On Test Selection, Prioritization, Bisection, and Guiding Bisection with Risk Models

Armin Najafi

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of (Master of Applied Science (Software Engineering)) at
Concordia University
Montréal, Québec, Canada

Dec 2018

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Armin Najafi**

Entitled: **On Test Selection, Prioritization, Bisection, and Guiding Bisection with Risk Models**

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

(Master of Applied Science (Software Engineering))

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

Signed by the final examining commitee:

______________________________ Chair
Dr. Lata Narayanan

______________________________ Examiner
Dr. Emad Shihab

______________________________ Examiner
Dr. Jinqiu Yang

______________________________ Supervisor
Dr. Peter C. Rigby

______________________________ Co-supervisor
Dr. Weiyi Shang

Approved by ______________________________
Dr Volker Haarslev, Graduate Program Director

January 2019 ______________________________
Dr Amir Asif, Dean
Faculty of Engineering and Computer Science

Abstract

On Test Selection, Prioritization, Bisection, and Guiding Bisection with Risk Models

Armin Najafi

The cost of software testing has become a burden for software companies in the era of rapid release and continuous integration. In the first part of our work, we evaluate the results of adopting multiple test selection and prioritization approaches for improving test effectiveness in of the test stages of our industrial partner Ericsson Inc.

In order to assist Ericsson with improving the test effectiveness of one of its large subsystems, we adopt test selection and prioritization approaches based on test execution history from prior research. By adopting and simulating those approaches on six months of testing data from our subject system, we confirm the existence of valuable information in the test execution history. In particular, the association between test failures provide the most value to the test selection and prioritization processes. More importantly, during this exercise, we encountered various challenges that are unseen or undiscussed in prior research. We document the challenges, our solutions and the lessons learned as an experience report. Our experiences can be valuable for other software testing practitioners and researchers who would like to adopt existing test effectiveness improvement approaches into their work environment.

In the second part of our work, we explore batch testing in test execution environments and how it can help to reduce the test execution costs. Software testing is one the costliest stages of software development life cycle. One approach to reducing the test execution costs is to group changes into batches and test them at once. In this work, we study the impact of batch testing in reducing the number of test executions to deliver changes and find culprit commits. Based on the failure rate we run simulations to determine the optimal batch size for three projects at Ericsson. Flaky test failures are tests that pass and fail on the same change. We factor test flakiness into our simulations as they increase the number of executions to test changes. The larger the flake rate the smaller the batch size. Although batch testing can help to reduce the test executions, unlike testing each change independently, when there is a failure a bisection must be done to find the likely true failing culprit commit. We introduce a novel technique where we guide bisection based on two risk models: a bug model and a test execution history model. We isolate the risky commits by testing them individually, while the less risky commits are tested in a single large batch. Our results show that batch testing in ideal environments with low test failure rates can reduce the test executions up to 72%. We also show that test flakiness will limit the savings to 42% as larger batches increase the probability of flakiness and hence extra executions. Moreover, we show that risk calculation approaches can be used to effectively predict the culprit commits in a failing batch. Furthermore, we show that culprit predictions can be used with our TestTopN approach to help to reduce the test executions up to 9% compared to our FifoBisection baseline. The results we present in this thesis have convinced Ericsson developers to implement our culprit risk predictions in the CulPred tool that will make their continuous integration pipeline more efficient.

This thesis is organized as a "manuscript" thesis with a background literature chapter followed by subsequent chapters that are accepted or submitted papers.

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 supervisors, Dr. Peter Rigby, and Dr. Weiyi Shang. This work would not have been possible without their guidance, support and encouragement.

In addition, I would also like to thank Ericsson Inc. for providing us with their full support, knowledge, and necessary hardware. My special thanks go to Chris Griffith, Gary McKenna, Wasiq Waqar, Vishal Pravin, Ladan Maxamud, Fred May, Danny Lee, and Jerome Lambourne for their valuable feedback and support.

Additionally, I would like to thank Concordia University for providing me with this opportunity to be part of this exciting journey. Last but not least, I would like to thank my parents and family for their love and constant support. I could not have accomplished this without their support and motivation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is organized as a “manuscript” thesis with a background literature chapter followed by subsequent chapters that are accepted or submitted papers.

Testing is an important, yet time consuming and costly process, especially for large software systems. Prior research estimates that testing consumes between 30% to 50% of the time in software development life-cycle [PZTM13]. For example, an industrial case study shows that it takes over two days to complete testing on a medium size video conferencing system [MGS13].

The cost of testing has become a burden for software companies during rapid release [Bec00], where continuous integration techniques are widely adopted in practice to receive feedback from testing as soon as changes are made to the source code [DMG07]. With thousands of commits made to the source code every day, it is challenging to keep up with the speed of development. Google’s version control repository receives over 16,000 commits every day [PL16], which results in a median of 27 test requests per minute [ZSR18]. Our industrial collaborator Ericsson has faced the same challenges for testing its large-scale software systems. Moreover, the changes in their testing processes may be even greater due to the complex testing infrastructures that are purposely designed for each software subsystem.

Test selection and test prioritization are proposed by prior research to improve the effectiveness of test executions [ASD14, HGCM15, ERP14, KP02, ZSR18]. With test selection, tests are either executed or skipped on the fly. With test prioritization, tests are reordered such that more test failures can be captured earlier using limited testing resources. We leverage prior research to assist Ericsson with improving the testing process of one its large-scale software systems. Since prior research is typically evaluated on a particular industrial subject system (e.g., from Microsoft or Google), it is unclear to what extent the existing approaches can improve the test effectiveness in our subject system and whether there are challenges that are unseen by prior research.

In the first part of this thesis, we share our industrial experience for adopting test selection and prioritization approaches to improve the test effectiveness of one of the large software systems at Ericsson. In particular, the adopted approaches are based on historical test failure frequency, test failure association, and the costs associated with the testing process. In order to evaluate the usefulness of these adopted approaches, we simulate applying these approaches using six months of testing data from a large-scale system at Ericsson. Our results show the importance of test execution history in enabling the test selection and prioritization approaches to help with software testing in practice. We encountered many engineering and design challenges for adopting and applying the approaches to the testing processes of Ericsson. In the end, we conquered the problems and documented the challenges, our solutions and the lessons learned as an experience report. We believe that our experience in adopting existing test selection and prioritization approaches can help software practitioners and researchers who want to adopt software testing approaches into their work environment.

The major contributions of this part are:

- We adopt and evaluate test selection and prioritization approaches with the goal of improving test effectiveness in a large industrial system with a complex testing infrastructure.
- We demonstrate the value of test execution history for improving test effectiveness in a large-scale industrial system in practice.
- We provide an industrial experience report that documents the challenges that are encountered and our lessons learned during the adoption process of the test selection and prioritization approaches.

In the second part of this thesis, we study batch testing and its implications for reducing test executions in large-scale software development environments. To isolate test failures, many companies have adopted the DevOps strategy of testing each individual commit. While effective at isolation there are substantial computation requirements. To limit the resource requirements, many software companies, including Ericsson, have adopted batching to reduce the cost of testing. Batch testing groups commits and allows all of them to be tested at once. When the batch passes, all of the commits can proceed in the continuous integration pipeline at once and save the resources.

Although batch testing can reduce test executions, it introduces a new problem. When a batch fails, the culprit commits causing the batch failure need to be identified. One of the common approaches used for finding a culprit in a group of failing commits is bisection. When commits are ordered, GitBisection [GBS] can use an ordered binary search to identify the culprit in log(n) time. However, GitBisection cannot be used in our studied subsystem in Ericsson as commits are not merged yet and have no order. In this regard, Ericsson has adopted an approach named bisection were batches are split in half and tested until the culprit is found.

In the first part of this work, we study the impact of batch testing on reducing the test executions in environments with various test failure rates. In practice, test failure rates in test environments tend to be very low. For example, on Chrome only 12.5% of tests fail [ZSR18]. Batch testing offers the highest savings in test environments with low failure rates.

In the second part, we examine flaky tests, which can pass and fail on the same commit. Google reports that 1 in 7 tests are flaky and that 84% newly failing tests are actually flaky failures [Mic16]. Flaky tests are exacerbated by batching, as the batch size grows the probability that one or more commits will have a flaky test failure also grows.

In the last part of this work, we propose more efficient approaches for finding the culprits when a batch failure happens. First, we propose risk-based approaches for calculating risk values for the commits of a batch. Then we propose a TestTopN approach for testing the riskier commits individually and the rest of the lest risky commits together in a separate batch. We study how this approach can reduce the test executions compared to the bisection approach used in Ericsson. We propose two risk calculation approaches. The first approach is based on well-studied bug model literature [KSA$^+$13]. Our second approach is based on using test execution history and the file changes of the commits.

More specifically, we answer the following three research questions.

RQ1: What is the most cost-effective *BatchSize* for the number of culprits discovered during testing?

Batching commits for testing is more efficient with a low test failure rate, i.e. *CulpritRate*. The smaller the number of the test fails, the larger the *BatchSize* as there will be few failures. In contrast, the higher the *CulpritRate* the larger the number of bisections resulting in more executions.

In this work, we have studied the impact of batch size on the test executions and what an most cost-effective *BatchSize* should be. We have seen that in an ideal environment for projects that have lower *CulpritRate*, a higher batch size will lead to fewer executions.

The higher the *CulpritRate* the smaller the most cost-effective *BatchSize*. For example, Project A has a *CulpritRate* of over two times Project C and with a *BatchSize* of 4, the savings are 46%, while Project C can have a *BatchSize* up to 9 with savings of 72%.

RQ2: What is the most cost-effective *BatchSize* when some bisections are done as a result of flaky failures?

Test flakiness is an inevitable part of any test environment. Google reports that 1 in 7 tests are flaky and that 84% newly failing tests are actually flaky failures [Mic16]. A flaky test failure is defined as a test that passes and fails on the same commit. We study the impact of test flakiness in finding the most cost-effective *BatchSize*. Our study shows higher *FlakeRate* will limit the most cost-effective *BatchSize*. When commits are tested individually, a flaky failure does not affect other commits and the number of executions remains constant. In contrast, the larger the *BatchSize* the higher the probability that at least one of the commits in the batch will be flaky. Any flaky batch failure incurs the penalty of an unnecessary bisection.

The higher the *FlakeRate* the smaller the *BatchSize* and the smaller the savings in executions. For example, Project B has a 1/3 higher *FlakeRate* than Project C and a *BatchSize* of 4 saves 14% of the executions compared to 41%, respectively. With flaky failures, Project C's most cost-effective *BatchSize* and savings are reduced from a *BatchSize* of 9 and execution savings of 72% to 4 and 41%, respectively.

RQ3: Can risk models predict the culprit commit and reduce the number of executions to find the culprits on failing batches?

Batch testing is effective in reducing the test executions, however, introduces a new problem. When a batch fails, the root cause of the failure, i.e. culprit, needs to be found among the failed commits. We use commit risk models to predict the culprit commit when a batch fails. We use two types of models, Bug models, and historical test information.

Bug models have been effective at identifying the commits that are most likely to lead to future bugs, i.e. bug introducing changes. [HBB$^+$12, RHTZ13, TMH$^+$15, Aki71, DLR10, Has09, ZPZ07, MW00, MPS08, NZZ$^+$10, RPH$^+$11, KZJZ07, GFS05, LHSR06, MK92, CMRH09, MKAH14, Moc10, SJI$^+$10, SMK$^+$11, KSA$^+$13]. We use these techniques to identify which of the commits is the most likely culprit. We then test the riskiest commits individually and batch the remaining commits.

Our second approach is based on using test execution history. Test executions history has been largely studied for performing test selection and prioritization [KP02, ERP14, ASD14, ZSR18, CMBA17]. In contrast, we use test execution history to predict a culprit commit given a batch test failure. Particularly we use the relationship between file changes and test failures extracted from the test execution history. Campbell et al. [CMBA17] have proposed an approach for suggesting tests to run based on file changes. We reverse this idea by determining the most likely culprit given the failing test and the files under change. We found that our risk-based approaches can significantly reduce the number of test executions by predicting the culprit commits.

Both culprit risk prediction models are effective, but *TestExecutionHistory* outperforms *BugModel*. *TestExecutionHistory* is able to predict the culprits using the Top2 predictions are sufficient and correct 63% and 66% of the time for Projects B and C with *BatchSizes* = 4. Compared to *FifoBisection* this results in -9.0% and -7.6% fewer executions, respectively.

The results we present here have convinced Ericsson developers to implement our culprit risk predictions in the CulPred tool that will make their continuous integration pipeline more efficient.

The thesis is structured as a manuscript thesis with a background literature review and two complete papers. The complete introduction for each paper and research questions is contained in the paper chapter in which they are fully presented. Chapter 2, gives an extensive literature review for our studied topics. Chapter 3 is dedicated to our experience report regarding adopting different test selection and prioritization approaches in the test environment of Ericsson. Chapter 4 is dedicated to our experiments regarding batch testing and isolating the culprits. Finally, we conclude the thesis in Chapter 5.

Chapter 2

Background and Literature

Software testing over the time has received a lot of attention from research communities and there have been numerous proposals ranging from how the testing should take place, how to revise the tests and improve the test effectiveness to ways for finding the root causes of the test failures and automatic classification of them. In this section, we go over some of the background and literature related to software testing and improving test effectiveness in general and then more specifically doing so using test executions history. Later on, we explore the literature related to fault localization and more specifically locating root causes in batch testing.

2.1 Test Selection

Improving regression testing is a very well studied subject in software engineering literature and has attracted a lot of attention from both industry and academic communities. The proposed techniques for tests selection and prioritization range from optimization algorithms for maximizing some objective function, maximizing code coverage of the test executions to minimizing the execution time. Kazmi *et al.* [KJMG17] and Suleiman *et al.* [SAH17] propose extensive surveys on the recent studies related to tests selection and prioritization.

Laali *et al.* [LLH+16] talks about three ways that can make regression testing more effective. test suite minimization, test case selection, and test case prioritization. For the first two techniques, a subset of test cases is selected from a tests pool to be executed. On the other hand, in the third approach, none of the tests are excluded but just the order of running them are changed. This work proposes an approach for online prioritization of the test cases based on tests coverage data. This work is an improvement to the work done by Zhou *et al.* [Zho10] which is instead based on off-line information. These works are closely related to our approach, however, as opposed to us they rely on tests coverage data such as statement or branch coverage information. Our approach instead relies on empirically inferred information about the previous test results and how effective each test has been in its history.

Koochakzadeh *et al.* [KG10] propose a human-assisted approach for removing the tests redundancy. They build upon their previous work [KGM09] that only relies on test coverage data and show that utilizing test coverage data alone can be misleading for eliminating the tests redundancy and result in weaker test cases. They show that their new approach facilitates the process of test redundancy analysis for testers and that they get better results than their previous fully automated method. Our work also partly addresses the matter of tests redundancy and their importance. However, we take a different approach to eliminating it. We do not rely on tests coverage data and also do not require any human intervention. Instead, we use association rules mining techniques for mitigating the problem.

Labuschagne *et al.* [LIH17] propose a study for cost measurement of regression testing in practice. They study 61 Java projects running on Travis CI and find that 18% of test suite executions fail and that 13% of these failures are flaky. Among the non-flaky failures, only 74% were true positives and the remaining 26% were false positives. Their study emphasizes the importance of the works like ours for improving test effectiveness of the continuous integration flow in large software projects.

Noor *et al.* [NH17] propose a comparison of using different test measures in a logistic regression model for ranking the effectiveness of test cases. They use this ranking for prioritizing test case executions and compare the results. The measures that they use include historical fault detection rate, method coverage, changed method coverage, size of tests and a similarity-based quality metric that they define. They conclude that no individual metric will outperform others in all of their projects, however, a combined set of metrics will give superior prioritization results. This work can complement our study by using logistic regression as a method for calculating ranking values of the test cases. However, our approach does not depend on test case coverage in any way.

Shi *et al.* [SYGM15] compare and combine a test-suite reduction and a regression test selection technique. Their test reduction technique permanently removes tests from the flow. On the other hand, their test selection technique suggests running only tests that their outcome will be changed by the new changes introduced in the new revision of the software. Their approach, as opposed to ours, relies on code coverage information like statement or branch coverage. They also do not talk about the concept of tests prioritization as we describe it.

Saha *et al.* [SZKP15] propose a new approach for solving the test prioritization problem by reducing it to a standard information retrieval problem. They assume that test cases and source code usually embody meaningful identifiers and comments which can be treated as natural language. Therefore, information retrieval techniques can be utilized on them to give priority to running tests. They mainly focus on addressing the limitations of mostly coverage-based test prioritization techniques such as profiling overhead and also the change of coverage information over time when significant changes are applied to the main code base. This work does not directly relate to our study as we do not utilize test coverage information to our techniques. However, can be a complementary approach for improving our test selection results.

Wang *et al.* [WNT17] propose a tool named Quality-Aware Test Case Prioritization which aims to improve the limitation of current coverage-based test case prioritization algorithms. They mention typical related coverage-based techniques assign higher priorities to test cases that have higher dynamic or static code coverage. However, they do not take into account the fault-proneness of different code segments. They aim to solve this limitation by benefiting from two widely studied approaches in code inspection research, namely static bug finders and defect prediction models. Their approach does not directly relate to our study as we do not use test code coverage in our approaches. Instead, we identify more effective tests based on a few metrics extracted from tests execution history and prioritize them for next runs.

Nardo *et al.* [NABL] propose a study in which they evaluate 7 coverage-based test improvement techniques on a common carefully designed industrial system to give a good measure for comparing the performance of the different approaches. They evaluate four prioritization techniques, a test selection technique, a test suite minimization technique and a hybrid approach that combines selection and minimization. They also examine the effect of choosing different coverage criteria on the results. Among their findings is that they conclude test suite minimization using finer grained coverage criteria could provide 79.5% savings in execution costs while maintaining a fault detection capability level above 70%. Their approach, as opposed to ours, relies on code coverage information. However, their findings confirm ours. Moreover, they correctly point to the trade-off between tests reductions and tests fault detection capability, something that we have encountered in our approaches as well.

Shi *et al.* [SGG$^+$14] propose a new measure for evaluating test reduction techniques. They mention 3 limitations in the traditional evaluation of test reduction techniques and claim their new evaluation criteria will address those. Their metrics mostly address coverage-based test reduction techniques which are irrelevant to our study.

Kumar *et al.* [KSK] consider the concept of tests improvements as an optimization problem that they try to solve using a fuzzy-ant colony optimization technique. They define a fitness function based on code coverage, client requirements coverage, fault coverage, mutant-killing score in minimum effort, and cost. By optimizing the fitness function they try to find a subset of test cases that maximize the achievement of all of the test objectives. Their approach is irrelevant to our study, however, they also emphasize the importance and possibility of improving the test executions in software development processes.

Gligoric *et al.* [GEM15] propose a tool named Ekstazi based on dynamic analysis of file dependencies. In their approach for each test entity, they create a collection of files that are accessed in test executions. Then they suggest tests based on whether the dependent files have been changed or not. Their approach relies on the dependency analysis of the test suites and tracking what files have been accessed during test execution. Our approach on the other hand, mostly relies on empirical analysis of the test executions history and how effective each test has been in the past. This makes it ideal for situations where performing dependency analysis on test suites is impractical.

Panichella *et al.* [POPDL15] propose an approach for test case selection using genetic algorithm. They attempt to improve the current state of the art by diversifying the solutions (sub-sets of the test suites) generated during the search process. They mention how other approaches such as greedy algorithms or multi-objective genetic algorithms have been experimented in previous works, but they do not demonstrate a clear winner. Besides, their combination does not necessarily produce better results as well. They attempt to address this by introducing a new approach named Diversity based Genetic Algorithm (DIV-GA). This work does not directly relate to our study.

Xu *et al.* [XGKS14] propose a test selection mechanism based on fuzzy expert systems. Their approach correlates the information represented by customer profile, analysis of prior test case results, system failure rate, and changes in system architecture. Using the aforementioned information they suggest a list of test cases that are potentially critical for a regression testing scenario which then can be used in the decision-making process of regression test case selection. For building their fuzzy expert system, they have consulted human experts to construct a knowledge base that can be later fed to a fuzzy inference engine. This work aims to solve the same issue as we do, however, their approach to addressing the problem and the information they use is different from our study.

Souza *et al.* [dSPdAB14b] propose a comparison study for binary multi-objective particle swarm optimization approaches used for test case selection. They consider the test selection process as an optimization problem that tries to find a subset of test cases which optimizes one or more objective functions. They account for maximizing requirements coverage and minimizing the cost of test execution efforts in their objective function. For finding the optimal value they propose a new approach based on particle swarm optimization. In a similar work, [dSPdAB14a] authors consider maximizing branch coverage while minimizing execution time as their objectives. Apart from the common goal of test selection, these studies do not directly relate to our approaches.

Fourneret *et al.* [FCB$^+$14] propose a test classification and selection tool based on UML/OCL models to validate a new version of the system. Their approach is based on dependence analysis of behaviors from state charts and class diagrams. This work does not directly relate to our study.

Kumar *et al.* [KC15] propose a new test case prioritization technique based on the coupling information between different modules of a program. The idea is to find the modules that are most affected and run their relevant test cases so that more bugs can be found earlier. They use call graphs as an architectural design for coupling information between modules and module dependency matrices as a quantitative measure for the dependence of the system modules. This work can compliment our study by adding the coupling information as a new dimension to the analysis. However, we tend to stay away from call graphs and tests coverage data as we claim in certain scenarios it is difficult to obtain accurate enough information about them for doing such analysis.

Konsaard *et al.* [KR15] propose a technique for test case prioritization based on genetic algorithm. Their prioritization technique attempts to achieve full code coverage in a timely manner. This study relies on code coverage and is irrelevant to our work.

Klindee *et al.* [KP15] propose a test case prioritization approach based on information retrieval and text similarity techniques for an introduced change request. They store test case documents describing the test cases. Then later on by querying on a change request, they retrieve the list of the most relevant test cases. They also rank those test cases using the analytic hierarchy process. This study is irrelevant to our work as we do not keep track of test case description documents and also change requests. Instead, we propose a generic approach to trimming the tests pool in a test environment.

Singh *et al.* [SS14] propose a multi-criteria test prioritization technique for improving regression testing. They utilize code coverage, branch coverage, re-usability coverage, path coverage and fault coverage for their prioritization technique and compare their results with other available methods. This study is irrelevant to our work as we do not utilize coverage information in our analysis.

Tyagi *et al.* [TM14] propose multi-objective particle swarm optimization technique for test case prioritization. Their approach relies on three steps, removing redundant test cases using simple matrix operation, selecting test cases which cover all the faults in minimum execution time using particle swarm optimization techniques and then prioritizing test cases based on the ratio of fault coverage to the execution time. The ranking methodology of this study is similar to our approach for test case prioritization. However, we use the number of false positives, number of true positives and average execution time for suggesting an order for the test executions.

Magalhães *et al.* [MaBMM16] propose a test case selection and prioritization approach based on information retrieval techniques. They address the problem of finding the relevant test cases given a list of test cases and change requests described in natural language. This approach does not directly relate to our study.

2.2 Test Selection Using Execution History

Kim *et al.* [KP02] propose a history based test prioritization technique. They mention the problems with regression testing in resource-constrained environments and how a prioritization technique can help the environment. Their prioritization technique gives probabilities for each test case to be executed during the next test session run. They propose 3 approaches for calculating the probability values. The first approach is based on execution history, the test cases that have not been executed recently, get a higher probability. The second approach is based on fault detection effectiveness. Tests that have revealed more faults recently will get a higher chance to be executed again. And the last approach is based on the coverage of program entities like statements, paths, functions, etc. For example, using functions as the program entities of interest, a higher probability is given to test cases that cover functions which are infrequently covered in the past testing sessions. This work aligns very well with parts of our approaches for test prioritization discussed in this work, although with some differences. Our approach relies more on the attributes of the test execution history while calculating the probability values. These are attributes such as the number of false positives and test cases average duration time. Besides, we do not use tests coverage information as we claim it is impractical to obtain that type of information in a clean and reliable manner from complicated and interleaved software development environments such as our industry partner which has a huge volume of low-level hardware-related programming procedures.

Anderson *et al.* [ASD14] study reducing regression tests based on tests execution history. There are two main techniques utilized in their study. Frequency and association analysis of the test executions. In the frequency analysis approach or also referred to as most common failures, test cases that previously failed the most are recommended as test cases that are likely to fail in the future. Their second approach is described as failure by association where failures in certain subsets of tests are used to determine other subsets that are likely to fail. This work is closely related to our techniques. Their frequency approach confirms our results obtained from our environment. However, their association analysis technique differs from ours in a few ways. First of all, they rely on smoke tests for generating their association rules. They run some smoke tests in prior to the next regression test run. Using the results of the smoke tests they predict which regression tests will fail consequently, and use them in their test selection process. Our approach, however, is based on the fact that redundant tests should not be executed. If one test can find the failure the execution of another correlated test should be eliminated. They conclude that their association analysis did not improve the results significantly. On the other hand, our approach for removing the redundant test executions using association rules mining significantly improves the results.

Herzig *et al.* [HGCM15] propose a tool named THEO which stands for Test Effectiveness Optimization using Historic data. THEO is a generic test selection tool that is supposed to accelerate test processes without sacrificing product quality. It is based on a cost model which dynamically skips tests when the expected cost of running one exceeds the expected cost of removing it. This work is closely related to ours. Our discussed cost analysis method is a replication of Herzig's approach which we compare to our other proposed methods in our study. We also propose an attempt to improve this approach by utilizing the tests association information in the cost analysis formulas, however, it does not prove to outperform the Herzig's method. From the other hand, our prosed frequency and association analysis approach outperforms Herzig's costs analysis method significantly.

Anderson *et al.* [ASD15] propose a classification based approach for predicting the test failures. They show that using attributes like historical test case pass and fail information, organizational information, code complexity information, and code change information, test failures can be predicted. This data can be used to help improve the development of certain code areas or maximize the benefits of scarce testing resources. The classification based approach proposed in this work can complement our research and open new research directions for our future studies. However, we mostly focus on test selection and ways to reduce test executions efforts most efficiently while maintaining the software quality.

Elbahum *et al.* [ERP14] propose a time-window based test selection and test prioritization technique for improving regression testing in continuous integration development environments. They use their test selection technique in a pre-submit phase in their studied testing process so that they can run a subset of the required tests instead of all. Moreover, they use their test prioritization technique in the post-submit phase to find the faults faster. This work similarly to ours do not rely on test coverage data and likewise demonstrates the importance and effectiveness of test selection and prioritization techniques for improving regression test executions.

Yuecai et al. [ZSR18] propose three test re-prioritization approaches that use co-failure distributions of tests to dynamically re-prioritize test executions. CoDynaQSingle is their first re-prioritization technique which changes the order of running test dynamically based on the recent test failures. In order to address the starvation problem for this approach, they proposed two other re-prioritization techniques, named CoDynaQDouble and CoDynaQFlexi. Our study adopts CoDynaQSingle approach as one of our test prioritization techniques.

2.3 Fault Localization

Due to the increasing scale of software developments, and consequently increasing complexity of software debugging, fault localization techniques have become a popular field of research during the past few years. Software fault localization while being crucial is widely recognized to be one of the most expensive, and time-consuming activities in the process of software development [WGL$^+$16, Ves85].

There have been hundreds of different attempts to locate the riskier and error prone parts of a code base and guide the testers and developers toward the root causes of the test failures. [WGL$^+$16] provides an extensive review of the fault localization literature. Fault localization techniques mostly attempt to find the buggy statements in a code base. Our approach, on the other hand, attempts to find the buggy commit among a group of batch tested commits. Technically, most of the fault localization techniques available in the literature can complement our debugging approach by enhancing the process of finding the buggy statements after the culprit change has been located. Apart from this, they do not directly relate to our methodology.

Zhang *et al.* [ZZ14] propose a bug localization approach using Markov logic. The problem is to find the buggy statements of the code base and they do so by combining different information sources like statement coverage, static program structure information, and prior bug knowledge. This approach does not directly relate to our study.

Cellier *et al.* [CDFR11, CDFR08] propose an interactive tool for multiple faults localization using formal concept analysis and association rules mining. They use execution traces like executed lines or variable values to suggest occasions when appearing certain events in a trace will most likely lead to an execution failure. This study does not directly relate to our work as we do not use code statements coverage.

Nessa *et al.* [NAW$^+$08] propose a fault localization technique based on N-gram analysis. They use the exact execution sequence of the failed test cases to suggest failing patterns. N-grams are subsequences of length N. They choose N-grams that appear more than a certain number of times in failing traces and also calculate the conditional probability of a test case being failed given that the N-gram appears in that test case's trace. This approach does not directly relate to our study.

Ji *et al.* [JXxC$^+$04] propose a model-based statistical approach for locating faults at input level. They extract failure patterns from a given set of failure samples and then use it to analyze the statistical correlation between those patterns and the observed failures to locate the faults. This approach does not directly relate to our study.

Baah *et al.* [BPH10a] propose an approach in which they combine the concepts of program dependence graph and dependency networks to construct a probabilistic graphical model of program behavior. Program dependence graph is like a flow graph of the program statements and the dependency network is a type of probabilistic graphical model. Their approach facilitates probabilistic reasoning about program behaviors which can be used in fault localization and fault comprehension. This approach does not directly relate to our study.

Aberu *et al.* [AVG09] utilizes the problem of minimal hitting sets (MHS) in the context of model-based diagnosis as a way of locating faults. MHS is the problem of finding an approximate collection of minimal hitting sets of a collection of sets. In this context, the minimal hitting sets are the solutions for the diagnostic problem. These solutions are basically programs statements that are spotted to be faulty. This study does not directly relate to our work as we locate faults in commits level as opposed to program statements level.

Wong *et al.* [WQ09] propose a fault localization approach based on neural networks. They train a back propagation neural network using statement coverage data and the test execution results (success or failure). Using the trained network they compute the suspiciousness of each executable statement to be faulty. This study does not directly relate to our work as we do not intend to predict faulty code statements and we do not use test coverage information.

Ascari *et al.* [AAPV09] extend [WQ09] by applying the neural network technique to object-oriented programs. They also investigate the use of support vector machines as a better and more efficient training algorithm. They report similar results based on class methods covered by different test cases and ranking them by the possibility of being faulty. This study does not directly relate to our work.

Wong *et al.* [WDG^{+}12] propose an improved version of their previous work [WQ09] by using RBF instead of back propagation neural network which are believed to have several advantages including a faster learning rate, and resistance to some issues like paralysis and local minima. They train a neural network to learn the relationship between the statement coverage information of a test case and its corresponding execution results. Then use virtual test cases that cover only one code statement as the input to the trained neural network to get the suspiciousness of each corresponding statement in terms of its likelihood of being faulty. This study does not directly relate to our work.

Briand *et al.* [BLL07] attempt to improve the approach presented by Jones *et al.* [JHS02]. They use code coverage information and also test specifications to create a C4.5 decision tree and report probability rankings for faulty code statements. They attempt to address the Tarantula's [JHS01, JHS02] difficulty to deal with the presence of multiple faults. They categorize test failures such that failed test cases in the same partition most likely fail due to the same root cause. This approach does not directly relate to our study.

Jeffrey *et al.* [JGG08] propose a fault localization approach using value replacement. For a given failed test case, they search for statements that change the test results by trying to apply different values to the statements. When they find such cases they call it Interesting Value Mapping Pair (IVMP) and they show that IVMPs often occur at faulty statements or statements that are directly linked to faulty statements via a dependency edge. This study does not directly relate to our work.

Zhang *et al.* [ZGG06] propose a fault localization approach based on predicate switching. They mention that finding buggy statements by changing the states of the programs is an expensive process and involves a massive search space especially in cases where there exist float or integer variables. Instead, they propose to utilize branch predicate switching. They force the execution along the different paths of branches like an if-else branch and see if they can get the correct test result. As branch predicates are either true or false, the search space will be reduced significantly. This work does not directly relate to our study.

Abramson *et al.* [AFMS95] propose an automatic debugging approach named relative debugging based on the process of comparing a modified code against a correct reference code. The idea behind the tool is to find program bugs by runtime comparison of the internal state of a new program and a reference version of it. This approach does not directly relate to our study.

Zeller *et al.* [Zel02] propose a fault localization approach named delta debugging. They analyze the difference in the program states of a failing run and a passing run related to a bug in question and narrow the search down to a small set of variables. They search for suspicious variables in the failed run using their values from the corresponding point in the passing run and repeating the program execution. This approach does not directly relate to our study.

Pearson *et al.* [PCJ$^+$17] perform an experiment by replicating 7 previously proposed fault localization techniques on real faults. They point to the fact that these approaches have been evaluated using artificial faults, hence they use real faults in their study. While replicating the methods using artificial faults, they can verify the results of 70% of the studies. On the other hand, they show that using real faults in their analysis leads to the results of 40% of the techniques to be refuted and the other 60% to be statistically insignificant. In the end, they propose a hybrid approach utilizing the more significant factors of all of the approaches which are claimed to outperform the other methods. Our study uses real faults for evaluation. Besides, we locate a culprit commit in a group of batch tested commits. This is not the goal of the above-mentioned approaches.

Liblit *et al.* [LNZ$^+$05] propose a statistical approach for identifying software bugs known as Liblit05 in the literature. They rerun the tests many times and utilize a random sampling methodology for instrumenting the software executions and calculate probability values of the software predicates to be faulty based on whether they have been observed in successful or failed executions. Then they report the predicates in a prioritized order based on a calculated importance score. This approach does not directly relate to our study as we do not do software instrumentation.

Liu *et al.* [LFY$^+$06] propose another statistical technique named SOBER for fault localization which attempts to improve Liblit05 [LNZ$^+$05]. Their approach is inspired by the concepts of hypothesis testing. As opposed to Liblit05 which selects predicates correlated with program failures, they model predicate evaluations in both correct and incorrect runs and treat the divergence of the two models as a measure of fault relevance. This approach does not directly relate to our study as we do not rely on dynamic analysis of the runtime behavior of the program executions.

Wong *et al.* [WDX12] propose a cross-tabulation statistical technique for fault localization. They use the coverage information of the code statements and the relevant test execution results to calculate suspiciousness probabilities for code statements. In their technique, they construct a crosstab for each statement and then a hypothesis test is used to measure the dependence between the execution results and the coverage of each statement. The exact suspiciousness of each statement depends on how much its coverage and the execution results are correlated. This study does not directly relate to our approach as we do not use test coverage information.

You *et al.* [YQZ12] propose a statistical approach for fault localization using the execution sequence of the program predicates. Predicates are branches, returns, and scalar-pairs. They note the predicates of a program as the vertices of a graph, the transition of two sequential predicates in the execution trace of a program as edges of the graph, and the frequency of each transition as the label of the edges. Then they apply hypothesis testing to evaluate the statistical difference between edge values among the failed and passed executions of the program. As for all of the statistical fault localization methods, this approach also relies on collected information from plenty of program executions. This study does not relate to our work as we do not rely on coverage information of the programs and also we do not rerun the same version of the software multiple times to find the root cause of the failures.

Baah *et al.* [BPH10b] propose a statistical fault localization technique based on causal inference. They present a causal effect estimator that covering a statement has on program failures. The estimator is based on a linear regression model trained on the data from coverage of statements and their predecessors in the control-dependence graph. This study does not directly relate to our work as we do not utilize code coverage data and do not predict program statements suspiciousness values.

Modi *et al.* [MRA13] propose a statistical bug localization technique that attempts to improve the predicate-based bug localization techniques by using program phases. Program phases are intervals of program execution that show similar values for architectural metrics like branch misprediction rates, cache miss rates, CPU or Memory usages. Predicate-based techniques, on the other hand, collect predicate profiles such as branch conditions, return values, comparison of pairs of variables over multiple program executions and correlate that with the execution results. This study does not directly relate to our work as we do not utilize program instrumentation.

Renieres *et al.* [RR03] propose a fault localization technique based on program spectra and models. Program spectra are collections of program execution features such as the number of times each line of the program is executed, function call counts, program paths, and program slices. After collecting the different instances of program spectra from multiple runs, they abstract the spectra so that they can be comparable. They need at least one failed spectrum and multiple successful ones. Then they build a model from the successful spectra and differentiate that with the failing run and map the results back to source code as suspicious locations. They use different approaches for their modeling and differentiating. Union of the spectra of all successful runs is one and finding the intersection of them is another. Also, they propose a nearest neighbor approach to find the closest passed run to a fail run and find the difference between them. This study does not directly relate to our approach as we do not utilize any software instrumentation or profiling methods.

Jones *et al.* [JHS01, JHS02] propose a visualization tool based on an approach named Tarantula for highlighting the program statements based on their possibility to be faulty for assisting with fault localization. They use test results and also the code coverage information of the test cases to calculate color codes for suspicious source code statements. These color codes are supposed to help the developers localize the bugs. For calculating the statements, they use a combination of measures like the number of failed test cases that do not cover a statement or number of successful test cases that cover a statement. This study does not directly relate to our work as we do not rely on code coverage and also we aim to locate a culprit change in a batch testing process as opposed to source code statements.

Wong *et al.* [WDGL14] propose a coefficient-based fault localization technique named DStar which is based on code coverage information and test results. Like other coefficient-based techniques, they use measures like the number of failed test cases that cover the statement or number of successful test cases that do not cover the statement to calculate a suspiciousness value for each statement of the code and report them to the user. DStar's coefficient formula proves to be superior to its similar fault localization counterparts. This study does not directly relate to our work as we do not use code coverage in our method.

Weiser *et al.* [Wei81] propose the concept of program slicing and specifically static slicing. A program slice is the portion of the code statements that could affect values of a certain set of variables at certain points of the program. The idea of slicing is to reduce the number of statements that a developer needs to check and debug. Static slicing is the practice of doing so using static analysis of the program source code. This work does not directly relate to our study.

Korel *et al.* [KL88] propose dynamic slicing as an improvement to static slicing. The problem with static slicing is that it can involve excessive statements that do not really affect the variables of interest as it does not consider the runtime states of the program during the execution. Dynamic slicing is proposed to fix this issue so that fewer and more accurate set of statements will be left for the programmers to debug. The dynamic slice of an incorrect value at a certain point of a program (or known as a backward dynamic slice) includes all of the statements that actually affect the value of a variable at that certain point [ZGG]. This work does not directly relate to our study.

Gupta *et al.* propose two other dynamic slicing approaches, forward dynamic slice of the minimal failure-inducing input difference and the bidirectional dynamic slice of a critical predicate [GHZG05, ZGG06]. According to [ZGG], the minimal failure-inducing input difference is the part of the input that is found to cause a program failure given that input and the critical predicate is an execution instance of a conditional branch predicate such that if the outcome of the predicate's execution instance is reversed, the program terminates producing the correct output. In [ZGG] the authors propose to use a combination of backward, forward and bidirectional dynamic slicing methods to come up with a smaller set of faulty statement candidates known as multiple points dynamic slices. These studies do not directly relate to our work.

Sterling *et al.* [SO05] propose an approach for isolating bugs named program chipping. They use specific simplifications on different program elements like blocks, loops, and if statements to chip away parts of a program that do not relate to a failure. This approach is similar to program slicing, however, with slicing, a set of statements are extracted based on the program behavior with respect to a variable or group of variables, but in program chipping, statements are chosen based on the behavior of the overall program, somehow similar to delta debugging. This study does not directly relate to our approach.

Mohapatra *et al.* [MMK04] propose a dynamic slicing technique specifically for object-oriented programs to take into account matters like classes, dynamics binding, encapsulation, inheritance, and polymorphism. Their technique is based on the extended system dependence graph proposed by Horwitz *et al.* [HRB88]. This approach does not directly relate to our study.

Agrawal *et al.* [AHLW95] propose a fault localization technique based on execution slices. Static slices are statements that may affect the output of a variable. Dynamic slices are statements that actually do affect the output of a variable under a specific input. On the other hand, execution slices are all of the statements that are executed while running a test under a specific input. Execution slicing is basically based on collecting test coverage information which is relatively easy to collect, compared to resources required for constructing dynamic slices [WGL$^+$16]. It is shown that by getting a difference between the execution slice of a failed and passed test run (known as execution dice), the debugging context can be effectively narrowed down for the developers. This study does not directly relate to our approach.

Wong *et al.* [WSQM03] propose a technique using software architectural designs. They attempt to extend the ideas of fault localization and execution slicing from the source code level to high-level specification and description language (such as SDL). This work does not directly relate to our study.

Wong *et al.* [WQ06] propose a program debugging technique using execution slices and also data dependency between blocks of a program. A basic block or also shortly known as a block is defined as a sequence of consecutive statements or expressions that do not contain any transfers of control except at the end. They first use the difference between the execution slice of a failed test execution and its passed versions. Then they refine the search by using the data dependency between code blocks, i.e. cases where a variable used in a block is defined in another block or vice versa. This study does not directly relate to our approach.

Ren *et al.* [RST+04] propose a tool for change impact analysis of Java programs. Their approach uses two different versions of a software to extract a set of atomic changes. These atomic changes include added, deleted and changed classes, added, deleted and changed methods, and etc. Then they construct a dynamic call graph for each of the available regression tests using the test execution traces. Using this analysis they can report a subset of regression tests that can potentially be affected by a given set of atomic changes. Also, they can determine what subset of atomic changes can be related to a test failure. The concept of this study is very closely related to our work with some differences. They harvest atomic changes, like method or class modifications, and use those as the building blocks of their analysis. However, we focus on the change (commit) level. It's very easy to track a change back to a developer or just revert the change to get the development back on track pretty quickly. A more important difference is their usage of tests call graphs or basically coverage information. We have proposed an empirical approach that can infer the correlation between file changes and test failure over the time, without requiring to analyze the tests call graphs or coverage information. We believe extracting call graphs are not always easy as in the case of our industrial partner which have complicated and interleaved hardware level programming procedures.

2.4 Fault Localization in Change Level

In the previous section, we went over different studies that aim for finding the faulty statements or at least narrowing down the search for the buggy statements for the developers. Our work, however, aims to find a culprit change in the first place. Companies sometimes test groups of commits (changes) together in a batch, with the goal of saving part of their testing infrastructure and resources. In such cases, after a test failure, the first problem is to find which of the commits have been the culprit. In this section, we review some of the studies related to our work in finding the culprit changes in testing environments.

Ziftci et al. [ZR17] propose an approach for finding the culprit commits using the software build system. They have experimented their approach in Google using Google's build system. The problem they attempt to solve is the matter of finding the culprit change when a post-submit test has failed on a group of changes together. They assume changes get submitted after passing the pre-submit tests. They describe pre-submit tests to run on every change separately, so it is obvious which change was the root cause in case of a test failure. However, they define post-submit tests as the lengthy and more expensive tests that cannot be run on every change individually. In turn, they are run on every n changes together or periodically every n hours. For these tests, a failure requires pinpointing the culprit change. They propose an approach that suggests suspiciousness values to the changes involved in a post-submit test failure. They use build files to generate a build graph so that they only consider the changes that have modified any of the files that the test of interest depends on. The bigger the size of the change, more suspicious it is. Another factor is the dependency distance between the files and the test of interest. The farther the dependency is, the less probable the change is to be the culprit. This study directly relates to our work with some considerations. First of all, our study aims to find the culprit changes in an environment that the code is not submitted (merged) yet. So we don't have any specified order for the group of changes. On the other hand, we do not use static test coverage or test dependency information. Instead, we propose an empirical approach that learns the correlations between files changes and test failures over time. Our approach assumes much less knowledge from the software ecosystem and therefore can be more easily integrated into other software environments. Another point of difference is the evaluation approach. They do not rerun the tests and therefore cannot locate the exact culprit change deterministically. Instead, they rely on human feedback which may be easily biased and may not be reliable. In our study, however, we evaluate the results against the exact deterministic culprit of every batch testing process which is available from the test executions history.

Kamei *et al.* [KSA$^+$13] propose a risk analysis approach in change level. They construct a logistic regression model for analyzing changes using different factors under six high-level categories of diffusion, size, purpose, history, and developer experience to calculate the risk values. The number of modified subsystems, lines of code added, the average time interval between the last and the current change, and recent developers experience are among the utilized metrics. This work is closely related to our study and can be a good complement to our culprit change prediction methodology. The difference, however, is that this study does not deal with any test failure or a deterministic fact that there is a fault somewhere in the system or not. So their predictions are just guesses implying that there is a risk of having a bug somewhere in the code. Our approach, on the other hand, deals with the concrete test failures and the fact of having bugs somewhere in the code that need to be located.

Yang *et al.* [YLX$^+$15] propose a deep learning based technique for predicting the faulty changes. They use an advanced deep learning algorithm named Deep Belief Network for extracting a set features for measuring the changes. Then they train a logistic regression classifier for predicting the risk values of the changes. Similar to [KSA$^+$13], this approach also just predicts the risks associated with different changes but does not associate them with any concrete test failure. Our approach, however, starts from a concrete test failure and attempts to locate the change associated with that test failure.

Kim *et al.* [KJZ08] propose an approach for classifying the developer changes as buggy or clean. They extract features like the lines modified in each change, author and time of the change, complexity metrics and etc. from software revision history and train a Support Vector Machine classifier to predict the changes as buggy or clean. This approach also just examines the risk associated with each submitted change without connecting them to a concrete fault localization context of a test failure. Our approach, on the other hand, does so using an empirical approach that points to the culprit change that is involved in a test failure.

Śliwerski *et al.* [SZZ05] propose an approach known as SZZ for finding bug-inducing changes. They start with a bug fix report, then they find where the changes have been applied to and then extract the last time those locations have been modified. In this way, they introduce the associated changes as bug-inducing ones. They use the built-in annotation feature of software configuration management (SCM) in their analysis. This study is related to ours in the sense of trying to locate a buggy change, however, the approach they use and its context is different from ours. We aim to locate a faulty change among a group of batch tested commits. On the other hand, this approach attempts to locate the bug-inducing change which is associated with a bug fix change.

Kim *et al.* [KZPW06] similar to [SZZ05] propose an approach for locating the bug-inducing changes after a bug fix change has been reported. This study attempts to improve [SZZ05] by using annotation graphs [ZKZW06] instead of the built-in annotation feature of SCM used by [SZZ05]. Their new approach shows to have higher accuracy in finding the bug-inducing changes. This study aims to find a related buggy change to a reported bug fix change. Therefore, it does not directly relate to our study. We attempt to locate a buggy change in a group of changes tested in batch.

Wen *et al.* [WWC16] propose an information retrieval (IR) based technique for locating bug-inducing changes. As opposed to [SZZ05] and [KZPW06], this approach does not rely on a bug fixing change to locate the relevant bug-inducing one. Instead, this approach relies on bug reports. Bug reports are submitted by testers or developers whenever a bug is found somewhere in the system. A bug report consequently leads to a debugging process. Their approach is based on the assumption that change logs usually include descriptions about the intention or functionality of the changed pieces of a code which share common tokens with their relevant bug reports. Therefore, they propose an IR-based bug localization technique called Locus and predict bug locations both in the change and file levels. As this study relies on text analysis of the bug reports, it does not directly relate to our study as we do not utilize such information.

Git bisect [GBS] is a tool that helps to find the culprit change in a software revision history. It uses binary search on a given range of commits from the user. It is manual and requires rerunning the tests. Although we were inspired by this approach, it does not directly relate to our methodology.

Chapter 3

Improving Test Effectiveness Using Test Executions History: An Industrial Experience Report

This chapter has been submitted verbatim and been accepted in the SEIP track of the ICSE 2019 conference

3.1 Abstract

The cost of software testing has become a burden for software companies in the era of rapid release and continuous integration. Our industrial collaborator Ericsson also faces the challenges of expensive testing processes which are typically part of a complex and specialized testing environment. In order to assist Ericsson with improving the test effectiveness of one of its large subsystems, we adopt test selection and prioritization approaches based on test execution history from prior research. By adopting and simulating those approaches on six months of testing data from our subject system, we confirm the existence of valuable information in the test execution history. In particular, the association between test failures provide the most value to the test selection and prioritization processes. More importantly, during this exercise, we encountered various challenges that are unseen or undiscussed in prior research. We document the challenges, our solutions and the lessons learned as an experience report. Our experiences can be valuable for other software testing practitioners and researchers who would like to adopt existing test effectiveness improvement approaches into their work environment.

3.2 Introduction

Testing is an important, yet time consuming and costly process, especially for large software systems. Prior research estimates that testing consumes between 30% to 50% of the time in software development life-cycle [PZTM13]. For example, an industrial case study shows that it takes over two days to complete testing on a medium size video conferencing system [MGS13].

The cost of testing has become a burden for software companies during rapid release [Bec00], where continuous integration techniques are widely adopted in practice to receive feedback from testing as soon as changes are made to the source code [DMG07]. With thousands of commits made to the source code every day, it is challenging to keep up with the speed of development. Google's version control repository receives over 16,000 commits every day [PL16], which results in a median of 27 test requests per minute [ZSR18]. Our industrial collaborator Ericsson has faced the same challenges for testing its large-scale software systems. Moreover, the changes in their testing processes may be even greater due to the complex testing infrastructures that are purposely designed for each software subsystem.

Test selection and test prioritization are proposed by prior research to improve the effectiveness of test executions [ASD14, HGCM15, ERP14, KP02, ZSR18]. With test selection, tests are either executed or skipped on the fly. With test prioritization, tests are reordered such that more test failures can be captured earlier using limited testing resources. We leverage prior research to assist Ericsson with improving the testing process of one its large-scale software systems. Since prior research is typically evaluated on a particular industrial subject system (e.g., from Microsoft or Google), it is unclear to what extent the existing approaches can improve the test effectiveness in our subject system and whether there are challenges that are unseen by prior research.

In this paper, we share our industrial experience for adopting test selection and prioritization approaches to improve the test effectiveness of one of the large software systems at Ericsson. In particular, the adopted approaches are based on historical test failure frequency, test failure association, and the costs associated with the testing process. In order to evaluate the usefulness of these adopted approaches, we simulate applying these approaches using six months of testing data from a large-scale system at Ericsson. Our results show the importance of test execution history in enabling the test selection and prioritization approaches to help with software testing in practice. We encountered many engineering and design challenges for adopting and applying the approaches to the testing processes of Ericsson. In the end, we conquered the problems and documented the challenges, our solutions and the lessons learned as an experience report. We believe that our experience in adopting existing test selection and prioritization approaches can help software practitioners and researchers who want to adopt software testing approaches into their work environment.

The major contributions of this paper are:

- We adopt and evaluate test selection and prioritization approaches with the goal of improving test effectiveness in a large industrial system with a complex testing infrastructure.
- We demonstrate the value of test execution history for improving test effectiveness in a large-scale industrial system in practice.
- We provide an industrial experience report that documents the challenges that are encountered and our lessons learned during the adoption process of the test selection and prioritization approaches.

The remainder of this paper is organized as follows. Section 3.3 describes the background of the subject system and its testing process. Section 3.4 discusses the approaches that are adopted by our study to improve the test effectiveness of our subject system. Section 3.5 presents the results of evaluating the adopted approaches. Section 3.6 discusses the challenges that we have encountered and the lessons learned during the experiments. Section 3.7 discusses the threats to the validity of our findings. Section 3.8 presents other related research in the literature. Finally, Section 3.9 concludes the paper.

3.3 Background and Subject System

In this section, we explain the background required for this paper, i.e., the subject system that we studied at Ericsson. Figure 1 presents an overview of the testing process of our studied system.

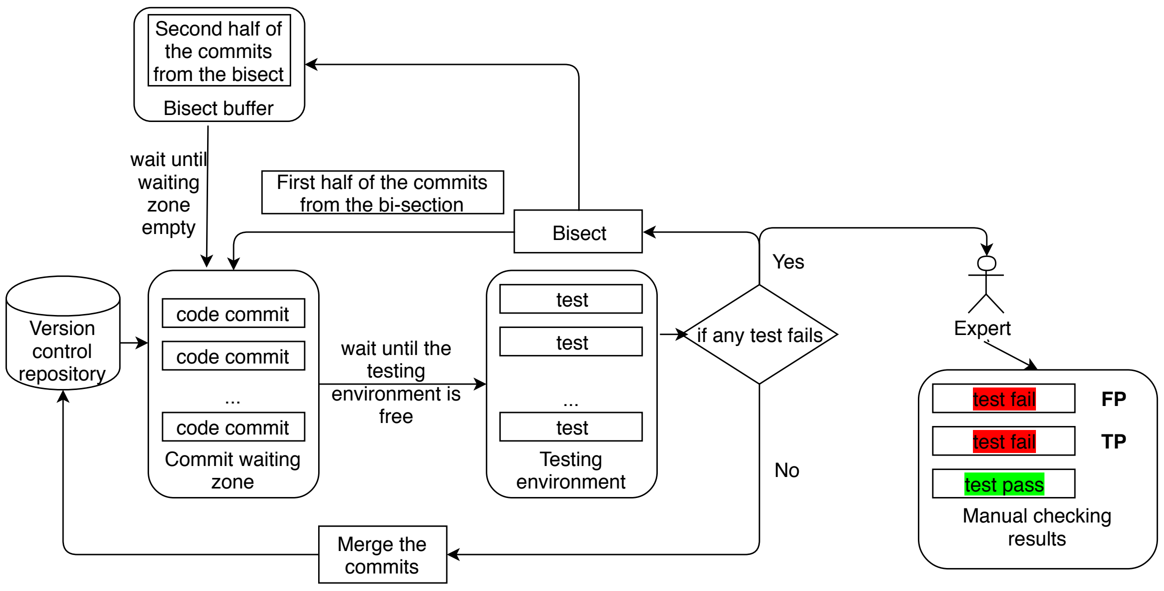

Figure 1: An overview of the testing process of *ELS*.

3.3.1 Subject system

The subject system in this study is a large-scale software component at Ericsson. The software system has a large user base across the world and is currently being developed on a daily basis for new features and performing maintenance activities. The teams that develop this system consist of a large number of developers and testers across the globe. The system is developed using a modern typical programming language that is hosted in a typical version control repository. To ease the discussion about the subject system, we refer to it as *ELS* (Ericsson's Large-scale System) in the rest of this paper. Due to the criticality of the system, our study is conducted based on simulating rerunning tests on the software system according to the test results from the six months of real test execution data.

3.3.2 Testing process

The testing process in *ELS* is conducted on special purpose testing infrastructures with limited resources. Therefore, in general, the changes to the source code of *ELS* are queued to get tested in such a complex environment. In particular, the testing process consists of four steps. 1) Collecting commits. The commits that are made in the version control repository are collected and put in a queue to be tested as soon as the testing environment is available. 2) Testing the commits. Once the testing environment is free, commits from the queue are consolidated as a batch and are moved into the testing environment. There exists a large number of pre-designed tests which are executed on the new commits. If all of the tests pass, the commits will be merged into the main trunk of the version control repository. If any of the tests fail, the commits will be sent back for a bisection process [Zel99]. 3) Bisection. The bisection process splits the commits of a failed batch in half by their time stamps. The first half of the commits are sent directly to the queue, with other commits that are already waiting in the queue to be tested. The second half of the commits will be waiting for the queue to be empty to enter the queue. 4) Manual check of the test results. Due to the complex testing infrastructure, not all test failures are due to software bugs. Therefore, once the test failures are located in one single commit (after multiple iterations of the bisection process), a system expert will manually check the failures. A test failure will be labeled as a false-positive if it is not caused by a software bug, but rather by an infrastructure issue in the testing environment. Or it can be labeled as a true-positive if the test failure is identified to be an actual bug in the main software system. Please note that the testing process described above does not include all details about the software quality assurance process in Ericsson. There may exist other approaches, infrastructures or test stages available in the testing flow. However, for the scope of this study, we only focus on the above-described testing process.

3.4 Adopted Approaches

Software testing and regression testing improvement have been largely studied in the software engineering literature. Kazmi et al. [KJMG17] and Suleiman et al. [SAH17] provide extensive reviews on the recent studies in the literature for test selection and prioritization. In this section, we present the approaches that we have adopted in order to improve the test effectiveness of *ELS* in Ericsson. In particular, we adopted approaches that perform test selection and test prioritization.

3.4.1 Test Selection

We adopt test selection approaches that are based on prior test failure frequency, association, and the costs of the testing process. Test selection approaches are applied before the start of each batch, for which all of the test execution results before the day of testing are used as learning data for our analysis. We do not learn from the test execution data from the same day of the test executions, as test failure results need to be manually labeled as true-positives or false-positives by the testers by the end of each day. Therefore, we only obtain the latest labels at the end of each day.

Based on Failure Frequency

Intuitively, tests that previously failed frequently are more likely to fail again later [ASD14, ZSR18]. Therefore, the frequency of past test failures can be used as an indicator for suggesting test selection opportunities.

Microsoft's FreqSelect Anderson et al. [ASD14] propose an approach that calculates the frequency of test failures using the test executions history. The tests that failed more frequently in the past are recommended to be selected again later.

FreqSelect Our adopted approach is closely related to the test selection approaches proposed by Anderson et al. [ASD14], where only the tests that have prior failures are selected. In addition, to consider the cases where test failures can be false-positives in our testing process, we only consider the tests that have prior true-positive test failures as opposed to considering all of the test failures.

Based on Failure Association

As the prior study shows, there exists a large number of co-failures in test executions [ZSR18]. Hence, associations can be effectively leveraged for improving test effectiveness [ZSR18].

Microsoft's AssocSelect Anderson et al. [ASD14] perform association analysis on the test failures. In particular, failures in certain subsets of tests are used to determine other subsets that are likely to fail. With the identified association rules, Anderson et al.'s approach selects the tests that are more likely to fail again.

AssocFreqSelect We adopted *Microsoft's AssocSelect* with the goal of minimizing the redundancy among tests by only selecting the cheapest test to run if multiple tests are associated with each other. In particular, we leverage test failures in each batch (see Section 3.3) to perform association rules mining using the Apriori algorithm [AMS$^+$96]. We obtain a list of association rules pointing to the test cases that tend to co-fail with each other. Some rules may have low confidence and should not be used to perform selection. Therefore, we only consider the association rules that have over 0.8 confidence.

Afterward, we use these rules to dynamically select the test cases that are most effective and at the same time have the least amount of redundancy among each other. We first rank the test cases based on their probability of finding true-positive test failures as extracted from the test executions history (c.f. the *FreqSelect* approach). Then we iterate through the list from the top one by one and check if there exists any association rule between the current test and any of the previously analyzed ones. If there exists such a rule, we only keep the test that has a shorter average execution time in the list. In this way, we remove the redundancy among the tests while shortening the tests duration time and maintaining the effectiveness of the tests in finding true-positive failures.

The goal of our *AssocFreqSelect* approach is different from *Microsoft's AssocSelect* by Anderson et al. *Microsoft's AssocSelect* aims to select the tests that are more likely to fail using failure associations. However, *AssocFreqSelect* aims the opposite, i.e., associated test failures may be redundant and they should not be executed. *AssocFreqSelect* removes tests based on the existence of association to avoid redundancy among the test failures.

Based on Cost

Test execution comes with a cost, especially for the complex testing infrastructure that is used by *ELS*. With such high testing costs, tests can be skipped if skipping them is more cost-effective than executing them [HGCM15].

Microsoft's Theo Herzig et al. [HGCM15] propose a tool named THEO that stands for Test Effectiveness Optimization using Historic data. THEO is designed based on a cost model that dynamically skips tests when the expected cost of running one exceeds the expected cost of skipping it. In particular, *Microsoft's Theo* calculates two values for every test execution, cost of execution and cost of skip, and decides which action is more economical. Similarly, in our *TheoSelect* approach, we select test cases based on an online cost analysis for every test case execution. Two costs are calculated for every test case execution: the cost of execution and cost of skip, calculated as follows:

$$\begin{aligned} cost_{exec} &= cost_{machine} + (P_{FP} * cost_{inspect}) \\ cost_{skip} &= P_{TP} * cost_{escaped} * time_{delay} * \#engineers \end{aligned} \tag{1}$$

In this equation, $cost_{machine}$ is an estimate of the costs associated with running a test in the test environment for a certain amount of time. $cost_{inspect}$ is an estimate for the time delay and the costs associated with the engineers that will perform the inspections after a test failure. This parameter mainly affects the costs associated with running a test case when a false-positive needs to be inspected. $cost_{escaped}$ represents the average cost of an escaped defect. #engineers is the number of engineers that will get involved when a new slip-through is introduced into the test flow. $time_{delay}$ is the amount of time delay that a new slip-through will impose on the test flow. Moreover, P_{FP} and P_{TP} are the probabilities for finding a true-positive or a false-positive for every test case. These probabilities are extracted from the test executions history. Consequently, after evaluating the equations, if the cost of running a certain test case is shown to be higher than skipping it, the test case execution will be skipped or vice versa.

***TheoSelect*:** We have adopted the approach proposed by Herzig et al. [HGCM15] as is. We call this approach *Microsoft's Theo* and is identical to our *TheoSelect* with different parameters. *Microsoft's Theo* is a generic test selection tool that selects test cases based on a cost analysis of each execution on the fly. We cannot unveil the exact values that we have chosen for the parameters due to confidentiality reasons. However, as the ratio of the parameters matter in our use case, we set $cost_{machine}$ to 1 unit, $cost_{escaped}$ to 13.1 unit and $cost_{inspection}$ to 30 units as the ones used by Herzig et al. [HGCM15].

3.4.2 Test Prioritization

Our main goal in test prioritization is to find more failures in shorter duration time. For this purpose, we have been inspired by some of the works in the literature as follows.

***Kim's Prio*:** Kim et al. [KP02] propose a test prioritization technique based on test execution history. The prioritization technique gives priority scores to each test for their execution. They propose three approaches for calculating the scores. First, the tests that have not been executed recently are given a higher score. Second, the tests that have revealed more faults recently will receive a higher score. Finally, a higher score is given to the tests that cover functions which are infrequently covered in the past testing sessions.

***Google's Prio*:** Elbaum et al. [ERP14] propose another test selection and prioritization technique. Their approach prioritizes tests that have recently found failures in a specified time window based on the previous execution history.

***Prio*:** Our prioritization technique is similar to *Kim's Prio* and *Google's Prio*. In our approach, we give a higher priority to tests that have found more failures and have a shorter average execution time. We rank the test cases based on the ranking values calculated as follows:

$$priority\_value = \frac{\#total\_failures}{total\_executions\_duration} \tag{2}$$

where the $\#total\_failures$ is the true-positive failures detected by the tests and the $total\_executions\_duration$ is all the time that is spent on executing the tests. The total duration of test execution has been incorporated into the equation in order to normalize the effectiveness of the test cases. In this way, we give a higher priority to tests that find more failures with shorter execution time, i.e., less effort.

Comparing *Kim's Prio* and *Google's Prio*, our approach relies only on the test executions history, such as the number of true-positive failures and test executions' duration to calculate the priority values. We do not use test coverage information since all too often, it is not available in the test process of *ELS* in Ericsson. Similar to test selection approaches, we learn on all of the test execution results until one day before of every test.

3.5 Results of the Adopted Approaches

In this section, we present the simulation results of our adopted approaches (see Section 3.4) using six months of test execution data from *ELS*. We present the results for test selection approaches and test prioritization approaches will follow.

3.5.1 Test Selection

We use three metrics to evaluate the adopted approaches for test selection.

Total test execution time reduction: Our first metric demonstrates how much test execution time is reduced by performing each of our test selection techniques. In particular, we first calculate the total time that is needed to execute all of the tests. Then using each test reduction technique, we calculate the required time to only execute the tests that are selected (i.e., not removed) by each technique. Finally, we calculate the reduction percentage based on the total time needed for running all of the tests and running only the selected tests by each selection approach.

Number of slip-through test failures: Our second metric, reveals the percentage of the true-positive test failures that may have been missed due to a test not being selected. The value zero for this metric means that our test selection approach does not remove any test that would have been a true-positive test failure. Intuitively the lower the value of this metric is, the better the approach is. This metric is particularly important since the slip-through true-positive test failures may have a direct impact on the quality of the product and the end users.

Total cost reduction: This metric is a measure for showing the impact of the test reductions and their side effects in terms of a concrete cost value. For this metric we calculate the costs of running each test case or encountering a missed true-positive using the parameters given in Section 3.4. In particular, cost of running each batch can be calculated as follows:

$$\begin{aligned} cost_{batch} = \sum_{test\_cases} test\_case_{execution\_time} * cost_{machine} \\ + \#FP * cost_{inspection} \\ + \#slip\_throughs * cost_{escaped} * time_{delay} * \#engineers \end{aligned} \tag{3}$$

where the total cost is the sum of the costs of the test infrastructure during the test execution, the effort by the practitioners to inspect any false-positive test failures and the cost of the slip-through test failures if an important test is not selected. Afterward, we calculate the total cost by assuming that all of the tests are selected, i.e., there exist no slip-through test failures. Finally, the total cost reduction is calculated based on the cost of using each test selection approach and the total cost of selecting all tests as follows.

$$reduction\_percentage\_in\_cost = 100 * (1 - \frac{simulated\_total\_cost}{total\_cost}) \tag{4}$$

Table 1 shows the results of our evaluations for each of our approaches. The results show that by only considering the frequency of past test failures, reducing the test execution time is not trivial. In fact, by only saving 0.01% of the test execution time, the approach let 10.22% of the true positive test failures untested. Such missed true-positive failures result in an increase in cost. On the other hand, the *AssocFreqSelect* approach shows significant improvement over the *FreqSelect* approach. The *AssocFreqSelect* approach is able to maintain a similar slip-through rate but reducing both the time of test executions (13.91% reduction) and the costs (13.08% reduction). Our results confirm the findings from recent research by Zhu et al. [ZSR18] where the association between the test failures are found to be effective for re-ordering the tests.

Associations between tests are a valuable source of information for improving test effectiveness.

The cost analysis approach, i.e. *TheoSelect*, is designed to optimize the cost of testing. The results for this approach demonstrates much higher (almost three times) cost reduction in testing

Table 1: Overall comparison of the three adopted test selection approaches.

	Reduction in execution time	Slip-throughs	Reduction in cost
FreqSelect	0.01%	10.22%	-0.73%
AssocFreqSelect	13.91%	11.36%	13.08%
TheoSelect	41.78%	34.65%	39.23%

compared to *AssocFreqSelect*. On the other hand, the slip-through test failures are also almost three times of that compared to *AssocFreqSelect*. Therefore, the practitioners can choose to lower the total costs and prefer to tolerate the fact that some failures may be missed by the testing process. This can be especially tolerable if it can be proved that there are other mechanisms in the flow that can catch these missed failures (c.f., Section 3.6). Otherwise, having too many slip-throughs, despite reducing the costs can have a significant impact on the end users.

Even by taking advantage of a cost-based test selection approach, practitioners may still face the trade-off between the slip-through test failures and the cost of testing.

3.5.2 Test Prioritization

There are generally two criteria for evaluating test prioritization; Reaching the first test failure, or reaching all test failures as early as possible [ZSR18]. In order to evaluate the adopted approach, we consider both of the two metrics for our evaluations.

Reduced execution time until the first failure. Our first prioritization specific metric simply accounts for how fast a given order of running tests can find the first test failure. For each batch, we first reorder all of the tests using our test prioritization approach. Based on the prioritized order, we calculate the total execution time until we encounter the first test failure. We compare the test execution time to the actual test execution time that was spent to reach the first test failure in the test execution history from *ELS*. In order to minimize the bias of the original order in the test execution history of *ELS*, we also compare the test execution time with prioritizing the tests in random order. We repeat the random prioritization 1,000 times for each batch and calculate the average execution time needed to reach the first test failure.

Area under the curve of the cumulative lift chart of the test failures. The second metric compares the order given by our adopted approach with the optimal order. The optimal order gives priority to the tests that would fail in each batch and have shorter execution time. In particular, we use cumulative lift charts [MK09]. Figure 5 shows an example of the cumulative lift chart of

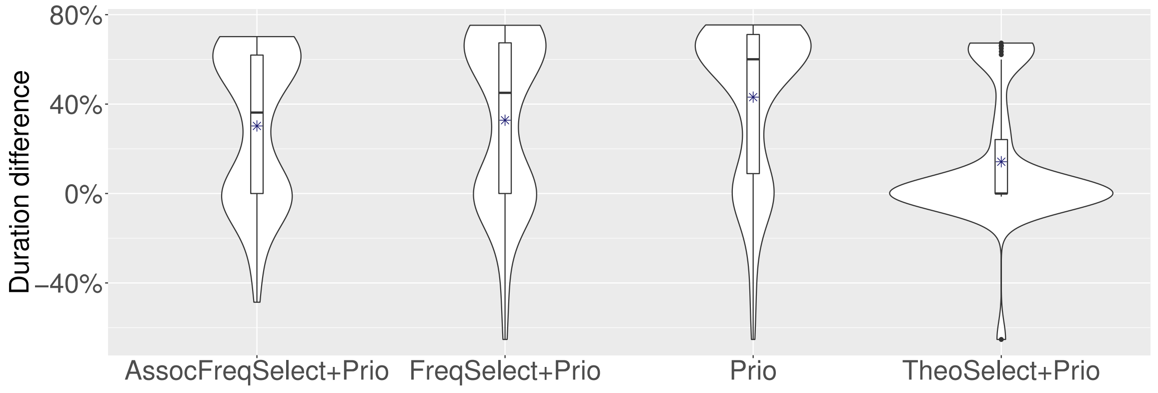

Figure 2: Bean plots of test execution time reduction by comparing our prioritization with the original order

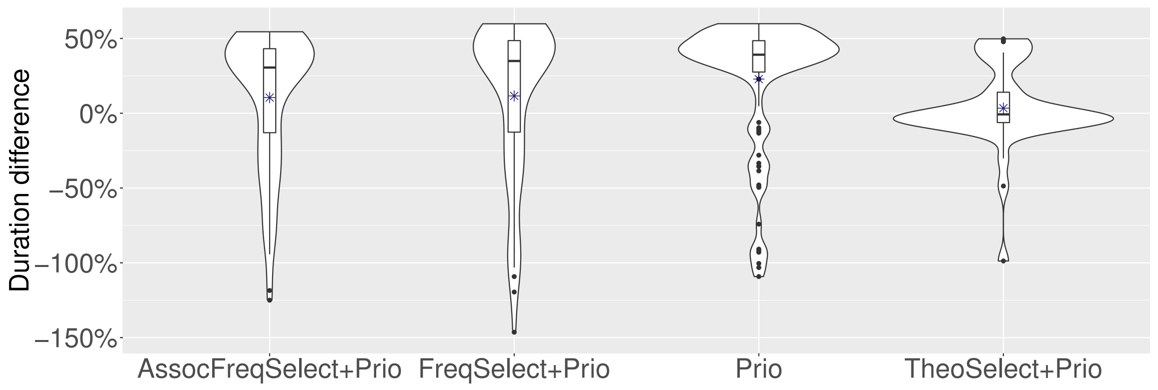

Figure 3: Bean plots of test execution time reduction by comparing our prioritization with the random order

the test failures of a batch, where the y-axis shows the number of test failures and the x-axis shows the percentage of the spent test execution time so far. For each batch, we generate such chart for both the optimal order and the order given by our adopted approach. The effectiveness of the prioritization technique is evaluated by the size of the area under the curve in each chart. Hence, for each batch, we calculate the size of the area under the curve for the optimal order and the given order by our adopted approach. We calculate the percentage (P_{opt}) of the area under the curve of the optimal order, that is covered by the order from our adopted approach. Therefore, P_{opt} has a range from 0 to 1. The closer P_{opt} is to 1, the better our prioritization algorithm is, i.e., closer it is to the optimal prioritization of the tests.

Our test prioritization approach can be used after the test selection approaches or as a standalone approach (as practitioners may not want to skip any tests). Figures 2 and 3 show the evaluation of our approaches for reaching the first failure, i.e., the reduced execution time until the first failure

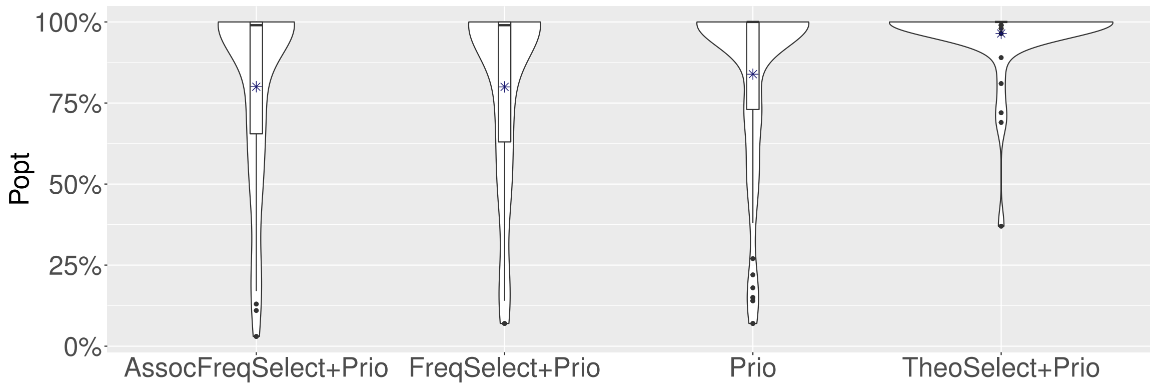

Figure 4: Distribution of the P_{opt} values of our prioritization results.

happens. The results show that our *Prio* approach can effectively reduce the test executions without having any impact on the product quality. Using our prioritization algorithm after our selection techniques leads to further reductions in total test executions. However, this comes with the cost of introducing missed true positives into the product line and a worse test ordering. Figure 2 compares our prioritization techniques with the current default order obtained from the execution history of our subject system in Ericsson. This figure shows that all of our approaches demonstrate significant improvement over the default order in required test execution time for reaching the first failure. Particularly, our best approach, standalone *Prio* reduces the time for finding the first failure, with a median of 60.05%. Figure 3 shows similar evaluations for comparing the results of our approaches to the results obtained from averaging 1,000 iterations of random order. Although prior studies show that prioritizing tests in random order can be surprisingly effective [ZSR18, KP02], our standalone *Prio* approach, demonstrates significant improvement over the averaged random order, with a median of 38.01%.

Figure 4 presents the results of our evaluations using the metric related to the area under the curve of the commutative lift chart of the test failures (P_{opt}). The median value of P_{opt} for each batch is almost 100% for all of our adopted approaches. Such results show that for almost half of the batches, the order provided by our approaches are either exactly or very close to the optimal order. The results show that similar P_{opt} performance is demonstrated when our *Prio* approach is used standalone or combined with *FreqSelect* or *AssocFreqSelect*. On the other hand, although combining *Prio* with *TheoSelect* shows a better prioritization, it is considered to be due to the removal of too many tests by *TheoSelect*, leading to a rather easier ordering for each batch.

Test prioritization by simply considering the past effectiveness of the tests can significantly help with reducing the time to reach the first failure as well as providing a close-to-optimal order for running the tests.

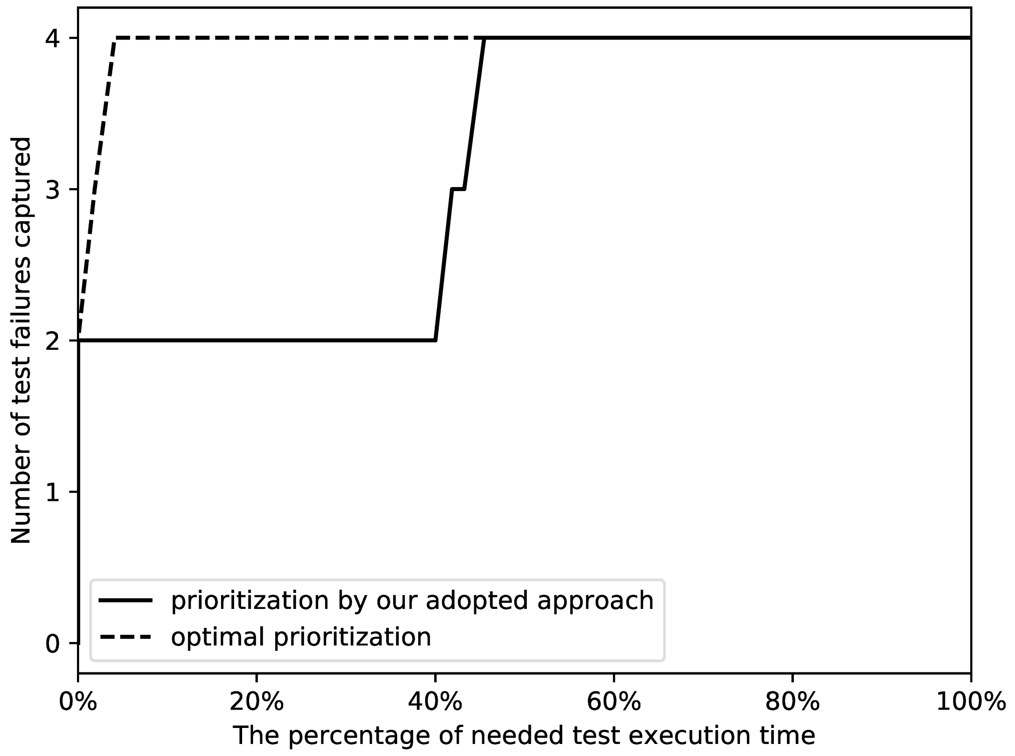

Figure 5: An example of the cumulative lift chart of test failures. The dashed line is for the optimal order and the other line is for the order given by our adopted test prioritization approach.

3.6 Challenges and Lessons Learned

In this section, we discuss the challenges that we faced during the implementation of our test selection and prioritization techniques in the industry and the lessons that we learned from our experiences. The challenges and lessons can shed more light on the path for other researchers and practitioners who would like to incorporate test selection and prioritization techniques in their software testing processes.

Challenge 1: Being reluctant to skipping unnecessary tests

Description: We started off our research by suggesting tests to be completely skipped or even removed from the test flow. However, our experience shows that the test managers and developers are reluctant to removing tests from the test flow in practice because they are concerned about leaving some product features untested. However, such a conservative approach is one of the reasons for the ever-growing costs of software testing in the industry.

There is always a rather conservative approach for test executions by having a preference for increasing the level of quality assurance of the software with the price of spending more time and resources. As a result, even though our initial analysis revealed that some tests can be removed due to their low effectiveness, the preference of the test managers was to keep them anyway. Therefore, we put more emphasis on our test prioritization techniques as in prioritization, no tests will be removed and only the order of running them will be changed.

Solution: We leveraged two approaches to address this challenge. First of all, we focused our techniques on test prioritization rather than test removal, i.e., instead of removing the tests, we adopted techniques that assign a low priority to running them. Therefore, the practitioners can still execute those tests if it is necessary and the required extra resources are available. Second, instead of only providing historical evidence, we aimed to provide an explanation on why those tests can be removed. For example, we can explain that Test A can be removed since the exercised APIs are obsolete, as opposed to only show that Test A has never detected a bug in the system.

Lessons learned: Due to the important role of testing in practice, developers and testers are often very wary of any changes in their testing processes. Therefore, they would prefer a solution that is less invasive and is less likely to do any harm in their processes.

> Lesson 1: To help with the engagement of our research, we find that a less intrusive approach is much easier to be accepted.

After having discussions with the practitioners, we find that some of the tests are implemented with a special purpose. For example, a test may be particularly designed to capture a very rare but critical issue. In such cases, testers would certainly not skip those tests since no other tests aim to detect the bugs for those scenarios.

> Lesson 2: Tests that are shown to be ineffective may be particularly designed for a special purpose as opposed to being obsolete. Test selection and prioritization techniques should consider the special characteristics of those tests.

Challenge 2: Coping with false-positive test failures

Description: Failures in test results may not always be associated with software issues. In testing processes of our industrial partner, we often observe test failures that are false-positives. One reason for this is the complicated hardware and test infrastructure involved in testing the code in Ericsson, the test failures may just be associated to a hardware failure or a noise in the system as opposed to an actual problem in the main product code. Such false-positives introduce challenges to our test selection and prioritization techniques. While learning the history of test executions, we need to take the false-positives into consideration. In addition, our approach should prioritize the tests that give fewer false-positive failures.

Solution: In order to address the false-positives in the test results, we first incorporate analysis of the test logs in order to assist with determining whether a test failure is due to the test infrastructure or a real product issue. After we identify the false-positives in our test results, we change the existing test selection and prioritization approaches to consider the existence of false-positives. The original *Microsoft's FreqSelect* and *Microsoft's AssocSelect* consider all test failures. However, our adopted approaches *FreqSelect* and *AssocFreqSelect* only consider the test execution history with true-positive test failures. Hence our approaches favor selection and prioritization of the tests that provide a higher number of true-positive test failures.

Lessons learned: We observe a prevalence of false-positive test failures in our experiments. Particularly, in certain software systems, the field environment can be noisy. Therefore, the testing infrastructure and the environment may contribute to a significant amount of test failures as false-positives.

> Lesson 3: Proactively addressing the false-positives in the test results can help practitioners in accepting our suggestions for test selection and prioritization techniques.

Challenge 3: Trade-off between slip-through test failures and testing costs

Description: Reducing the test costs can lead to an increase in the chance of having slip-throughs, i.e. having bugs in the final software product. The general goal for test selection and prioritization is to minimize both the costs of tests and also the number of slip-throughs. However, since there is a trade-off between the two, it is challenging to decide how much slip-throughs are accepted when aiming to reduce the cost of tests.

Solution: Although we provide different approaches for test selection and prioritization, there is no gold standard to help us decide what is a good trade-off between the slip-through rate and the costs. First, we tried to leverage the approach proposed by Herzig et al. [HGCM15] in order to generalize both the slip-throughs and the test costs into one consolidated cost metric. Moreover, we conducted meetings with practitioners to seek policies that may especially have been put in place for making decisions about the slip-through trade-offs.

Lessons learned: We were surprised to find that practitioners not always have a strong negative opinion towards having slip-throughs, with the argument that some bugs will be certainly caught in the following test stages and it may be more economical to let certain bugs slip through certain test stages, and be caught later in future test stages.

Lesson 4: Slip-through test failures are not always a negative phenomenon. Practitioners may prefer to lower the costs of the tests at certain stages and use other testing strategies to catch the slip-through failures in other stages of the test flow, given that the practice will lead to lowering the costs and will not affect the quality of the final product.

Challenge 4: Coping with test dependencies

Description: Tests are observed to be frequently co-failing during our experiments. Our approach *AssocFreqSelect* aims to identify the tests that often co-fail in order to reduce the testing costs. Intuitively, the co-failure of multiple tests may be due to test dependencies. Test dependencies are often considered harmful in prior research [ZJW+14]. For example, if two tests have a dependency on ordering (Test A needs to run before Test B), we may not leverage our test selection and prioritization techniques to reduce the testing costs.
Solution: We aim to identify and resolve these dependencies to improve the quality of the tests. However, in most of the associations that were identified by our *AssocFreqSelect* approach, we could not find any functional dependencies among the tests.

By manually examining the tests and conducting deeper investigations with the practitioners, we found that the tests that are found to co-fail with each other, are often both dependent on some common external resource. For example, it can be the case that both tests depend on a common hardware component that is currently failing to load. This can lead both of the tests to fail at the same time.

Current research on test dependencies does not focus on the cases where external resources are the root cause of the co-failures. Therefore, we had to manually label the tests with their external dependencies with the relevant hardware to help with maintaining the tests.
Lessons learned: There exist many tests in our study that co-fail with each other, even though they may be functionally independent or may be considered irrelevant to each other. We find research efforts may be allocated to provide more sophisticated static analysis techniques to automatically identify the root cause of the associations among the tests, leading to a more optimized test selection and prioritization approach.

Lesson 5: Tests that co-fail may not be functionally dependent on each other. Instead, they may co-fail due to their co-dependence on some external resource.

Challenge 5: Deciding on the granularity of the tests

Description: There exist different granularity levels for testing software code. Unit tests are more focused and localized while end-to-end system tests have a larger scope and scale. It is sometimes challenging for the practitioners to decide the right granularity level of each testing process. On one hand, the focused unit tests are less costly to run and investigate. On the other hand, the end-to-end system tests may be more realistic and be more effective in catching bugs that are more likely to exist in real environments.

Solution: We analyzed the associations obtained from the co-failure analysis of our *AssocFreqSelect* approach, with the goal of examining whether there exist any associations among the lower level unit tests and the more expensive end-to-end system tests. If so, practitioners may consider executing only the unit tests in order to save the costs associated with the end-to-end system tests.

Lessons learned: We find that there exists an insignificant overlap among the unit tests and the end-to-end system tests in our studied subsystem. In particular, we find that 80% of all true-positive test failures that are detected in the test results are detected by the end-to-end system tests and only 34% of them are detected by the unit tests. Such results show that 1) The more expensive end-to-end system tests capture most of the true-positive test failures and 2) The two test categories do not overlap heavily since only 13% of the failures are detected by both of the categories.

> Lesson 6: The test failures from the unit tests and the end-to-end system tests have little overlap. Removing the more expensive tests in this case in order to reduce costs may lead to major consequences in the overall product flow.

3.7 Threats to Validity

In this section, we discuss the threats to the validity of the findings of this paper.

External validity. Our study is only conducted on one area of the testing processes of one of the subsystems of Ericsson. Although *ELS* may carry many characteristics that are common in large-scale software systems, the findings and experiences may not be generalizable. For example, many of our findings are associated with the complex testing infrastructure that is specially designed for *ELS*. In addition, our results are only based on the simulation of six months testing results. However, adopting test selection and prioritization techniques on systems with shorter test execution history may result in different experiences. Finally, the parameters that are used for tuning our adopted approaches may be only suitable for *ELS*.

Construct validity. The evaluation of our adopted approaches is based on the simulation of the testing processes. Therefore, the actual quality of the system has not been evaluated by incorporating our approaches into practice. In addition, the goals of our test selection and prioritization approaches may not always align with the goals of the practitioners. For example, our association based test selection approach aims to reduce the redundancy among the tests while practitioners may opt to favor those redundancies to retrieve more information about the test failures. Further research may focus on evaluating such adopted approaches considering different goals of the practitioners.

The parameters used in cost functions in the *TheoSelect* approach are either estimated by the practitioners or adopted from the *Microsoft's Theo* approach. These parameters may not always exactly match the reality. Sensitivity analysis on the parameters may complement our findings and experiences.

Our test evaluation is based on training on all data that is available prior to the testing day. However, the testing results that are closer to the testing day may have a better power to predict the testing results. Future research may evaluate the best duration of training data in an industrial testing environment. Our test evaluation simulation has an assumption for independence among the tests, meaning tests can be re-ordered or skipped without impacting others. However, as observed by Zhang et al. [ZJW$^+$14] this assumption may not always hold. Future research by leveraging our approaches in practice may consider the dependencies among the tests for better test selection and prioritization results.

Internal validity. Our approach relies on the labels that are provided by practitioners to decide whether a test failure is a false-positive. These labels may be inaccurate and inconsistent due to the experience and subjective bias from the practitioners. Further validation of the labels may complement the results of our study.

The simulation of evaluating the adopted approaches does not consider the impact of the approaches on batches of commits which are being tested. For example, skipping certain tests may lead to different testing processes. However, our simulation cannot change the execution of tests in history. Future research may study such impacts by using the adopted approaches in practice to select and prioritize tests on the fly.

3.8 Related Work

Tremendous research efforts have been dedicated to improving test effectiveness. The effectiveness of the tests is optimized typically by test case minimization, selection, and prioritization, as proposed by Laali et al. [LLH$^+$16].

Static analysis and test coverage are leveraged as the major source of information to improve the effectiveness of the tests [NH17, SYGM15, WNT17, NABL, KSK, KR15, SS14, JKS12, KHT13, NTV$^+$14, QCR08]. For example, in a recent work, Saha et al. [SZKP15] propose to address the test prioritization problem by reducing it to a standard information retrieval problem. They assume that test cases and source code usually embody meaningful identifiers and comments which can be treated as natural language. Therefore, information retrieval techniques can be utilized on them to give priority values for running the tests. Nardo et al. [NABL] evaluate seven coverage-based test improvement techniques on a common carefully designed industrial system. Their findings show that finer grained coverage information can be leveraged to provide 79.5% savings in execution costs while maintaining a fault detection capability level above 70%. On the other hand, Koochakzadeh et al. claim that the test coverage information itself can be misleading for eliminating the test redundancy and can result in weaker test cases [KG10, KGM09]. Despite prior research, static analysis and test coverage based approaches are not suitable for our subject system. Due to the complexity of our subject system, the source code is often not or only partially available. Therefore, in this paper, we adopted approaches that depend on historical test execution information.

Historical information about the test executions is a valuable source for improving test effectiveness. A recent work by Zhu et al. [ZSR18] demonstrates the use of learning co-failures in test execution history to assist with test prioritization. Their approach prioritizes the tests by using the co-failure history of the tests and information about the tests that just recently failed at any moment. Noor et al. [NH17] build logistic regression models from the test executions history in order to rank the effectiveness of test cases. Anderson et al. [ASD15] propose a classification based approach for predicting the test failures based on the test executions history.

Research also leverages search-based techniques to optimize test effectiveness. Panichella et al. [POPDL15] propose an approach for test case selection using a customized genetic algorithm named Diversity based Genetic Algorithm (DIV-GA). Their approach is shown to improve the current state of the art by diversifying the solutions (sub-sets of the test suites) generated during the search process. Multi-objective based approaches are used in prior research by Tyagi et al. [TM14] and Souza et al. [dSPdAB14b] to assist with test prioritization and selection. Both pieces of research use the multi-objective particle swarm optimization technique. Their objective functions include covering more faults, maximizing the test coverage and minimizing the execution time. We do not consider the use of a search-based approach due to its limitation on scalability and the need for optimizing a large number of tests for the subject system.

Empirical studies are conducted on the testing practices. Labuschagne et al. [LIH17] propose a study for cost measurement of regression tests in practice. They study 61 Java projects running on Travis CI and find that 18% of test suite executions fail and that 13% of these failures are flaky. Among the non-flaky failures, only 74% were true-positives and the remaining 26% were false-positives. Their study emphasizes the importance of the works like ours for improving test effectiveness of continuous integration flows in large software projects.

3.9 Conclusion

Software testing consumes a significant amount of resources in software projects. Therefore, it is of major importance for our industrial partner to improve its test effectiveness. In this paper, we adopted and customized multiple test selection and prioritization techniques from prior research to improve the effectiveness of testing processes for *ELS*. By simulating the use of our adopted approaches on six months of testing data, we demonstrated the effectiveness of our approaches in reducing the costs, and test execution time. Our results show that for test selection, a combination of association and frequency analysis of the test failures can give acceptable results. However, our best results are obtained when doing test prioritization and using *Prio* approach. This approach which is basically based on frequency analysis of the past test failures significantly outperforms all of our other prioritization approaches. Therefore, we conclude that test prioritization is the most effective and the least invasive approach for saving costs in testing processes. On the other hand, test selection approaches can be used in scenarios where slip-throughs are tolerable. More importantly, we documented our challenges and lessons learned as an experience report. Such information is valuable for practitioners who are willing to adopt test selection and prioritization techniques into their day-to-day workflow. Our findings highlight the opportunities and challenges involved in leveraging test execution history for improving test effectiveness in both research and practice.

Chapter 4

Bisecting Commits and Guiding Test Bisection to find Culprits with Risk Models

This chapter will be submitted verbatim to a journal

4.1 Abstract

Software testing is one the costliest stages of software development life cycle. One approach to reducing the test execution costs is to group changes into groups and test all the changes as a batch. In this work we study the impact of batch testing in reducing the number of test executions to deliver changes and find true failures, i.e. culprit commits. Based on the *FailureRate* we run simulations to determine the optimal *BatchSize* for three projects at Ericsson. Flaky test failures are tests that pass and fail on the same change. We factor test flakiness into our simulations as they increase the number of executions to test changes. The larger the *FlakeRate* the smaller the *BatchSize*. Although batch testing can help to reduce the test executions, unlike testing each change independently, when there is a failure a bisection must be done to find the likely true failing culprit commit. We introduce a novel technique where we guide bisection based on two risk models: a bug model and a test execution history model. We isolate the risky commits by testing them individually, while the less risky commits are tested in a single large batch. Our results show that batch testing in ideal environments with low test failure rates can reduce the test executions up to 72%. We also show that test flakiness will limit the savings to 42% as larger batches increase the probability of flakiness and hence extra executions. Moreover, we show that risk calculation approaches can be effectively

used to predict the culprit commits in a failing batch. Moreover, we show that culprit predictions can be used with our TestTopN approach to help to reduce the test executions up to 9% compared to our *FifoBisection* baseline. The results we present here have convinced Ericsson developers to implement our culprit risk predictions in the CulPred tool that will make their continuous integration pipeline more efficient.

4.2 Introduction

Software testing is one of the costliest stages of the software development process. Prior research estimates that testing consumes between 30% to 50% of the time in software development life cycle [PZTM13]. To isolate test failures, many companies have adopted the DevOps strategy of testing each individual commit. While effective at isolation there are substantial computation requirements. To limit the resource requirements, many software companies, including Ericsson, have adopted batching to reduce the cost of testing. Batch testing groups commits and allows all of them to be tested at once. When the batch passes, all of the commits can proceed in the continuous integration pipeline at once and save the resources.

Although batch testing can reduce test executions, it introduces a new problem. When a batch fails, the culprit commits causing the batch failure need to be identified. One of the common approaches used for finding a culprit in a group of failing commits is bisection. When commits are ordered, GitBisection [GBS] can use an ordered binary search to identify the culprit in log(n) time. However, GitBisection cannot be used in our studied subsystem in Ericsson as commits are not merged yet and have no order. In this regard, Ericsson has adopted an approach named bisection were batches are split in half and tested until the culprit is found.

In the first part of our work, we study the impact of batch testing on reducing the test executions in environments with various test failure rates. In practice, test failure rates in test environments tend to be very low. For example, on Chrome only 12.5% of tests fail [ZSR18]. Batch testing offers the highest savings in test environments with low failure rates.

In the second part we examine flaky tests, which can pass and fail on the same commit. Google reports that 1 in 7 tests are flaky and that 84% newly failing tests are actually flaky failures [Mic16]. Flaky tests are exacerbated by batching, as the batch size grows the probability that one or more commits will have a flaky test failure also grows.

In the last part of our work, we propose more efficient approaches for finding the culprits when a batch failure happens. First, we propose risk-based approaches for calculating risk values for the commits of a batch. Then we propose a TestTopN approach for testing the riskier commits individually and the rest of the lest risky commits together in a separate batch. We study how this approach can reduce the test executions compared to the bisection approach used in Ericsson. We propose two risk calculation approaches. The first approach is based on well-studied bug model literature [KSA$^+$13]. Our second approach is based on using test execution history and the file changes of the commits.

More specifically, we answer the following three research questions.

RQ1: What is the most cost-effective *BatchSize* for the number of culprits discovered during testing?

Batching commits for testing is more efficient with a low test failure rate, i.e. *CulpritRate*. The smaller the number of the test fails, the larger the *BatchSize* as there will be few failures. In contrast, the higher the *CulpritRate* the larger the number of bisections resulting in more executions.

In this work, we have studied the impact of batch size on the test executions and what an most cost-effective *BatchSize* should be. We have seen that in an ideal environment for projects that have lower *CulpritRate*, a higher batch size will lead to fewer executions.

The higher the *CulpritRate* the smaller the most cost-effective *BatchSize*. For example, Project A has a *CulpritRate* of over two times Project C and with a *BatchSize* of 4, the savings are 46%, while Project C can have a *BatchSize* up to 9 with savings of 72%.

RQ2: What is the most cost-effective *BatchSize* when some bisections are done as a result of flaky failures?

Test flakiness is an inevitable part of any test environment. Google reports that 1 in 7 tests are flaky and that 84% newly failing tests are actually flaky failures [Mic16]. A flaky test failure is defined as a test that passes and fails on the same commit. We study the impact of test flakiness in finding the most cost-effective *BatchSize*. Our study shows higher *FlakeRate* will limit the most cost-effective *BatchSize*. When commits are tested individually, a flaky failure does not affect other commits and the number of executions remains constant. In contrast, the larger the *BatchSize* the higher the probability that at least one of the commits in the batch will be flaky. Any flaky batch failure incurs the penalty of an unnecessary bisection.

The higher the *FlakeRate* the smaller the *BatchSize* and the smaller the savings in executions. For example, Project B has a 1/3 higher *FlakeRate* than Project C and a *BatchSize* of 4 saves 14% of the executions compared to 41%, respectively. With flaky failures, Project C's most cost-effective *BatchSize* and savings are reduced from a *BatchSize* of 9 and execution savings of 72% to 4 and 41%, respectively.

RQ3: Can risk models predict the culprit commit and reduce the number of executions to find the culprits on failing batches?

Batch testing is effective in reducing the test executions, however, introduces a new problem. When a batch fails, the root cause of the failure, i.e. culprit, needs to be found among the failed commits. We use commit risk models to predict the culprit commit when a batch fails. We use two types of models, Bug models, and historical test information.

Bug models have been effective at identifying the commits that are most likely to lead to future bugs, i.e. bug introducing changes. [HBB$^+$12, RHTZ13, TMH$^+$15, Aki71, DLR10, Has09, ZPZ07, MW00, MPS08, NZZ$^+$10, RPH$^+$11, KZJZ07, GFS05, LHSR06, MK92, CMRH09, MKAH14, Moc10, SJI$^+$10, SMK$^+$11, KSA$^+$13]. We use these techniques to identify which of the commits is the most likely culprit. We then test the riskiest commits individually and batch the remaining commits.

Our second approach is based on using test execution history. Test executions history has been largely studied for performing test selection and prioritization [KP02, ERP14, ASD14, ZSR18, CMBA17]. In contrast, we use test execution history to predict a culprit commit given a batch test failure. Particularly we use the relationship between file changes and test failures extracted from the test execution history. Campbell et al. [CMBA17] have proposed an approach for suggesting tests to run based on file changes. We reverse this idea by determining the most likely culprit given the failing test and the files under change. We found that our risk-based approaches can significantly reduce the number of test executions by predicting the culprit commits.

Both culprit risk prediction models are effective, but *TestExecutionHistory* outperforms *Bug-Model*. *TestExecutionHistory* is able to predict the culprits using the Top2 predictions are sufficient and correct 63% and 66% of the time for Projects B and C with *BatchSizes* = 4. Compared to *FifoBisection* this results in -9.0% and -7.6% fewer executions, respectively.

The results we present here have convinced Ericsson developers to implement our culprit risk predictions in the CulPred tool that will make their continuous integration pipeline more efficient.

This paper is structured as follows. In Section 4.3, we discuss the batching and bisection process used at Ericsson. In Section 4.4, we explain how we guide the bisection process to fewer executions. In Section 4.5. we introduce the theory behind bisection and introduce our commit risk models. In Section 4.6, we introduce our simulations methodology and data used in the study. In Sections 4.7, 4.8, 4.9, we present results for each of our research questions. In Section 4.10, we describe the threats to validity and how we mitigate them. In Section 4.11, we discuss related work to our study. In Section 4.12, we present our contributions and conclude the work.

4.3 Background on Batching and Bisection

To reduce test execution costs, instead of testing each new commit submitted by developers individually, commits can be collected in groups called batches and tested together. Ericsson uses this technique to reduce test executions as part of its Continuous Integration processes.

In this context, every batch is a group of one or more commits that require specific tests. As developers submit new changes, commits enter the test queue. The batching process consists of periodically collecting commits from the top of the queue and running their required tests. As tests are combined into a single build and tested together, batching can reduce test executions significantly.

Although batch testing can reduce test executions significantly, when a test fails on a batch, we need to isolate the commit that is causing the failure, i.e. the culprit commit. One approach to isolate the culprit commit is to run a binary search on the commits contained in a batch. This process is called Bisection at Ericsson and is run until the culprit is isolated.

The bisection process involves splitting the commits of the batch in half and is illustrated in Figure 6. This produces two new batches that are each half the size of the original batch. If the tests pass on a batch, we know the culprit is not among these commits. If a batch fails, we continue the bisection process. The stopping condition is when the remaining batches contain a single commit and the tests on that commit fail. A single commit with a failing test is the isolated culprit. The culprit will be subject to further investigations by testers or developers.

For example, in Figure 6, the process starts with Batch #1. This batch fails because it includes one culprit commit: Commit #102. The commits in Batch #1 are split into Batch #2 and Batch #3. Batch #2 passes because it includes no culprits. Therefore, all of its commits get delivered. Batch #3 however, fails because it includes the culprit commit. Batch #3 is split into Batch #4 and #5. Batch #5 passes. Batch # 4 however fails and because it consists of one commit it is the culprit.

Mathematically, we know that a binary search always requires $log_2(n)$ executions. However, as can be seen in the example, the tests must be run on both sides of the split binary tree. As a result, we must run $2 * log_2(n)$ executions. Since we must determine if the starting batch passes, we need an additional execution. With n commits, the number of executions required to find a single culprit is

$$2 * log_2(n) + 1 \tag{5}$$

4.4 Guiding Bisection based on Risk

Commits are batched and tested in the order in which they arrive (FIFO queue). However, some commits contain more risky changes than others [KSA+13]. Our goal is to model the risk and group changes such that risky commits are tested individually while less risky commits are grouped into batches that will likely pass without requiring bisection. In this section, we describe how we guide bisection by risk, in subsequent sections we show how we calculate the risk for our two models: *TestExecutionHistory* and *BugModel*.

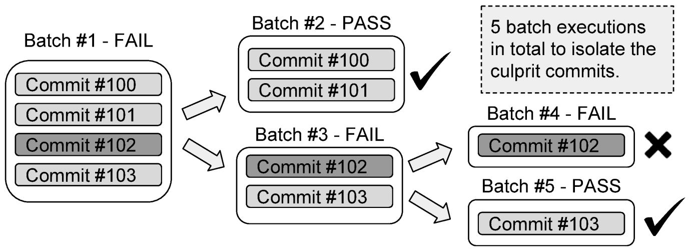

Figure 6: *FifoBisection*. When a batch of commits fails, a bisection is performed to find the culprit commit. Since an execution is required for each binary split, there are $2 * log_2(n) + 1$ executions required to find a culprit. To bisect 4 commits, we must run 5 executions. However, if the batch passes, we would need 1 execution to test the 4 commits.

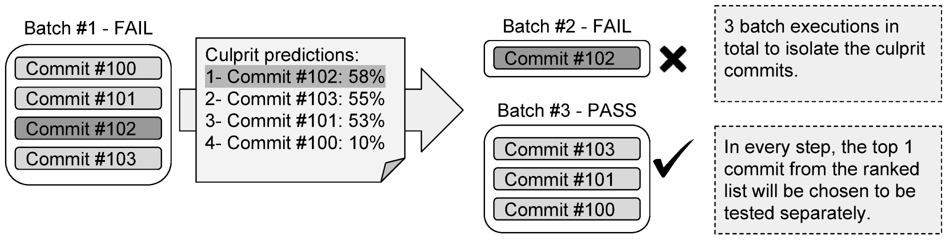

Figure 7: TestTopN. The riskiest N commits are tested individually, with the remaining commits combined in a single batch. In this case, top_1 reduces the number of required executions to three compared to the 5 in Figure 6.

The bisection process is inefficient because when any batch fails, there are at least $2 * log(n)$ additional executions, where n is the number of commits in the failing batch. We introduce the TestTopN approach to isolate the top N riskiest commits and test them individually while batching the remaining commits into a single large batch.

Figure 7 provides an illustration of top_1. The process starts with Batch #1, which fails because it includes a culprit. After the failure, we calculate risk values for the commits and test the top_n individually. For top_1 we individually test the riskiest commit. The remaining commits are tested as a single batch of size three. To find the culprit and deliver the commits, we need three executions instead of the five used for normal bisection (see Figure 6).

4.5 Culprit Risk Models

BugModels have been used to suggest risky files and changes [KSA$^+$13] that are more likely to contain future bugs. However, it has been difficult for developers to act upon these predictions as they do not indicate specific problems in the source code. *TestExecutionHistory* has been used to determine which tests should be run for a set of files as well as to prioritize tests in a queue [CMBA17], but has not been used to create batches of commits. In this work, we modify these approaches to assign risk to each of the changes under test and to determine which commit is most likely to be the true test failure, i.e. culprit. We guide the bisection processes by testing high risk commits individually.

4.5.1 *BugModel*

BugModels have a long history in the software engineering literature. Researches have predicted which files and modules are most likely to contain defects, i.e. which are riskiest[HBB$^+$12, RHTZ13, TMH$^+$15, Aki71, DLR10, Has09, ZPZ07, MW00, MPS08, NZZ$^+$10, RPH$^+$11, KZJZ07, GFS05, LHSR06, MK92, CMRH09, MKAH14, Moc10, SJI$^+$10, SMK$^+$11].

In contrast, Kamei et al. [KSA$^+$13] quantified the risk of a commit instead of an individual file or module. In this way, the authors were able to alert developers to the changes that may need additional review. However, the measures are difficult to act upon because they simply indicate that a change is "large" or that a developer has less experience, instead of indicating specific problems in the source code. Since our goal is to simply identify the riskiest commit and test to determine if it is the culprit, these models are sufficient.

Instead of training on the likelihood that a change will introduce a bug, we are interested in how likely a commit is to fail tests, i.e. is a culprit indicating a system fault. We train a logistic regression model to distinguish the commits that are most likely culprits, so our unit is the commit. We use many of the measures proposed by Kamei et al. [KSA$^+$13]. As each commit may have multiple file changes, if a measure is related to specific files, the average of the metric over all of the file changes of the commit is considered. The measures are explained below:

1. **Number of line changes**: Total number of lines deleted and inserted in the commit.
2. **Number of file changes**: Total number of file changes in the commit.
3. **Number of modified subsystems**: Kamei et al. [KSA+13] define a subsystem as the root directory of a file path in a project tree. For this metric, we simply count the number of file changes that have different root directories.
4. **Commit message:** A boolean value based on availability of "bug", "fix", or "defect" in the commit message [KSA+13].
5. **Developers:** Number of developers that were involved in change history of the changed files of the commit, averaged over the files.
6. **Experience:** Experience of the author of the commit on each of the changed files of the commit, averaged over the files.
7. **Change time interval:** Time interval between the current change and the previous change of each of the changed files of the commit, averaged over the files.

4.5.2 *TestExecutionHistory*

Previous work has shown that tests that have failed in the past are likely to continue failing [ASD14, KP02, ERP14, ZSR18]. Preliminary work at Ericsson has shown a relationship between failing tests and the files in the change under test [CMBA17]. While these works use test history to select and prioritize the tests that should be run for a change, we are the first to use this relationship to determine which change in a failing batch is the likely culprit. For each historical culprit, we record the tests that fail and the files that were changed, so that we can calculate how likely a test is to fail for a given file. The following process is used to calculate a culprit score for each commit in a batch.

1. Given the frequency of historical file and test failures, we calculate the probability that the culprit is related to a file and test as:

$$Prob(file_n, test_x) = \frac{\#file_n\_fails\_test_x}{\#total\_fails\_test_x} \tag{6}$$

2. We normalize this probability, by the number of lines changed in the file over the total lines changed in the commit:

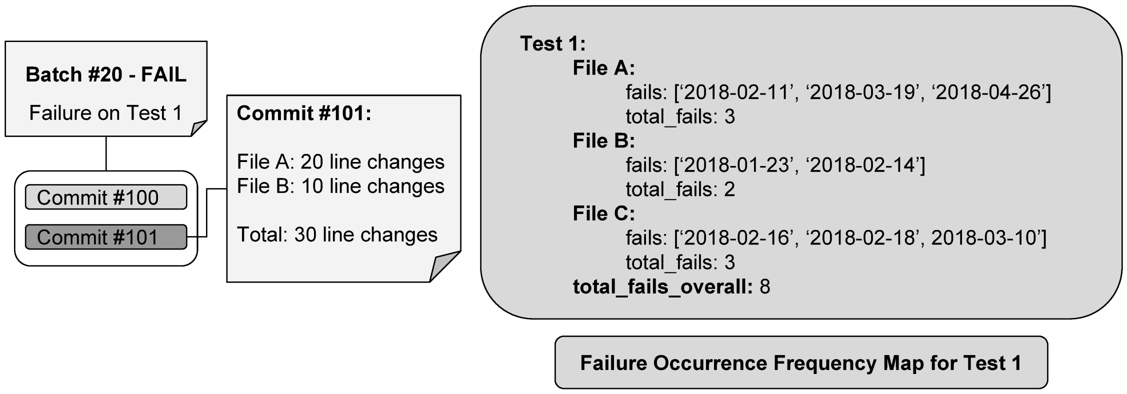

Figure 8: Calculating the commit culprit score based on file and test failure frequency.

$$Prob_{norm}(file_n, test_x) = \frac{\#line\_changes\_in\_file_n}{\#line\_changes\_in\_commit} * Prob(file_n, test_x) \tag{7}$$

3. We sum across all files in the commit to calculate the culprit score for the commit:

$$culprit\_score(commit) = \sum_{n=files, x=tests} Prob_{norm}(file_n, test_x) \tag{8}$$

Figure 8 shows an example. Let us assume Batch #20 is a new failure and we want to calculate culprit risk scores for its commits, Commit #100 and #101. Commit #101 has two file changes, File A and File B. To calculate the culprit probability for File B and Test 1, we see that in the past Test 1 has failed 8 times and 2 of those times File B was under change: $Prob(file_B, test_1) = 2/8$. As there are 30 lines changed in Commit #101 and File B has 10 line changes, $Prob_{norm}(file_B, test_1)$ is calculated as 2/8 * 10/30. We similarly calculate $Prob_{norm}(file_A, test_1)$ and sum the two values to get the final risk score for Commit #101.

4.6 Simulation Methodology and Data

We evaluate three projects at Ericsson which we name A, B, and C. The test practices and bisection techniques have been described in the background Section 4.3. Project A is the smallest with 1.3K commits. Project B is larger with 3.5K commits. Project C is the largest with 9.5K commits.

The goal of this work is to find the most cost-effective *BatchSize* given the *CulpritRate* and the *FlakeRate*. To perform a simulation, we need to know the *CulpritRate* and the *FlakeRate*. When a test fails on a commit and an issue with the software is discovered, we consider this commit to be the "root cause" or "culprit" for the test failure. During batching, commits are grouped together and bisection must be used to identify which commit is the cause or culprit of the failing batch. We define the *CulpritRate* to be the number of culprit commits divided by the total number of commits for the project.

$$CulpritRate = \frac{\#CulpritCommits}{\#TotalCommits} \tag{9}$$

The respective *CulpritRate* for projects A, B, and C is 6.8%, 8.6%, and 2.8% culprits. Project A and B have a similar *CulpritRate*, with Project B having slightly more culprits. Project A and B have slightly over twice as many culprits as Project C.

We must quantify the *FlakeRate* for our simulations because flaky test failures require additional bisections and executions. A flaky test failure is defined as a test that passes and fails on the same commit. We define a *FlakyBatch* as a batch that initially fails, but does not lead to an individual failing commit, i.e. no culprits are identified. We define the *FlakeRate* as the number of flaky batches divided by the total number of batches.

$$FlakeRate = \frac{\#FlakyBatches}{\#TotalBatches} \tag{10}$$

The respective *FlakeRates* for Projects A, B, and C are 23.6%, 23.1%, and 16.9%. Projects A and B have a similar *FlakeRate*, with Project A being slightly higher. Projects A and B have 1/3 more flaky failures than Project C.

4.6.1 The Impact of *BatchSize* on *FlakeRate*

We calculated the overall *FlakeRate* for all *BatchSizes*. However, the larger the *BatchSize* the higher the probability that at least one of the commits in the batch will be flaky. We do not know the *FlakeRate* for individual commits or tests. Instead, we know the size of the batch and whether or not it was a *FlakyBatch*. Any flaky batch failure incurs the penalty of unnecessary bisection and executions that must be accounted for in a simulation. We create a logistic regression model to determine the probability that a batch of size n will result in a flaky failure.

In Figure 9, we plot the logistic regression line indicating the probability of failure for batch sizes 1 to 20. Ericsson requested that we do not show the actual *FlakeRate* for batches, so the y-axis is unlabeled. However, it is clear that as the *BatchSize* grows, the probability that a batch is flaky increases dramatically.

Equation 11 shows how we correct for these flaky batches. We multiply the number of batches by the *FlakeRate* to give us the expected number of flaky batches. Each flaky batch requires an additional bisection, the cost in executions is defined in Equation 5.

$$\#FlakyExecutions = \#Batches * FlakeRate * 2 * \log_2(n) \tag{11}$$

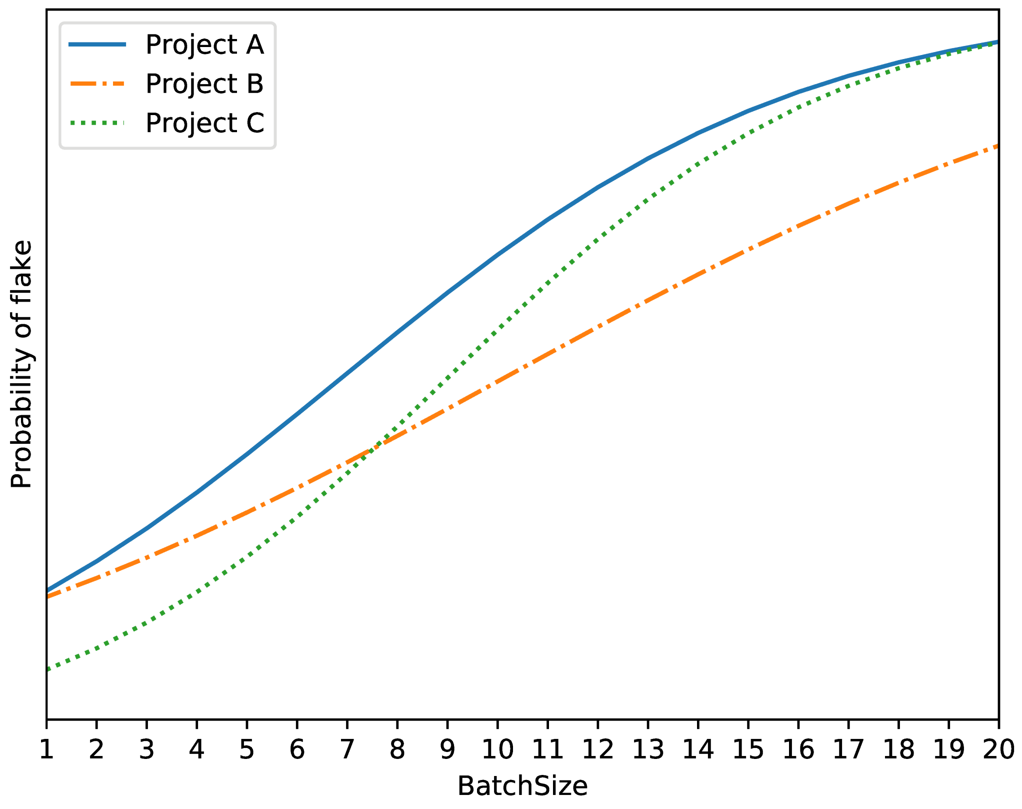

Figure 9: Probability of a flaky failure for each *BatchSize*. The probability is estimated with a logistic regression model for each project. At Ericsson request we anonymized the y-axis

We then add these additional FlakyExecutions to the executions required to find all the true culprits:

$$\#ExecutionsWithFlakes = \#CulrpitExecutions + \#FlakyExecutions \tag{12}$$

For example, for Project C, a *BatchSize* of 8 is about two times more likely to have a flaky failure than *BatchSize* of 4. Therefore, requires additional executions.

4.6.2 Simulation Methodology

Ericsson testers evaluate batch test failures on a daily basis. We run daily simulations using a simple incremental framework that has been commonly used in the research literature [HN15, JLYX17, XJRZ12, BN10].

Our simulation period runs for 9 months and covers thousands of commits which lead to hundreds of culprits. We use the first 3 months as an initial training period. After this period, we test the approaches on the commits that are available for the test each day, $t = 90$ to $t = 270$. To predict whether a failure on day $D = t$ will lead to a culprit, we train on the historical data from $D = 0$ to $D = t - 1$ and test on $D = t$. We repeat this training and testing cycle for each day until we reach $D = 270$.

We also run a simulation, using a sliding training window of three months. In this case, to predict whether a batch failure on day $D = t$ will lead to a culprit, we train on the historical data from $D = t - 90$ to $D = t - 1$ and test on $D = t$. We repeat this training and testing cycle for each day until we reach $D = 270$.

While we simulate batching on Ericsson data, our method and measures are not tied in any way to Ericsson data. To run this simulation on other projects, one simply needs the test outcomes for each change. The test outcome will allow one to calculate the *CulpritRate* and *FlakeRate*.

4.7 Result 1: *BatchSize* given *CulpritRate*

RQ1: What is the most cost-effective *BatchSize* for the number of culprits discovered during testing?

Batching commits for testing is more efficient with a low test failure rate, i.e. *CulpritRate*. The higher the *CulpritRate* the larger the number of bisections resulting in more executions. In the extreme case, where there are no test failures, all commits could be placed in a single massive batch requiring a single passing execution and saving n-1 executions, where n is the total number of commits.

In practice, the *CulpritRate* tends to be very low. For example, on Chrome 12.5% of tests fail [ZSR18]. Since the vast majority of tests do not fail, testing all commits individually wastes resources. Theoretically, the lower the *CulpritRate*, the higher the *BatchSize*.

The first step in finding most cost-effective *BatchSize* is quantifying how often tests and commits under test fail and lead to an investigation by developers. This matter relates to the *CulpritRate* of te projects. For confidentiality reasons, we cannot report the *CulpritRate* for the three Ericsson projects. However, projects A and B have higher culprit rates, at least two times higher than Project C. Project B has a slightly higher *CulpritRate* than Project A. With these variable *CulpritRates*, we simulate the savings relative to testing all commits individually, *TestAllCommits*, for *BatchSizes* 1 through 20.

Figure 10 shows the simulation results. The savings are substantial even for the smallest *BatchSize* = 2 commits. The figure shows that this batch size requires 34%, 34%, and 44% fewer executions for projects A, B, and C, respectively.

We see that the savings are logarithmic, with the majority of the savings occurring with *BatchSizes* up to 4. For Project C with the lowest *CulpritRate*, we note that the savings plateau with *BatchSizes* greater than 9 providing little additional savings. The maximum saving is 50%, 47%, and 74% for the projects respectively.

These savings and *BatchSizes* validate our conjecture. Project A and B have similar *CulpritRates* and see similar most cost-effective *BatchSizes* and savings in executions. Project B has a slightly higher *CulpritRate* than project A and also sees slightly less savings. Project A and B have two times more culprits than Project C. Project C has the highest *BatchSize* and the greatest savings.

The higher the *CulpritRate* the smaller the most cost-effective *BatchSize*. For example, Project A has a *CulpritRate* of over two times Project C and with a *BatchSize* of 4, the savings are 46%, while Project C can have a *BatchSize* up to 9 with savings of 72%.

4.8 Result 2: *BatchSize* given *FlakeRate*

RQ2: What is the most cost-effective *BatchSize* when some bisections are done as a result of flaky failures?

A flaky test failure is defined as a test that passes and fails on the same commit. We define a *FlakyBatch* as a batch that initially fails, but does not lead to an individual failing commit, i.e. no culprits are identified.

Flaky tests are a significant problem, with Google reporting that 1 in 7 tests are flaky and that 84% newly failing tests are actually flaky failures [Mic16]. When commits are tested individually, a flaky failure does not affect other commits and the number of executions remains constant. In contrast, the larger the *BatchSize* the higher the probability that at least one of the commits in the batch will be flaky. Any flaky batch failure incurs the penalty of an unnecessary bisection. As the *BatchSize* grows the probability of a flaky failure increases according to the models in Figure 9. The *FlakeRate* will limit the most cost-effective *BatchSize*.

In Section 4.6.1, we modeled the *FlakeRate* for batches of size 1 to 20 for each project. In this section, we adjust the simulation for the varying *FlakeRate* of our studied Ericsson projects. While we cannot report the actual *FlakeRate* at Ericsson, we note that projects A and B have 1/3 more flaky failures than Project C. Project A has a slightly higher *FlakeRate* than Project B.

Figure 11 shows the simulation results after correction for the *FlakeRate* of the projects for different batch sizes. At *BatchSize* = 2 we see a reduction in executions of 7%, 9%, and 30% respectively for projects A, B, and C. We see that the savings in executions are logarithmic up to *BatchSize* 2, 4, and 4, respectively. After *BatchSize* = 4 we see a decrease in the savings with an

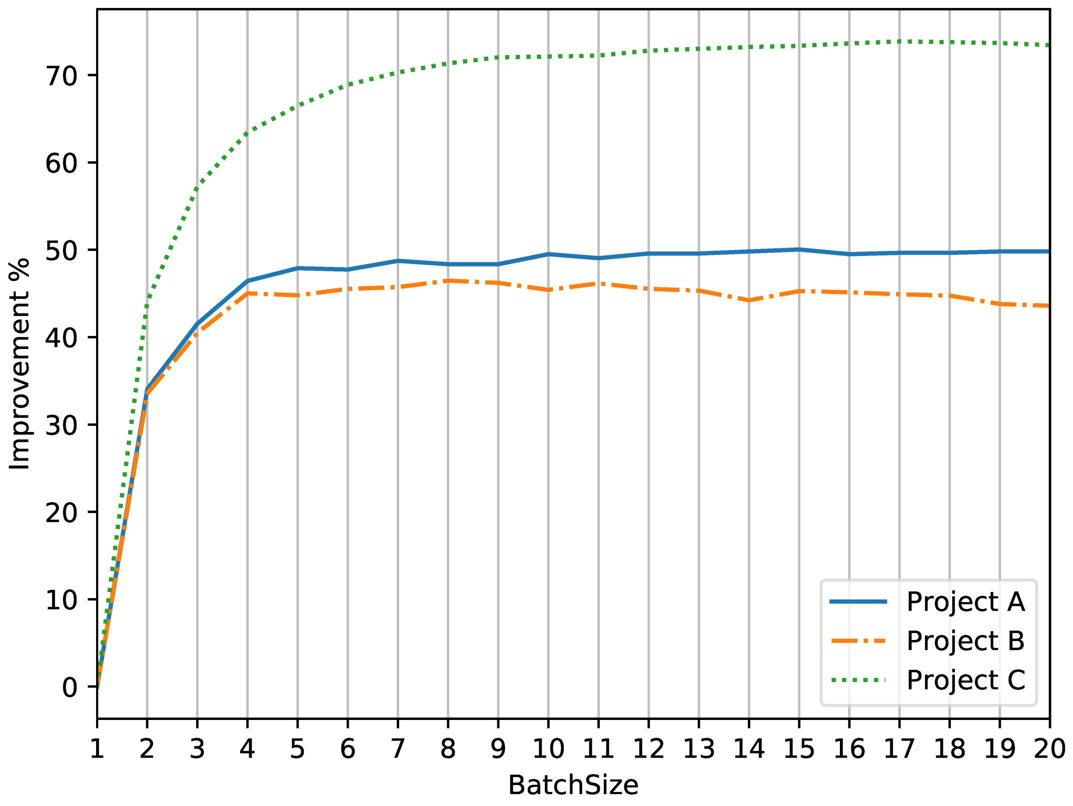

Figure 10: Improvement in test executions for different *BatchSizes*. In an ideal environment, we see a logarithmic increase with most of the savings in executions being realized before batches of size four.

increase in the number of executions as flaky failures become more frequent in larger batch sizes.

The maximum saving is 7%, 14%, and 41% for projects A, B, and C at *BatchSize* = 4 for projects B and C and at *BatchSize* = 2 for project A.

These savings and *FlakeRates* validate our conjecture. Acknowledging flaky failures reduces the most cost-effective *BatchSize*. Project A has the highest *FlakeRate* and the smallest most cost-effective *BatchSize* of 2 and lowest savings of 7%. Project B has slightly lower *FlakeRate* and has more commits than Project A, its most cost-effective *BatchSize* is 4 with savings of 14%. Projects A and B have a *FlakeRate* at least 1/3 larger than Project C. Project C has an most cost-effective *BatchSize* of 4 with savings of 41%.

Without considering the *FlakeRate*, Project C had a *BatchSize* of 9 and a savings of 72%. However, in Figure 11, we can see by a *BatchSize* of 4, Project C already saves 64%. Creating larger batches of commits leads to a higher probability that any one of them will be a flaky failure requiring additional wasted executions. The trade-off between savings and additional executions is optimized at a *BatchSize* of 4 for Project C. Clearly the *FlakeRate* must be taken into account when performing simulations to find the most cost-effective *BatchSize* for a software project.

The higher the *FlakeRate* the smaller the *BatchSize* and smaller the savings in executions. For example, Project B has a 1/3 higher *FlakeRate* than Project C and a *BatchSize* of 4 saves 14% of the executions compared to 41%, respectively. With flaky failures, Project C's most cost-effective *BatchSize* and savings are reduced from a *BatchSize* of 9 and execution savings of 72% to 4 and 41%, respectively.

4.9 Result 3: Risk Models to Predict Culprit Commits

RQ3: Can risk models predict the culprit commit and reduce the number of executions to find the culprits on failing batches?

In this section we use *BugModels* and *TestExecutionHistory* models to predict which commit in a batch is the true culprit. Our goal is to reduce the number of executions to find the culprit by testing high-risk commits in isolation. We isolate the top K riskiest commits and test these individually while combining the remaining less risky commits in a single large batch. In the background on bisection in Section 4.3, we use Figure 7 to illustrate how the riskiest commit, Top1, is tested in isolation, while the remaining 3 commits are tested in a batch. However, if the risk prediction is incorrect, we would need a maximum of 7 executions to find the culprit. In contrast, Figure 6 shows a bisection of a failing batch will always require 5 executions to find a single culprit. An accurate risk model will reduce the number of executions, while an inaccurate model can even increase the

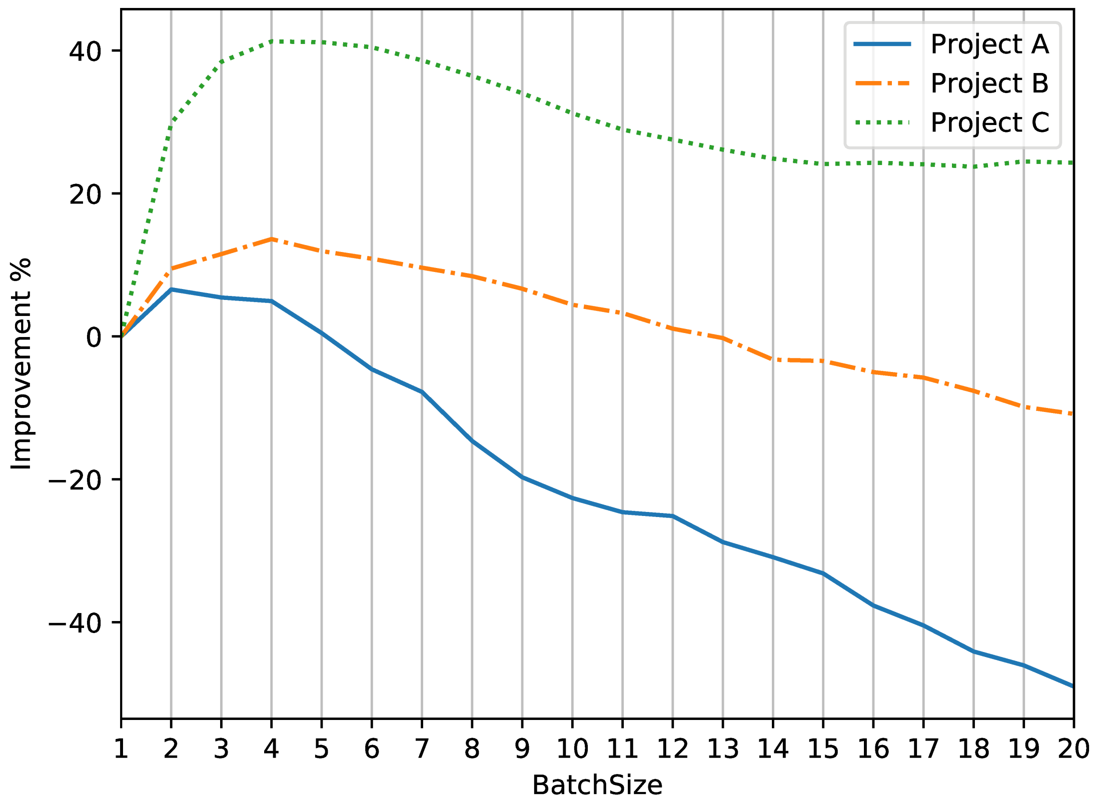

Figure 11: Improvements in test executions considering the *FlakeRates*. The *FlakeRate* controls the *BatchSize* and the project with the highest flake rate does not see any advantage above *BatchSize* = 2. Project C still attains high execution savings, at 41% with a *BatchSize* of 4.

number of executions to find culprits.

We evaluate the *BugModels* and *TestExecutionHistory* models on two evaluation measures: SufficientAndCorrectAtK, and PercentExecutionDifferenceWithBisection.

SufficientAndCorrectAtK determines how many of the total culprits in each batch are correctly predicted in the TopK suggested commits of the algorithm.

$$SufficientAndCorrectAtK = \frac{NumCorrectCulpritPredictionsAtK}{TotalCulprits} \tag{13}$$

For example, a batch with two culprits using $K = 1$ has a maximum SufficientAndCorrectAt1 of 1/2 or 50%, as no single prediction can find two culprits. In contrast, the maximum SufficientAndCorrectAt2 is 2/2 or 100%.

Our ultimate goal is to reduce the number of total executions. We calculate the difference in the number of executions for the risk models relative to the current process at Ericsson.

$$PercentExecutionDifferenceWithBisection = 1 - \frac{NumExecutionsBisection}{NumExecRisk} \tag{14}$$

A negative percent difference indicates a saving in executions when compared with *FifoBisection*, while a positive percentage indicates that the risk-based approach does not outperform *FifoBisection* and requires more executions.

***BatchSize* for Culprit Prediction.** In the previous section, we found that Project A has an most cost-effective *BatchSize* of 2, which means that when there is a test failure, there will always be two executions regardless of commit risk, i.e. both commits need to be tested individually. We exclude Project A from this analysis. In contrast, we found that the optimal *BatchSize* for bisection for Projects B and C is four commits. We use $BatchSize = 4$ to evaluate our risk-based algorithms. We also experimented with $BatchSize = 1 \ldots 16$ and found that size 4 produced the best result.

We evaluate Top1 and Top2 only because with a $BatchSize = 4$, TestTop3, TestTop4, and TestAll are equivalent requiring all commits to be tested individually. For example, if we test the Top 3 ranked commits in isolation the remaining batch has only one commit, so all commits are tested effectively in isolation, which is equivalent to TestAll.

Results for Culprit Risk Prediction

The results of our analysis are shown in Table 2 [1] [2]. For **TestTop1**, the top-ranked commit will be tested in isolation, while the remaining three commits will be tested as a batch. If the prediction is incorrect, we re-run the process on the next highly ranked commit. The *BugModel* with TestTop1 has SufficientAndCorrectAt1 of 22% and 34% for Projects B and C respectively. However, it requires

[1]Project A most cost-effective *BatchSize* is 2, so both must be tested regardless of risk.
[2]Considering how the TestTopN approach works we can see Top3 = Top4 = TestAll.

Table 2: Results of risk-based approaches

	$TopK = 1$		$TopK = 2$	
Project B	SufficientAndCorrect	Difference in Executions	SufficientAndCorrect	Difference in Executions
BugModel	22%	5.0%	59%	-7.4%
TestFile	33%	0.7%	**63%**	**-9.0%**
Project C	SufficientAndCorrect	Difference in Executions	SufficientAndCorrect	Difference in Executions
BugModel	34%	2.6%	59%	-4.3%
TestFile	46%	-5.0%	**66%**	**-7.6%**

5.0% and 2.6% more total executions than *FifoBisection*, for Projects B and C. *TestExecutionHistory* with TestTop1 has a SufficientAndCorrectAt1 of 33% and 46%, for Projects B and C respectively.

For Project B *TestExecutionHistory* with TestTop1 requires 0.7% more executions. However, for Project C we see fewer total executions are needed, -5.3%, when compared to *FifoBisection*.

When there is more than one culprit, a model that only predicts one culprit, i.e. TestTop1, will not be able to find all culprits and will require additional executions. Project B has batches with two or more culprits 25% of the time and clearly requires at least TestTop2. In contrast, Project C has two or more culprits only 6% of the time. The *BugModel*'s predictions are not accurate enough at TestTop1 and require more executions than *FifoBisection* due to these inaccurate predictions. In contrast, *TestExecutionHistory*'s Top1 prediction is accurate enough to reduce the number of execution, -5.0%, for Project C.

For **TestTop2**, the commits ranked 1 and 2 by the commit risk model are tested individually, while the other commits are tested in a single batch. The *BugModel* with TestTop2 has a RecallAt2 of 59% for both Project B and C. The corresponding values for *TestExecutionHistory* are 63% and 66%. *TestExecutionHistory* with Top2 is the most effective technique improving on the *BugModel* by 4 and 7 percentage points for the projects respectively. Both commit risk models are more effective than *FifoBisection* for Project B and C with -7.4% and -4.3% executions for *BugModel* and -9.0% and -7.6% executions for *TestExecutionHistory*, respectively. The model accuracies at Top2 are sufficient to reduce the number of executions when compared with *FifoBisection*.

Both culprit risk prediction models are effective, but *TestExecutionHistory* outperforms *BugModel*. *TestExecutionHistory* is able to predict the culprits using the Top2 predictions with a SufficientAndCorrectAt2 of 63% and 66% for projects B and C with $BatchSizes = 4$. Compared to *FifoBisection* this results in -9.0% and -7.6% fewer executions, respectively.

4.10 Threats to Validity

Our study only considers three projects in the software development environment of Ericsson. Although we believe that these projects can be good representatives of generic projects in industry, our results may not generalize to other projects. They also have varying size, Project A is the smallest, and variations in *FailureRate* and *FlakeRate*, Project A has twice as many culprits as C and Project B has 1/3 more flaky failures. Our methodology and simulation only requires the test outcomes on commits and can easily be applied to other projects to determine the most cost-effective *BatchSize* for a project.

Our simulations include simplifications of some of the Ericsson processes and may not exactly match the reality of the development environment of Ericsson. In order to verify our results, we suggest practitioners implement our approaches in production workflows and evaluate the results in real environments after determining the most cost-effective *BatchSize*.

Notes on other experimental conditions. As explained in Section 4.6.2, we have experimented two training models: a sliding window training model and also using all previous data. Our results show that the savings using a sliding window training model is slightly lower than using all previous data. Particularly, for our best approaches, i.e. TestTop2 *BugModel* and *TestExecutionHistory*, savings are -8.2% and -5.2% instead of -9.0% and -7.4% respectively for Project B and -7.0% and -3.6% instead of -7.6% and -4.3% respectively for Project C. Hence, the diagrams and distributions explained in this section are based on using all previous data at each iteration day. This parameter can be easily changed based on the results attained for other projects.

Moreover, our experiments show that some of our extracted features for creating bug models lead to deterioration of the results. Notably, features regarding average developer experience, number of file changes and average time interval among file changes have been excluded from because the reduced the quality of the culprit risk predictions.

Finally, we assume that a *FlakyBatch* will result in the same number of executions as is required to find a single culprit, i.e. $2 * log_2(n)$. However, given that the test is flaky, all tests may pass on the first split in the bisection. In the case of batches of size 8, finding a single culprit requires 6 executions. However, if all commits pass after the first split, there are only 2 additional executions required. Our approach is conservative when treating bisections adding the number of executions required to find a culprit even if one does not exist, i.e. a flaky batch.

Different combinations of commits in a batch can be another source of test flakiness. For example, when commit a and commit b are tested together they are flaky. However, when combined with commit c they are not flaky anymore. We have not taken into account the impact of combining different commits together on test flakiness.

4.11 Related Work

There are two reasons why commits are grouped: test efficiency and integration. When resources are scarce, e.g. expensive specialized test hardware, individual commits must be grouped together as there are not enough resources to test each commit individually. While unit tests can determine that each individual commit is working, we must test to ensure that when the changes are combined, i.e. integrated, there are no new faults. Regardless of the reason for a batch, once it fails the commit or commits that are causing the problem must be identified and fixed, i.e. the root cause or culprit must be found [Rei09].

One of the common approaches used for finding a culprit in a group of failing commits is bisection. When commits are ordered, GitBisection [GBS] can use an ordered binary search to identify the culprit in log(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) [ZR17].

At Ericsson, GitBisection is not possible because the commits are combined into an unordered group of changes. A bisection on an unordered set of commits is more expensive than an ordered GitBisection requiring $2 * log_2(n) + 1$ executions (see Section 4.3). Furthermore, at Ericsson, it is complicated to extract the static build dependency graph of which tests will be run. In our work, we evaluate the optimal batch size given the *FailureRate* and the *FlakeRate*. We then guide bisection by using historical models, instead of statical dependencies.

Test Selection

The goal of test selection is to choose the most appropriate tests to be run for a given change. In our work, we use these ideas to determine which change is the most likely culprit given the tests that have failed.

Early work on test selection used static analysis and code coverage [NH17, SYGM15, WNT17, NABL, KSK, KR15, SS14, JKS12, KHT13, NTV$^+$14, QCR08, LLH$^+$16]. The Google culprit finder [ZR17] uses similar information in the form of build test file dependencies to select the most likely culprit.

In contrast, our work builds on the history of test failures. To prioritize tests, previous works have used the recency of the test failures to determine which test is most likely to fail [KP02]. Building on these ideas, researchers have developed sliding windowed predictions [ERP14], used association rule mining [ASD14], and test co-failure distributions [ZSR18]. These only consider the tests and do not consider the unit under test. In contrast, a preliminary work at Ericsson determined which tests to run on the basis of which tests have failed with commits containing similar files [CMBA17]. We reverse this idea and instead of predicting which tests to run given the files in a change, we determine which is the most likely culprit given the failing test and the files under change.

Through test selection, Elbaum et al [ERP14] report a savings of 70% to 80% of executions. In our work, we run all the tests and are able to save 50% to 74% of executions. When flaky test failures are accounted for, the results show savings of 7% to 41% compared to our testing all of the commits as our baseline.

Anderson et al [ASD14] use association rule mining to predict the tests that are most likely to fail. They report a precision of up to 46.3 and recall of up to 76.6. Their goal is to select tests to run for a change, while in our work we select the commit that is most likely to have a failing test. Our average SufficientAndCorrectAt2 which is the same as RecallAt2 is 64.5%, and although has a different purpose is comparable to other results.

BugModels

Recent works have extensive studied bug predictions and bug models. The focus of earlier work has been on predicting defective software modules or evaluating the impact of different software metrics related to that [HBB+12, RHTZ13, TMH+15, Aki71, DLR10, Has09, ZPZ07, MW00, MPS08, NZZ+10, RPH+11, KZJZ07, GFS05, LHSR06, MK92, CMRH09, MKAH14, Moc10, SJI+10, SMK+11].

However, other studies focus on predicting defects on the change level. Predicting bugs on the change level makes it easier for developer address the issues and act on the predictions. For example, Kim et al. [KJZ08] propose an approach for classifying the developer changes as buggy or clean. They extract features like the lines modified in each change, author and time of the change, complexity metrics and etc. from software revision history and train a Support Vector Machine classifier to predict the changes as buggy or clean. This approach also examines the risk associated with each submitted change without connecting them to a concrete fault localization context of a test failure. Our approach, on the other hand, does so using an empirical approach that points to the culprit change that is involved in a test failure.

Kamei et al. [KSA+13] propose a risk analysis approach in change level. They construct a logistic regression model for analyzing changes using different factors under six high-level categories of diffusion, size, purpose, history, and developer experience to calculate the risk values. The number of modified subsystems, lines of code added, the average time interval between the last and the current change, and recent developer experience are among the utilized metrics. We adopt this study as one of our risk prediction approaches.

Yang et al. [YLX+15] propose a deep learning based technique for predicting the faulty changes. They use an advanced deep learning algorithm named Deep Belief Network for extracting a set features for measuring the changes. Then they train a logistic regression classifier for predicting the risk values of the changes. Similar to [KSA+13], this approach also just predicts the risks associated with different changes but does not associate them with any concrete test failure. Our approach, however, starts from a concrete test failure and attempts to locate the change associated with that test failure. Yang et al. [YLXS17] propose another similar study for predicting defective changes using a two-layer ensemble learning approach. Young et al. [YAB18] propose a replication of this study.

What all these studies have in common is predicting the buggy commits as early as possible, a process called just-in-time defect prediction. A problem with these bug models is that there is no concrete evidence that a suggested change is actually problematic and needs to be investigated as soon as possible. Our study, however, focuses on predicting culprit commits. A culprit commit is one of the multiple changes that have actually failed a test and needs to be found and addressed right away.

Kamei et al. [KSA+13] report an average precision of 37% and recall of 67%. Kim et al. [KJZ08] report an average precision of 61% and recall of 62% for the same projects. Yang et al. [YLX+15] use deep learning on the same projects and achieve an average precision of 35% and a recall of 69%. Our *BugModel* achieves an average SufficientAndCorrectAt2 (RecallAt2) of 59%. While our recall is comparable to previous works, our goal is to find the culprit that is causing a test failure instead of the potential introduction of a bug in a commit. By incorporating historical test information in the *TestExecutionHistory* model, we attain an average SufficientAndCorrectAt2 (RecallAt2) of 64.5%, which outperforms the *BugModel*.

4.12 Conclusion

The resources required for testing large-scale modern software systems has grown dramatically. Each change must be tested and integrated. To save resources, commits are grouped into batches for testing. We are the first work to study to examine the most cost-effective *BatchSize* based on the number of true test failures, *FailureRate*. Flaky tests are a known problem on all large systems, we factor *FlakeRate* into our simulations. The *FlakeRate* is more damaging with large batch sizes as the number of commits in a batch grows so does the probability of the batch failing due to a flaky test. We also use risk prediction models to more quickly isolate commits that are the likely culprits using *BugModels* and *TestExecutionHistory* models.

We make three major contributions:

1. We find the higher the *FailureRate* the smaller the most cost-effective *BatchSize*. We see a logarithmic increase with most of the savings in executions being realized before batches of size four. Our results show that Project C, with the lowest *FailureRate*, can optimally use *BatchSize* = 9 and have savings above 72% of executions.

2. We model the *FlakeRate*. With moderate levels of flakiness, the savings seen above a *BatchSize* = 4 do not outweigh the additional executions required to identify a flaky failure. The *FlakeRate* controls the *BatchSize* and the project with the highest flake rate does not see any advantage above *BatchSize* = 2. Project C still attains high executions savings, at 41% with a *BatchSize* of 4.

3. Using risk predictions from *BugModels* and *TestExecutionHistory* models, we are able to rank the commits by how likely they are to contain the culprit. We find that the *TestExecution-History* model achieves an average SufficientAndCorrectAt2 of 64.5% and outperforms the *BugModel*. By using these risk predictions compared to *FifoBisection* we need fewer executions, between -9.0% and -7.6%.

Our work opens a new area of research into culprit finding and prediction. While we have examined preliminary *BugModels* and modified the work on test selection to identify potential culprits in the *TestExecutionHistory* model, we feel that there is much further work to be accomplished. The results we present here have convinced Ericsson developers to implement our culprit risk predictions in the CulPred tool that will make their continuous integration pipeline more efficient.

Chapter 5

Conclusions

In this thesis, we proposed approaches for reducing test executions and improving test effectiveness in large-scale test environments. Software testing is one of the costliest stages of software development life cycle. In Chapter 3, we proposed our adopted approaches for test selection and test prioritization approaches in order to reduce the test executions in Ericsson. Our experiments revealed that test prioritization approaches are generally more efficient than test selection approaches in improving test processes. Moreover, it was shown that due to the non-invasive nature of test prioritization approaches, they have the least amount of impact on the final product quality. We observed that test prioritization using simple frequency analysis of the test failures was the most preferable approach in improving test effectiveness.

The major contributions of Chapter 3 are:

- We adopt and evaluate test selection and prioritization approaches with the goal of improving test effectiveness in a large industrial system with a complex testing infrastructure.
- We demonstrate the value of test execution history for improving test effectiveness in a large-scale industrial system in practice.
- We provide an industrial experience report that documents the challenges that are encountered and our lessons learned during the adoption process of the test selection and prioritization approaches.

The resources required for testing large-scale modern software systems has grown dramatically. Each change must be tested and integrated. As an approach to save the resources, commits can be grouped into batches for testing. In Chapter 4, we experimented batch testing and its impact on test executions and test flakiness. We are the first work to study to examine the most cost-effective *BatchSize* based on the number of true test failures, *FailureRate*.

We observed that higher batch sizes can reduce executions, however, also lead to higher test flakiness. Therefore, we proposed a methodology on how to find the optimal batch size for each test environment. Moreover, we proposed approaches for how to isolate a culprit commit when a new batch failure happens. Our results show that our *BugModel* and *TestExecutionHistory* approach can effectively reduce the test executions by predicting the buggy commits and testing them individually.

We make three major contributions in Chapter 4:

1. We find the higher the *FailureRate* the smaller the most cost-effective *BatchSize*. Our results show that test executions can be reduced up to 72% with our dataset with the lowest *FailureRate*.

2. We model the *FlakeRate* and its impact on most cost-effective *BatchSize*. The *FlakeRate* controls the *BatchSize*. We observe that test flakes can limit the savings.

3. Using risk predictions from *BugModels* and *TestExecutionHistory* models, we are able to rank the commits by how likely they are to contain the culprit. We find that our risk models on average are sufficient and correct in up to 64.5% of the time. Furthermore, reduce the test executions up to -9.0% compared to *FifoBisection* approach.

Our work opens a new area of research into culprit finding and prediction. While we have examined preliminary *BugModels* and modified the work on test selection to identify potential culprits in the *TestExecutionHistory* model, we feel that there is much further work to be accomplished. The results we present here have convinced Ericsson developers to implement our culprit risk predictions in the CulPred tool that will make their continuous integration pipeline more efficient.

Bibliography

[AAPV09] L. C. Ascari, L. Y. Araki, A. R. T. Pozo, and S. R. Vergilio. Exploring machine learning techniques for fault localization. In *2009 10th Latin American Test Workshop*, pages 1–6, March 2009.

[AFMS95] D. Abramson, I. Foster, J. Michalakes, and R. Sosic. Relative debugging and its application to the development of large numerical models. In *Proceedings of the IEEE/ACM SC95 Conference*, pages 51–51, 1995.

[AHLW95] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 143–151, Oct 1995.

[Aki71] Fumio Akiyama. An example of software system debugging. In *IFIP Congress (1)*, volume 71, pages 353–359, 1971.

[AMS$^+$96] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, A Inkeri Verkamo, et al. Fast discovery of association rules. *Advances in knowledge discovery and data mining*, 12(1):307–328, 1996.

[ASD14] Jeff Anderson, Saeed Salem, and Hyunsook Do. Improving the effectiveness of test suite through mining historical data. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 142–151, New York, NY, USA, 2014. ACM.

[ASD15] J. Anderson, S. Salem, and H. Do. Striving for failure: An industrial case study about test failure prediction. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 49–58, May 2015.

[AVG09] Rui Abreu and Arjan JC Van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In *SARA*, volume 9, pages 2–9, 2009.

[Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[BLL07] L. C. Briand, Y. Labiche, and X. Liu. Using machine learning to support debugging with tarantula. In *The 18th IEEE International Symposium on Software Reliability (ISSRE '07)*, pages 137–146, Nov 2007.

[BN10] P. Bhattacharya and I. Neamtiu. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, Sept 2010.

[BPH10a] G. K. Baah, A. Podgurski, and M. J. Harrold. The probabilistic program dependence graph and its application to fault diagnosis. *IEEE Transactions on Software Engineering*, 36(4):528–545, July 2010.

[BPH10b] George K. Baah, Andy Podgurski, and Mary Jean Harrold. Causal inference for statistical fault localization. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 73–84, New York, NY, USA, 2010. ACM.

[CDFR08] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Formal concept analysis enhances fault localization in software. In Raoul Medina and Sergei Obiedkov, editors, *Formal Concept Analysis*, pages 273–288, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[CDFR11] Peggy Cellier, Mireille Ducasse, Sébastien Ferré, and Olivier Ridoux. Multiple fault localization with data mining. In *SEKE 2011 - Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering*, pages 238–243, 07 2011.

[CMBA17] M. Campbell, K. Martin, F. Bozóki, and M. Atkinson. Dynamic test selection using source code changes. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 597–598, July 2017.

[CMRH09] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35(6):864–878, Nov 2009.

[DLR10] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 31–41, May 2010.

[DMG07] Paul Duvall, Stephen M. Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007.

[dSPdAB14a] L. S. d. Souza, R. B. C. Prudêncio, and F. d. A. Barros. A hybrid binary multi-objective particle swarm optimization with local search for test case selection. In *2014 Brazilian Conference on Intelligent Systems*, pages 414–419, Oct 2014.

[dSPdAB14b] L. S. de Souza, R. B. C. Prudêncio, and F. d. A. Barros. A comparison study of binary multi-objective particle swarm optimization approaches for test case selection. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 2164–2171, July 2014.

[ERP14] Sebastian Elbaum, Gregg Rothermel, and John 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.

[FCB$^+$14] E. Fourneret, J. Cantenot, F. Bouquet, B. Legeard, and J. Botella. Setgam: Generalized technique for regression testing based on uml/ocl models. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pages 147–156, June 2014.

[GBS] bisect - https://git-scm.com/docs/gitbisect.

[GEM15] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 211–222, New York, NY, USA, 2015. ACM.

[GFS05] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, Oct 2005.

[GHZG05] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 263–272, New York, NY, USA, 2005. ACM.

[Has09] A. E. Hassan. Predicting faults using the complexity of code changes. In *2009 IEEE 31st International Conference on Software Engineering*, pages 78–88, May 2009.

[HBB$^+$12] 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, Nov 2012.

[HGCM15] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. The art of testing less without sacrificing quality. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 483–493, Piscataway, NJ, USA, 2015. IEEE Press.

[HN15] K. Herzig and N. Nagappan. Empirically detecting false test alarms using association rules. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 39–48, May 2015.

[HRB88] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

[JGG08] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. Fault localization using value replacement. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 167–178, New York, NY, USA, 2008. ACM.

[JHS01] James A Jones, Mary Jean Harrold, and John T Stasko. Visualization for fault localization. In *in Proceedings of ICSE 2001 Workshop on Software Visualization*. Citeseer, 2001.

[JHS02] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM.

[JKS12] 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, Nov 2012.

[JLYX17] He Jiang, Xiaochen Li, Zijiang Yang, and Jifeng Xuan. What causes my test alarm?: Automatic cause analysis for test alarms in system and integration testing. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 712–723, Piscataway, NJ, USA, 2017. IEEE Press.

[JXxC$^+$04] Wu Ji, Jia Xiao-xia, Liu Chang, Yang Hai-yan, Liu Chao, and Jin Mao-zhong. A statistical model to locate faults at input level. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ASE '04, pages 274–277, Washington, DC, USA, 2004. IEEE Computer Society.

[KC15] H. Kumar and N. Chauhan. A coupling effect based test case prioritization technique. In *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 1341–1345, March 2015.

[KG10] Negar Koochakzadeh and Vahid Garousi. A tester-assisted methodology for test redundancy detection. *Adv. Soft. Eng.*, 2010:6:1–6:13, January 2010.

[KGM09] N. Koochakzadeh, V. Garousi, and F. Maurer. Test redundancy measurement based on coverage information: Evaluations and lessons learned. In *2009 International Conference on Software Testing Verification and Validation*, pages 220–229, April 2009.

[KHT13] N. Kukreja, W. G. J. Halfond, and M. Tambe. Randomizing regression tests using game theory. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 616–621, Nov 2013.

[KJMG17] Rafaqut Kazmi, Dayang N. A. Jawawi, Radziah Mohamad, and Imran Ghani. Effective regression test case selection: A systematic literature review. *ACM Comput. Surv.*, 50(2):29:1–29:32, May 2017.

[KJZ08] S. Kim, E. J. Whitehead Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, March 2008.

[KL88] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155 – 163, 1988.

[KP02] Jung-Min Kim and Adam 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.

[KP15] P. Klindee and N. Prompoon. Test cases prioritization for software regression testing using analytic hierarchy process. In *2015 12th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 168–173, July 2015.

[KR15] 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, June 2015.

[KSA$^+$13] 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, June 2013.

[KSK] Manoj Kumar, Arun Sharma, and Rajesh 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.

[KZJZ07] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *29th International Conference on Software Engineering (ICSE'07)*, pages 489–498, May 2007.

[KZPW06] S. Kim, T. Zimmermann, K. Pan, and E. J. Jr. Whitehead. Automatic identification of bug-introducing changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 81–90, Sept 2006.

[LFY$^+$06] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, 32(10):831–848, Oct 2006.

[LHSR06] Paul Luo Li, James D. Herbsleb, Mary Shaw, and Brian Robinson. Experiences and results from initiating field defect prediction and product test prioritization efforts at abb inc. In *ICSE*, 2006.

[LIH17] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. Measuring the cost of regression testing in practice: A study of java projects using continuous integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 821–830, New York, NY, USA, 2017. ACM.

[LLH$^+$16] Mohsen Laali, Huai Liu, Margaret Hamilton, Maria Spichkova, and Heinz W. Schmidt. Test case prioritization using online fault detection information. In Marko Bertogna, Luis Miguel Pinho, and Eduardo Quiñones, editors, *Reliable Software Technologies – Ada-Europe 2016*, pages 78–93, Cham, 2016. Springer International Publishing.

[LNZ+05] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 15–26, New York, NY, USA, 2005. ACM.

[MaBMM16] Cláudio Magalhães, Flávia Barros, Alexandre Mota, and Eliot Maia. Automatic selection of test cases for regression testing. In *Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing*, SAST, pages 8:1–8:8, New York, NY, USA, 2016. ACM.

[MGS13] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. Test case prioritization for continuous regression testing: An industrial case study. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 540–543, Washington, DC, USA, 2013. IEEE Computer Society.

[Mic16] John Micco. Flaky tests at google and how we mitigate them. `https://goo.gl/rxFGiw`, 2016.

[MK92] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5):423–433, May 1992.

[MK09] Thilo Mende and Rainer Koschke. Revisiting the evaluation of defect prediction models. In *PROMISE*, 2009.

[MKAH14] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 192–201, New York, NY, USA, 2014. ACM.

[MMK04] D. P. Mohapatra, R. Mall, and R. Kumar. An edge marking technique for dynamic slicing of object-oriented programs. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, pages 60–65 vol.1, Sept 2004.

[Moc10] Audris Mockus. Organizational volatility and its effects on software defects. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 117–126, New York, NY, USA, 2010. ACM.

[MPS08] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 181–190, May 2008.

[MRA13] Varun Modi, Subhajit Roy, and Sanjeev K. Aggarwal. Exploring program phases for statistical bug localization. In *Proceedings of the 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '13, pages 33–40, New York, NY, USA, 2013. ACM.

[MW00] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, April 2000.

[NABL] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system. *Software Testing, Verification and Reliability*, 25(4):371–396.

[NAW+08] Syeda Nessa, Muhammad Abedin, W. Eric Wong, Latifur Khan, and Yu Qi. Software fault localization using n-gram analysis. In Yingshu Li, Dung T. Huynh, Sajal K. Das, and Ding-Zhu Du, editors, *Wireless Algorithms, Systems, and Applications*, pages 548–559, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[NH17] Tanzeem Bin Noor and Hadi 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*, PROMISE, pages 2–11, New York, NY, USA, 2017. ACM.

[NTV+14] Cu Nguyen, Paolo Tonella, Tanja Vos, Nelly Condori, Bilha Mendelson, Daniel Citron, and Onn Shehory. Test prioritization based on change sensitivity: an industrial case study, 2014.

[NZZ+10] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 309–318, Nov 2010.

[PCJ+17] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 609–620, Piscataway, NJ, USA, 2017. IEEE Press.

[PL16] Rachel Potvin and Josh Levenberg. Why google stores billions of lines of code in a single repository. *Commun. ACM*, 59(7):78–87, June 2016.

[POPDL15] Annibale Panichella, Rocco Oliveto, Massimiliano Di Penta, and Andrea De Lucia. Improving multi-objective test case selection by injecting diversity in genetic algorithms. *IEEE Transactions on Software Engineering*, 41(4):358 – 383, 2015.

[PZTM13] L. Padgham, Z. Zhang, J. Thangarajah, and T. Miller. Model-based test oracle generation for automated unit testing of agent systems. *IEEE Transactions on Software Engineering*, 39(9):1230–1244, Sept 2013.

[QCR08] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 75–86, New York, NY, USA, 2008. ACM.

[Rei09] Donald G Reinertsen. *The principles of product development flow: second generation lean product development*, volume 62. Celeritas Redondo Beach, 2009.

[RHTZ13] Danijel Radjenovic, Marjan Hericko, Richard Torkar, and Ales Zivkovic. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397 – 1418, 2013.

[RPH+11] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. Bugcache for inspections: Hit or miss? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 322–331, New York, NY, USA, 2011. ACM.

[RR03] M. Renieres and S. P. Reiss. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 30–39, Oct 2003.

[RST+04] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of java programs. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 432–448, New York, NY, USA, 2004. ACM.

[SAH17] D. Suleiman, M. Alian, and A. Hudaib. A survey on prioritization regression testing test case. In *2017 8th International Conference on Information Technology (ICIT)*, pages 854–862, May 2017.

[SGG+14] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. Balancing trade-offs in test-suite reduction. In *Proceedings of the 22Nd ACM SIGSOFT*

International Symposium on Foundations of Software Engineering, FSE 2014, pages 246–256, New York, NY, USA, 2014. ACM.

[SJI$^+$10] Emad Shihab, Zhen Ming Jiang, Walid M. Ibrahim, Bram Adams, and Ahmed E. Hassan. Understanding the impact of code and process metrics on post-release defects: A case study on the eclipse project. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 4:1–4:10, New York, NY, USA, 2010. ACM.

[SMK$^+$11] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. High-impact defects: a study of breakage and surprise defects. In *SIGSOFT FSE*, 2011.

[SO05] Chad D. Sterling and Ronald A. Olsson. Automated bug isolation via program chipping. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, pages 23–32, New York, NY, USA, 2005. ACM.

[SS14] Satwinder Singh and Fategarh Sahib. Optimized test case prioritization with multi criteria for regression testing. 2014.

[SYGM15] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. Comparing and combining test-suite reduction and regression test selection. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 237–247, New York, NY, USA, 2015. ACM.

[SZKP15] Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E. Perry. An information retrieval approach for regression test prioritization based on program changes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 268–279, Piscataway, NJ, USA, 2015. IEEE Press.

[SZZ05] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[TM14] M. Tyagi and S. Malhotra. Test case prioritization using multi objective particle swarm optimizer. In *2014 International Conference on Signal Propagation and Computer Technology (ICSPCT 2014)*, pages 390–395, July 2014.

[TMH$^+$15] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto. The impact of mislabelling on the performance and interpretation of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 812–823, May 2015.

[Ves85] Iris Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459 – 494, 1985.

[WDG$^+$12] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham. Effective software fault localization using an rbf neural network. *IEEE Transactions on Reliability*, 61(1):149–169, March 2012.

[WDGL14] W. E. Wong, V. Debroy, R. Gao, and Y. Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, March 2014.

[WDX12] W. E. Wong, V. Debroy, and D. Xu. Towards better fault localization: A crosstab-based statistical approach. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(3):378–396, May 2012.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[WGL$^+$16] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, Aug 2016.

[WNT17] Song Wang, Jaechang Nam, and Lin Tan. Qtep: Quality-aware test case prioritization. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 523–534, New York, NY, USA, 2017. ACM.

[WQ06] W. Eric Wong and Yu Qi. Effective program debugging based on execution slices and inter-block data dependency. *Journal of Systems and Software*, 79(7):891 – 903, 2006. Selected papers from the 11th Asia Pacific Software Engineering Conference (APSEC2004).

[WQ09] W Eric Wong and Yu Qi. Bp neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 19(04):573–597, 2009.

[WSQM03] W. E. Wong, T. Sugeta, Yu Qi, and J. C. Maldonado. Smart debugging software architectural design in sdl. In *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003*, pages 41–47, Nov 2003.

[WWC16] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. Locus: Locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 262–273, New York, NY, USA, 2016. ACM.

[XGKS14] Zhiwei Xu, Kehan Gao, Taghi M. Khoshgoftaar, and Naeem Seliya. System regression test planning with a fuzzy expert system. *Information Sciences*, 259:532 – 543, 2014.

[XJRZ12] J. Xuan, H. Jiang, Z. Ren, and W. Zou. Developer prioritization in bug repositories. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 25–35, June 2012.

[YAB18] S. Young, T. Abdou, and A. Bener. A replication study: Just-in-time defect prediction with ensemble learning. In *2018 IEEE/ACM 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, pages 42–47, May 2018.

[YLX^{+}15] 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, Aug 2015.

[YLXS17] Xinli Yang, David Lo, Xin Xia, and Jianling Sun. Tlel: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology*, 87:206 – 220, 2017.

[YQZ12] Zunwen You, Zengchang Qin, and Zheng Zheng. Statistical fault localization using execution sequence. In *2012 International Conference on Machine Learning and Cybernetics*, volume 3, pages 899–905, July 2012.

[Zel99] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, pages 253–267, London, UK, UK, 1999. Springer-Verlag.

[Zel02] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pages 1–10, New York, NY, USA, 2002. ACM.

[ZGG] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faulty code by multiple points slicing. *Software: Practice and Experience*, 37(9):935–961.

[ZGG06] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 272–281, New York, NY, USA, 2006. ACM.

[Zho10] Z.Q. Zhou. Using coverage information to guide test case selection in adaptive random testing. In *Proceedings - International Computer Software and Applications Conference*, number Proceedings - 34th Annual IEEE International Computer Software and Applications Conference Workshops, COMPSACW 2010, pages 208–213, School of Computer Science and Software Engineering, University of Wollongong, 2010.

[ZJW$^+$14] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 385–396, New York, NY, USA, 2014. ACM.

[ZKZW06] Thomas Zimmermann, Sunghun Kim, Andreas Zeller, and E. James Whitehead, Jr. Mining version archives for co-changed lines. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 72–75, New York, NY, USA, 2006. ACM.

[ZPZ07] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 9–, Washington, DC, USA, 2007. IEEE Computer Society.

[ZR17] Celal Ziftci and Jim Reardon. Who broke the build?: Automatically identifying changes that induce test failures in continuous integration at google scale. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ICSE-SEIP '17, pages 113–122, Piscataway, NJ, USA, 2017. IEEE Press.

[ZSR18] 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, Sept 2018.

[ZZ14] Sai Zhang and Congle Zhang. Software bug localization with markov logic. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 424–427, New York, NY, USA, 2014. ACM.