the remaining commits on the basis of the number of files in a change (more

files, more likely to be the culprit) and the distance to the root of the build test dependency DAG (closer to the root, safer as more developers have assessed it by now) [101]. In our work, we do not have ordered commits and we do not have the static dependencies. As result, we run the entire test suite on each build. Future work is necessary to determine which of the individual tests can be run independently. Breaking individual tests out of a test suite is often non-trivial and can lead to flaky, unexpected test order dependencies [52], but could increase the effectiveness of batching.

6.4 Pooling Medical Tests

In medical tests, pool testing *i.e.* batching is an effective way to reduce the number of required test kits and thus decreasing costs. Dorfman [20] proposed an approach to detect infected individuals in a large population during World War II. He suggests pooling tests to reduce the cost and the time. If the test is negative, it means all members of that group do not have the disease, otherwise each individual needs to be tested separately, *i.e.* TestAll. Gajpal *et al.* [26] propose an approach to partition people into groups and test each group with one kit. Only, groups with a positive result need to be divided into subgroups and tested further. To improve the pooling process, double and multiple pooling place samples into more than one pool [10, 88]. If a pool tests positive, the samples that are common among other negative tested pools can be removed from further testing. Aragón-Caqueo *et al.* [3] study the effectiveness of batching in COVID-19 tests and report that batching gains more saving when the infection rate is lower. The interest in pool testing has risen dramatically with COVID-19, with these works being submitted in early 2020. Medical pool testing and software batch testing have the same mathematical background and it will be interesting to use the approaches developed in the medical world, *e.g.,* double pool testing, in and SE context and vice versa, *e.g.,* BatchStop4 in medical pools.

6.5 Conclusion and Recommendations

In this work, we introduced a mathematical basis for the batching approaches and make the following research contributions and recommendations for development practices.

Najafi *et al.* [60] showed that BatchBisect was an effective strategy on three projects at Ericsson

and could save 7%, 14%, and 41% of build test executions compared to TestAll. We reproduce this result on the Travis dataset and show that BatchBisect reduces the number of executions by between 22.35% and 57.55% with an average of 46.05%.

We introduce the Batch4 approach in Section 3.4, and we mathematically show that Batch4 requires a constant number of executions on failure, *i.e.* 5, which is the minimum for BatchBisect and saves up to two executions when there are multiple culprit builds in a batch. Batch4 reduces the number of execution required by 29.51% and 55.84% with an average of 47.63%. Batch4 is simpler and does not require bisection while saving an additional 1.57 percentage points on average relative to the total number of builds. We recommend that all continuous integration pipelines batch builds into size four for testing because no additional bisection machinery is required. We also release our tool that integrates with GitHub and Travis CI tool to allow developers to seamlessly batch pull-requests [5].

We introduce BatchStop4 which uses bisection until a batch of four is reached in which case we use Batch4. With this stopping condition, we mathematically that BatchStop4 is equivalent to BatchBisect when there is one culprit, but requires fewer executions when there is more than one culprit build. We see that BatchStop4 saves between 29.51% and 60.83% with an average of 50.31%, with an additional savings of 2.17 to 10.10 percentage points relative to BatchBisect. We recommend that any project already using BatchBisect should modify their algorithm to include a stopping condition for batches of size four. Our tool `BatchBuilder` allows developers to configure the batch size for their project.

We reproduce Najafi *et al.*'s [60] RiskTopN approach where the riskiest n changes in a batch are tested individually and the remaining builds in the batch are tested together. On the Travis projects under study, we find a reduction between 22.04% and 55.77% with an average of 44.04%. However, the simple Batch4 outperforms RiskTopN by 3.59 percentage points on average, and Batch4 does not require a risk model, so we do not recommend that developers adopt this approach.

We introduce the RiskBatch approach which adds builds to a batch until a risk threshold is reached. RiskBatch reduces the number of executions by between 25.93% and 57.43% with an average of 48.50%. The approach is complex requiring both bisection and a risk model and does not perform better than BatchStop4 except in the projects that has the highest F-score in risk model.

Projects that can build a highly accurate risk model may consider using this approach.

Appendix A

Learning Hyper-Parameters

We evaluated five hyper-parameters for Random Forest. Table A.1 shows that the difference between the default and the highly tuned random forests is minimal. In the thesis, we reported the results for the default random forest settings. We discuss the ranges of parameters we evaluated below.

- **Number of estimator**: This parameter shows the number of trees in the forest. As this number grows, the required time to train model will be longer. On the other hand, choosing a small number gives us less diverse range of probabilities which limits the efficiency of our risk-based approaches. We vary the number of estimates from 10, 50, 100, 200 400. The default value of this parameter, is 100 which was the optimum for only one of the projects.

Project	Number of Estimators	Max Depth	criterion	Min Samples Split	Min Samples leaf
ruby	10	50	gini	10	1
metasploit-framework	50	50	entropy	2	1
graylog2-server	100	No limit	gini	2	1
owncloud-android	10	No Limit	gini	2	2
vagrant	10	10	gini	2	1
gradle	400	No Limit	entropy	2	1
puppet	50	50	gini	2	1
opal	10	50	entropy	2	1
rspec-core	10	10	gini	2	1

Table A.1: Optimum hyper parameters value for each project

Project	F score	F score (default)	training time	training time (default)
ruby	0.55	0.54	00:00:50	00:04:20
metasploit-framework	0.21	0.21	00:01:35	00:02:58
graylog2-server	0.45	0.45	00:03:16	00:03:16
owncloud-android	0.28	0.26	00:00:19	00:02:01
vagrant	0.21	0.2	00:00:20	00:01:58
gradle	0.18	0.17	00:05:07	00:01:20
puppet	0.2	0.17	00:01:11	00:02:10
opal	0.22	0.2	00:00:22	00:02:34
rspec-core	0.3	0.29	00:00:32	00:03:44

Table A.2: F score and training time before and after hyper parameter tuning

Six projects had the best results with 10 estimators and the improvements were between one and five percentage points. The optimum value for two projects was 50 and for the last project was 400 which improved the result by two percentage points.

- **Maximum Depth**: This parameter shows the maximum depth of trees. We experimented training with values ranging from 10, 20, 500, 200, and no limit. The default value is None which means no limitation and was the best choice for four projects. The best value for four other projects was 50 and improved the F score between 1 and 4 percentage points. The last two projects had the best results with value of 10 which improved the F score between 3 and 4 percentage points compare to default.

- **Gini vs Entorpy**: This parameter specifies the function to measure the quality of a split using either Gini or entropy. The default value is Gini. Six of the projects had better results with Gini function and for the remaining four projects, entropy results better with between 1 and 3 percentage points.

- **Minimum Samples Split**: This parameter shows the minimum number of samples to split nodes. We evaluated F score by changing this value in range of 2, 5, 10, 20, 50, 100. The default value of two was the best choice for all projects except, ruby and gradle, that have the best value of 10 which improved the F score by 1 and 3 percentage points over default.

- **Min Samples Leaf**: This parameter shows the minimum number of samples required to be

at a leaf node. The default value of one worked best for all of the projects expect owncloud-android that has the best value of two which improved the F score by 2 percentage points over the default.

Appendix B

The `BatchBuilder` Tool

We implemented *`BatchBuilder`* as a free GitHub Application that can be installed on GitHub repositories [5]. Installation and configuration is simple through a ".batch.yml" file. *`BatchBuilder`* seamlessly allows developers to test pull requests and pushed changes in GitHub using Travis CI [5]. There is no change to the GitHub and Travis CI workflow or UI as each individual pull request will have its own build status.

B.1 Configuration

`size`. An integer which determines the length of the batch to start building and testing. It can be any positive number greater than or equal to 1.

`bisection`. Can be **true** or **false**. It specifies if bisection should be performed after build test failure. By setting this to "false" the Dorfman [20] method can be implemented in batching *i.e.* batching and TestAll on failure.

`stopAt`. An integer that specifies when to top bisection and revert to test all. For example by setting this to 4, BatchStop4 is implemented.

`maxWait`. An integer that determines maximum waiting time in seconds. If not enough changes exist in the batch, *`BatchBuilder`* waits no longer than `maxWait` seconds to send the changes to Travis CI.

B.2 Batching and Bisection

After receiving a new change which is a pushed commit or an accepted pull request, `BatchBuilder` adds it to a waiting list. Based on the predefined batch size in configuration file, if enough changes exist in the batch, a new branch named "batch" created by merging changes. This branch is submitted to Travis CI to build and test. After testing, if the verdict is "passed", all changes status on GitHub will be set to "successful". Otherwise, a bisection is performed and the batching process is repeated. Further details on the algorithm were discussed in Chapter 5.7.

Bibliography

[1] B. A. Alexeevich and D. M. Borisovich. Test bundling and batching optimizations, May 16 2019. US Patent App. 16/206,311.

[2] Y. Amannejad, V. Garousi, R. Irving, and Z. Sahaf. A search-based approach for cost-effective software test automation decision support and an industrial case study. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 302–311. IEEE, 2014.

[3] D. Aragón-Caqueo, J. Fernández-Salinas, and D. Laroze. Optimization of group size in pool testing strategy for sars-cov-2: A simple mathematical model. *Journal of Medical Virology*, 2020.

[4] L. Aversano, L. Cerulo, and C. Del Grosso. Learning from bug-introducing changes to prevent fault prone code. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 19–26, 2007.

[5] M. J. Beheshtian and P. C. Rigby. Batchbuilder github app. https://github.com/apps/batchbuilder, 2020.

[6] M. J. Beheshtian and P. C. Rigby. Replication package. https://github.com/CESEL/BatchBuilderResearch, 2020.

[7] M. Beller, G. Gousios, and A. Zaidman. Oops, my tests broke the build: An explorative analysis of travis ci with github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 356–367. IEEE, 2017.

[8] M. Beller, G. Gousios, and A. Zaidman. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 447–450. IEEE, 2017.

[9] J. Bendik, N. Benes, and I. Cerna. Finding regressions in projects under version control systems. *arXiv preprint arXiv:1708.06623*, 2017.

[10] A. Z. Broder and R. Kumar. A note on double pooling tests. *arXiv preprint arXiv:2004.01684*, 2020.

[11] T. Y. Chen and M. F. Lau. Dividing strategies for the optimization of a test suite. *Information Processing Letters*, 60(3):135 – 141, 1996.

[12] X. Chen, Y. Shen, Z. Cui, and X. Ju. Applying feature selection to software defect prediction using multi-objective optimization. In *2017 IEEE 41st annual computer software and applications conference (COMPSAC)*, volume 2, pages 54–59. IEEE, 2017.

[13] Y. Chen, R. L. Probert, and D. P. Sims. Specification-based regression test selection with risk analysis. In *Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*, page 1. IBM Press, 2002.

[14] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[15] C. Cho, B. Chun, and J. Seo. Adaptive batching scheme for real-time data transfers in iot environment. In *Proceedings of the 2017 International Conference on Cloud and Big Data Computing*, pages 55–59, 2017.

[16] L. Crispin and J. Gregory. *Agile testing: A practical guide for testers and agile teams*. Pearson Education, 2009.

[17] R. Cruz. Ricardo cruz.

[18] D. A. Da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, 43(7):641–657, 2016.

[19] T. de Wolff, D. Pflüger, M. Rehme, J. Heuer, and M.-I. Bittner. Evaluation of pool-based testing approaches to enable population-wide screening for covid-19. *arXiv preprint arXiv:2004.11851*, 2020.

[20] R. Dorfman. The detection of defective members of large populations. *The Annals of Mathematical Statistics*, 14(4):436–440, 1943.

[21] S. Dösinger, R. Mordinyi, and S. Biffl. Communicating continuous integration servers for increasing effectiveness of automated testing. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 374–377. IEEE, 2012.

[22] T. Durieux, R. Abreu, M. Monperrus, T. F. Bissyandé, and L. Cruz. An analysis of 35+ million jobs of travis ci. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 291–295. IEEE, 2019.

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

[24] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 235–245, New York, NY, USA, 2014. ACM.

[25] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 191–210. Springer, 2004.

[26] Y. Gajpal, S. Appadoo, V. Shi, and Y. Liao. Optimal multi-stage group partition for efficient coronavirus screening. *Available at SSRN 3591961*, 2020.

[27] P. Gestwicki. The entity system architecture and its application in an undergraduate game development studio. In *Proceedings of the International Conference on the Foundations of Digital Games*, pages 73–80, 2012.

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

[29] M. Gligoric, L. Eloussi, and D. Marinov. Ekstazi: Lightweight test selection. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 713–716. IEEE, 2015.

[30] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2011.

[31] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei. A unified test case prioritization approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):1–31, 2014.

[32] M. Harman and P. O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–23. IEEE, 2018.

[33] A. E. Hassan. Predicting faults using the complexity of code changes. In *2009 IEEE 31st international conference on software engineering*, pages 78–88. IEEE, 2009.

[34] C. Heger, J. Happe, and R. Farahbod. Automated root cause isolation of performance regressions during software development. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 27–38, 2013.

[35] K. Herzig, J. Czerwonka, B. Murphy, and M. Greiler. Selecting tests for execution on a software product, Nov. 3 2016. US Patent App. 14/699,387.

[36] M. Hilton. Understanding and improving continuous integration. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1066–1067, 2016.

[37] M. Hilton, N. Nelson, D. Dig, T. Tunnell, D. Marinov, et al. Continuous integration (ci) needs and wishes for developers of proprietary code. 2016.

[38] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 426–437. IEEE, 2016.

[39] D. Hoffman. Cost benefits analysis of test automation. *STAR West*, 99, 1999.

[40] J. Holck and N. Jørgensen. Continuous integration and quality assurance: A case study of two open source projects. *Australasian Journal of Information Systems*, 11(1), 2003.

[41] D. Jeffrey and N. Gupta. Test case prioritization using relevant slices. In *Proceedings of the 30th Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '06, pages 411–420, Washington, DC, USA, 2006. IEEE Computer Society.

[42] L. Jonsson, M. Borg, D. Broman, K. Sandahl, S. Eldh, and P. Runeson. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. *Empirical Software Engineering*, 21(4):1533–1578, 2016.

[43] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 11–20. IEEE, 2012.

[44] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2012.

[45] A. Kaur and S. Goyal. A genetic algorithm for regression test case prioritization using code coverage. *International journal on computer science and engineering*, 3(5):1839–1847, 2011.

[46] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 119–129, New York, NY, USA, 2002. ACM.

[47] S. Kim, E. J. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[48] S. Kim, T. Zimmermann, K. Pan, E. James Jr, et al. Automatic identification of bug-introducing changes. In *21st IEEE/ACM international conference on automated software engineering (ASE'06)*, pages 81–90. IEEE, 2006.

[49] P. Konsaard and L. Ramingwong. Total coverage based regression test case prioritization using genetic algorithm. In *2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, pages 1–6. IEEE, 2015.

[50] M. Kumar, A. Sharma, and R. Kumar. An empirical evaluation of a three-tier conduit framework for multifaceted test case classification and selection using fuzzy-ant colony optimisation approach. *Software: Practice and Experience*, 45(7):949–971, 2015.

[51] M. Laali, H. Liu, M. Hamilton, M. Spichkova, and H. W. Schmidt. Test case prioritization using online fault detection information. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 78–93. Springer, 2016.

[52] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie. idflakies: A framework for detecting and partially classifying flaky tests. In *2019 12th ieee conference on software testing, validation and verification (icst)*, pages 312–322. IEEE, 2019.

[53] M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö. The highways and country roads to continuous deployment. *Ieee software*, 32(2):64–72, 2015.

[54] J. Li. Successfully merging the work of 1000+ developers [at shopify]. https://engineering.shopify.com/blogs/engineering/successfully-merging-work-1000-developers, 2019.

[55] G. Manpages. git-bisect (1) manual page, 2015.

[56] D. Marijan, A. Gotlieb, and M. Liaaen. A learning algorithm for optimizing continuous integration development and testing practice. *Software: Practice and Experience*, 49(2):192–213, 2019.

[57] D. Marijan and M. Liaaen. Test prioritization with optimally balanced configuration coverage. In *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, pages 100–103. IEEE, 2017.

[58] M. Marré and A. Bertolino. Using spanning sets for coverage testing. *IEEE Trans. Softw. Eng.*, 29(11):974–984, Nov. 2003.

[59] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 233–242. IEEE, 2017.

[60] A. Najafi, P. C. Rigby, and W. Shang. Bisecting commits and modeling commit risk during testing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 279–289, 2019.

[61] A. Najafi, W. Shang, and P. C. Rigby. Improving test effectiveness using test executions history: an industrial experience report. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 213–222. IEEE, 2019.

[62] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan. Heterogeneous defect prediction. *IEEE Transactions on Software Engineering*, 44(9):874–896, 2017.

[63] C. Nguyen, P. Tonella, T. Vos, N. Condori, B. Mendelson, D. Citron, and O. Shehory. Test prioritization based on change sensitivity: an industrial case study, 2014.

[64] W. Niu, X. Zhang, X. Du, L. Zhao, R. Cao, and M. Guizani. A deep learning based static taint analysis approach for iot software vulnerability location. *Measurement*, 152:107139, 2020.

[65] T. B. Noor and H. Hemmati. Studying test case failure prediction for test case prioritization. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 2–11, 2017.

[66] H. Osman, M. Ghafari, and O. Nierstrasz. Hyperparameter optimization to improve bug prediction accuracy. In *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, pages 33–38. IEEE, 2017.

[67] S. K. Pandey, R. B. Mishra, and A. K. Tripathi. Bpdet: An effective software bug prediction model using deep representation and ensemble learning techniques. *Expert Systems with Applications*, 144:113085, 2020.

[68] G. Pinto, F. Castor, R. Bonifacio, and M. Rebouças. Work practices and challenges in continuous integration: A survey with travis ci users. *Software: Practice and Experience*, 48(12):2223–2236, 2018.

[69] A. Poth, M. Werner, and X. Lei. How to deliver faster with ci/cd integrated testing services? In *European Conference on Software Process Improvement*, pages 401–409. Springer, 2018.

[70] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 75–86, 2008.

[71] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič. Software fault prediction metrics: A systematic literature review. *Information and software technology*, 55(8):1397–1418, 2013.

[72] D. M. Rafi, K. R. K. Moses, K. Petersen, and M. V. Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 36–42. IEEE, 2012.

[73] M. T. Rahman, L.-P. Querel, P. C. Rigby, and B. Adams. Feature toggles: Practitioner practices and a case study. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 201–211, New York, NY, USA, 2016. Association for Computing Machinery.

[74] G. Rothermel and M. J. Harrold. A framework for evaluating regression test selection techniques. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 201–210, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[75] R. Saha and M. Gligoric. Selective bisection debugging. In *International Conference on Fundamental Approaches to Software Engineering*, pages 60–77. Springer, 2017.

[76] A. Sarkar, P. C. Rigby, and B. Bartalos. Improving bug triaging with high confidence predictions at ericsson. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 81–91, 2019.

[77] M. Shepperd, D. Bowes, and T. Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6):603–616, 2014.

[78] M. Sherriff, M. Lake, and L. Williams. Prioritization of regression tests using singular value decomposition with empirical change records. In *Proceedings of the The 18th IEEE International Symposium on Software Reliability*, ISSRE '07, pages 81–90, Washington, DC, USA, 2007. IEEE Computer Society.

[79] A. Shi, T. Yung, A. Gyori, and D. Marinov. Comparing and combining test-suite reduction and regression test selection. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 237–247, 2015.

[80] S. Shivaji, E. J. Whitehead Jr, R. Akella, and S. Kim. Reducing features to improve bug prediction. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 600–604. IEEE, 2009.

[81] M. Soni. End to end automation on cloud with build pipeline: the case for devops in insurance industry, continuous integration, continuous testing, and continuous delivery. In *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 85–89. IEEE, 2015.

[82] S. Srivastva and S. Dhir. Debugging approaches on various software processing levels. In *2017 International conference of Electronics, Communication and Aerospace Technology (ICECA)*, volume 2, pages 302–306. IEEE, 2017.

[83] D. Ståhl and J. Bosch. Industry application of continuous integration modeling: a multiple-case study. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 270–279. IEEE, 2016.

[84] A.-B. Taha, S. Thebaut, and S.-S. Liu. An approach to software fault localization and revalidation based on incremental data flow analysis. In *Computer Software and Applications Conference, 1989. COMPSAC 89., Proceedings of the 13th Annual International*, pages 527–534, Sep 1989.

[85] S. Tallam and N. Gupta. A concept analysis inspired greedy algorithm for test suite minimization. *ACM SIGSOFT Software Engineering Notes*, 31(1):35–42, 2005.

[86] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th International Conference on Software Engineering*, pages 321–332, 2016.

[87] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y.-C. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González. Bugswarm: mining and continuously growing a dataset of reproducible failures and fixes. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 339–349. IEEE, 2019.

[88] A. Viehweger, F. Kühnl, C. Brandt, and B. König. Increased pcr screening capacity using a multi-replicate pooling scheme. *medRxiv*, 2020.

[89] S. Wang, J. Nam, and L. Tan. Qtep: quality-aware test case prioritization. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 523–534, 2017.

[90] S. Wang and X. Yao. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability*, 62(2):434–443, 2013.

[91] C. Woskowski. Applying industrial-strength testing techniques to critical care medical equipment. In *International Conference on Computer Safety, Reliability, and Security*, pages 62–73. Springer, 2012.

[92] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, S. Y. Philip, et al. Top 10 algorithms in data mining. *Knowledge and information systems*, 14(1):1–37, 2008.

[93] S.-Q. Xi, Y. Yao, X.-S. Xiao, F. Xu, and J. Lv. Bug triaging based on tossing sequence modeling. *Journal of Computer Science and Technology*, 34(5):942–956, 2019.

[94] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26. IEEE, 2015.

[95] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, 22(2):67–120, 2012.

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

[97] L. Yu, E. Alégroth, P. Chatzipetrou, and T. Gorschek. Utilising ci environment for efficient and effective testing of nfrs. *Information and Software Technology*, 117:106199, 2020.

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

[99] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu. The impact of continuous integration on other software development practices: a large-scale empirical study. In *2017*

32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 60–71. IEEE, 2017.

[100] Y. Zhu, E. Shihab, and P. C. Rigby. Test re-prioritization in continuous testing environments. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 69–79, 2018.

[101] C. Ziftci and J. Reardon. Who broke the build? automatically identifying changes that induce test failures in continuous integration at google scale. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 113–122. IEEE, 2017.