Bisecting Commits and Modeling Commit Risk during Testing

Armin Najafi
Department of Computer Science and Software Engineering
Concordia University
Montréal, Québec, Canada
a_ajaf@encs.concordia.ca

Peter C. Rigby
Department of Computer Science and Software Engineering
Concordia University
Montréal, Québec, Canada
peter.rigby@concordia.ca

Weiyi Shang
Department of Computer Science and Software Engineering
Concordia University
Montréal, Québec, Canada
shang@encs.concordia.ca

ABSTRACT

Software testing is one of the costliest stages in the software development life cycle. One approach to reducing the test execution cost is to group changes and test them as a batch (*i.e.* batch testing). However, when tests fail in a batch, commits in the batch need to be re-tested to identify the cause of the failure, *i.e.* the culprit commit. The re-testing is typically done through bisection (*i.e.* a binary search through the commits in a batch). Intuitively, the effectiveness of batch testing highly depends on the size of the batch. Larger batches require fewer initial test runs, but have a higher chance of a test failure that can lead to expensive test re-runs to find the culprit. We are unaware of research that investigates and simulates the impact of batch sizes on the cost of testing in industry.

In this work, we first conduct empirical studies on the effectiveness of batch testing in three large-scale industrial software systems at Ericsson. Using 9 months of testing data, we simulate batch sizes from 1 to 20 and find the most cost-effective *BatchSize* for each project. Our results show that batch testing saves 72% of test executions compared to testing each commit individually. In a second simulation, we incorporate flaky tests that pass and fail on the same commit as they are a significant source of additional test executions on large projects. We model the degree of flakiness for each project and find that test flakiness reduces the cost savings to 42%. In a third simulation, we guide bisection to reduce the likelihood of batch-testing failures. We model the riskiness of each commit in a batch using a bug model and a test execution history model. The risky commits are tested individually, while the less risky commits are tested in a single larger batch. Culprit predictions with our approach reduce test executions up to 9% compared to Ericsson's current bisection approach.

The results have been adopted by developers at Ericsson and a tool to guide bisection is in the process of being added to Ericsson's continuous integration pipeline.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Culprit risk models, Batching and biseciton, Software testing, Empirical software engineering

ACM Reference Format:
Armin Najafi, Peter C. Rigby, and Weiyi Shang. 2019. Bisecting Commits and Modeling Commit Risk during Testing. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338944

1 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 [36]. To isolate test failures, some 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, some 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 resources. Although batch testing can reduce test executions, it introduces a new challenge. When a batch fails, the culprit commit causing the batch failure needs to be identified. One of the common approaches used for finding a culprit in a group of failing commits is bisection.

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 [51]. 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 [27]. 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. We propose risk-based approaches for calculating risk values for commits of a batch. Then we propose a TestTopK approach for testing the riskier commits individually and the rest of the less 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 [17]. Our second approach is based on using

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338944

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 allows a set of commits to be tested as a single unit. However, if a batch fails, a bisection process must be used to isolate the commit responsible for the failure, *i.e.* the culprit. A bisection involves a binary search, leading to an addition of $2 * log_2(n)$ executions to isolate a single culprit. Using three projects at Ericsson, we calculate the *CulpritRate* for each project and simulate varying *BatchSizes* to determine the most cost-effective *BatchSize* for each project.

We find that the higher the *CulpritRate* the smaller the most cost-effective *BatchSize*. For example, Project A has a *CulpritRate* 2.4 times higher than Project C and with a *BatchSize* of 4, the savings are 46%. In contrast, Project C can have a *BatchSize* up to 9 and saves 72% of test executions when compared with testing all commits. The *CulpritRate* can be trivially calculated from historical test results to determine the most cost-effective *BatchSize* for any project.

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. 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*. 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.

We find that the higher the *FlakeRate* the smaller the *BatchSize* and smaller the savings in executions. For example, Project B has a *FlakeRate* 1.37 times higher 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, it introduces a new challenge. 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, *BugModels*, and *TestExecutionHistory* models.

BugModels have been effective at identifying the commits that are most likely to lead to future bugs, *i.e.* bug introducing changes. [11, 17, 26, 28, 42, 43]. 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 [4, 6, 9, 18, 51]. 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.

We find that both culprit risk prediction models are effective, but *TestExecutionHistory* outperforms the *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. We note that if the prediction is wrong, we will simply run more test executions to find the correct culprit. Compared to *Bisection*, *TestExecutionHistory* saves 9.0% and 7.6% 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 2, we discuss the batching and bisection process used at Ericsson. In Section 3, we explain how we guide the bisection process to reduce the number of test executions to find culprits with risk models. In Section 4, we introduce our simulations methodology and data used in the study. In Section 5, we present results for each of our research questions. In Section 6, we describe the threats to validity and how we mitigate them. In Section 7, we discuss related work. In Section 8, we present our contributions and conclude the work.

2 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 the required tests. As tests are combined into a single build and tested together, batching can reduce test executions. However, when a test fails on a batch, we need additional test runs to isolate the commit(s) that is causing the failure, *i.e.* the culprit commit. One approach to isolating the culprit commit is to run bisection, which is a binary search on the commits contained in a batch.

The bisection process used at Ericsson involves splitting the commits of the batch in half as illustrated in Figure 1. 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. There can be multiple culprits and the search continues with each split until all the culprits are found.

For example, in Figure 1, 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

$$2 * log_2(n) + 1 \quad (1)$$

3 GUIDING BISECTION BASED ON CULPRIT RISK MODELS

Commits are batched and tested in the order in which they arrive (FIFO queue). However, some commits contain more risky changes than others [17]. 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 are more likely to pass without requiring bisection. In this section, we describe how we guide bisection by risk, in the subsequent sections we show how we calculate the risk using two models: *TestExecutionHistory* and *BugModel*.

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 TestTopK approach to isolate the top N riskiest commits and test them individually while batching the remaining commits into a single large batch.

Figure 2 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 1).

BugModels have been used to suggest risky files and changes [17] 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 [6], 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.

3.1 *BugModel*

BugModels have a long history in the software engineering literature. Research has predicted which files and modules are most likely to contain defects, *i.e.* which are riskiest [2, 7, 8, 10, 11, 13, 20, 25, 26, 28–32, 38, 39, 42, 43, 45, 53].

In contrast, Kamei et al. [17] 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 the action is to simply run a test, *BugModels* provide adequate information.

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 the simplest measures proposed by Kamei et al. [17], and leave more advanced *BugModels* to future work. As each commit may have multiple file changes, if a measure is related to specific files, the average of the measure 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. [17] 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 [17].
(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.

3.2 *TestExecutionHistory*

Prior work has shown that tests that have failed in the past are likely to continue failing [4, 9, 18, 51]. Preliminary work at Ericsson has shown a relationship between failing tests and the files in the change under test [6]. 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:

$$Prob(file_n, test_x) = \frac{\#file_n\_fails\_test_x}{\#total\_fails\_test_x} \quad (2)$$

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

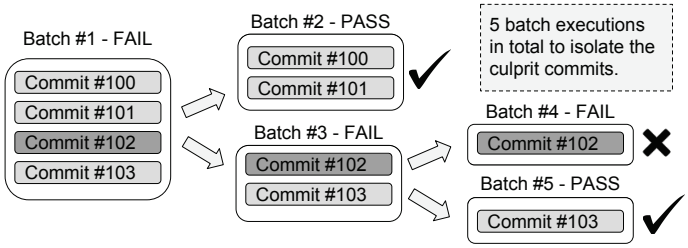

Figure 1: *Bisection*. When a batch of commits fails, a bisection is performed to find the culprit commit. Since an execution is required for each binary split, there are $2 * log_2(n) + 1$ executions required to find a culprit. To bisect 4 commits, we must run 5 executions. However, if the batch passes,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

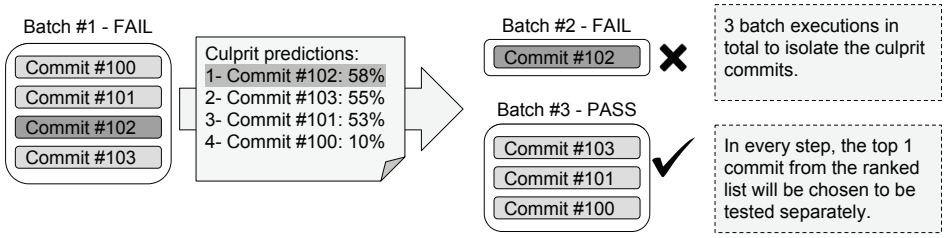

Figure 2: TestTopK. The riskiest N commits are tested individually, with the remaining commits combined in a single batch. In this case, top_1 reduces the number of required executions to three compared to the five in Figure 1.

$$Prob_{norm}(file_n, test_x) = \frac{\#line\_changes\_in\_file_n}{\#line\_changes\_in\_commit} * Prob(file_n, test_x) \quad (3)$$

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

$$culprit\_score(commit) = \sum_{n=files, x=tests} Prob_{norm}(file_n, test_x) \quad (4)$$

Figure 3 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 SIMULATION METHODOLOGY AND DATA

We evaluate three projects at Ericsson which we name A, B, and C over the period of January to September 2018. The test practices and bisection techniques have been described in Section 2. 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.

$$CulpritRate = \frac{\#CulpritCommits}{\#TotalCommits} \quad (5)$$

The respective *CulpritRate* for projects A, B, and C is 6.8%, 8.6%, and 2.8% . Project C has the fewest culprits. Project A and B have 2.4 and 3.1 times 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.

$$FlakeRate = \frac{\#FlakyBatches}{\#TotalBatches} \quad (6)$$

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 both have 1.4 times more flaky failures than Project C.

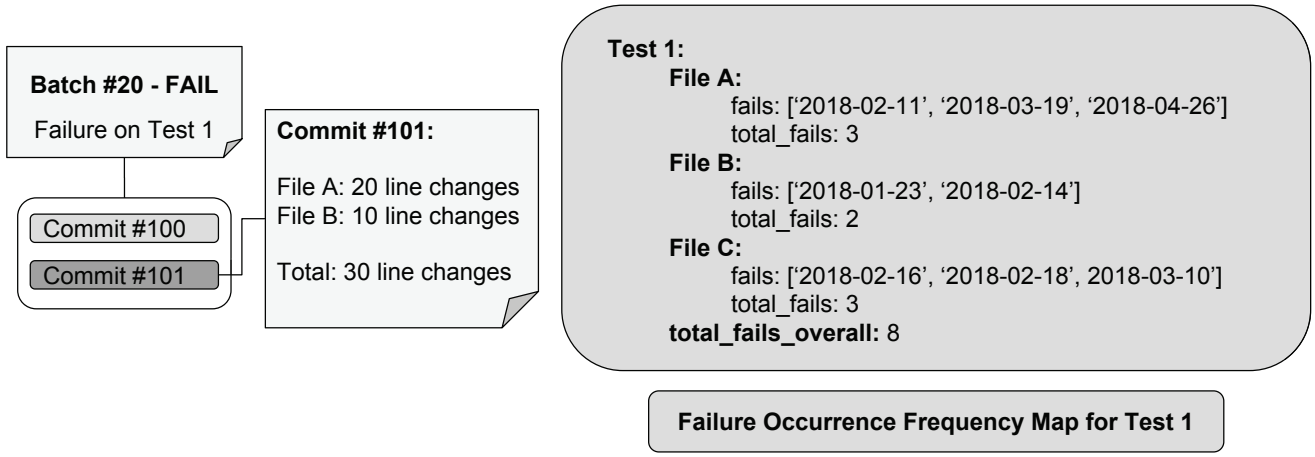

Figure 3: Calculating the commit culprit score based on file and test failure frequency.

4.1 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 4, 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.

Using this model, we correct the number of executions to include flaky failures. In Equation 7, 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 1.

$$\#FlakyExecutions = \#Batches * FlakeRate * 2 * \log_2(n) \quad (7)$$

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

$$Total\#Executions = \#CulrpitExecutions + \#FlakyExecutions \quad (8)$$

For example, for Project C, a *BatchSize* of 8 is about two times more likely to have a flaky failure than *BatchSize* of 4. We add these extra flaky executions for *BatchSize* 8.

4.2 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 [5, 14, 15, 47].

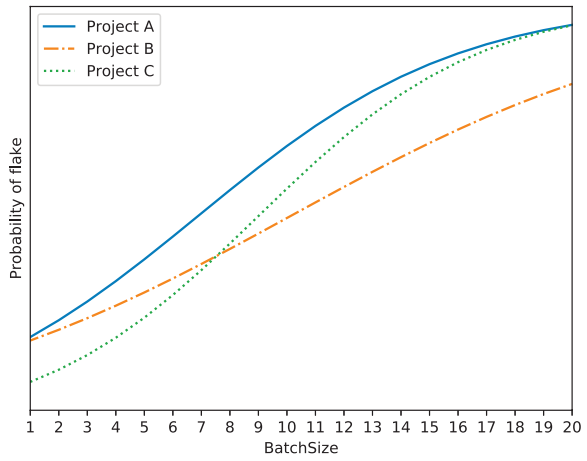

Figure 4: Probability of a flaky failure for each *BatchSize*. The probability is estimated with a logistic regression model for each project. We anonymized the y-axis at Ericsson's request.

Our simulation period runs for 9 months and covers 14k 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*.

5 RESULTS

In this section, we present the results by answering three research questions.

5.1 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 [51]. Since the vast majority of tests do not fail, testing all commits individually wastes resources. Theoretically, the lower the *CulpritRate*, the higher the *BatchSize*. We calculated the *CulpritRate* for each project and found 6.8%, 8.6%, and 2.8% culprits for A, B, and C, respectively. With these variable *CulpritRates*, we simulate the savings relative to testing all commits individually, *TestAllCommits*, for *BatchSizes* 1 through 20.

Figure 5 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 high *CulpritRates* and see similar cost-effective *BatchSizes* and savings in executions. Project A and B have more than two times as many culprits as 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* 2.4 times higher than Project C and with a *BatchSize* of 4, the savings are 46%. In contrast, Project C can have a *BatchSize* up to 9 and saves 72% of test executions when compared with testing all commits.

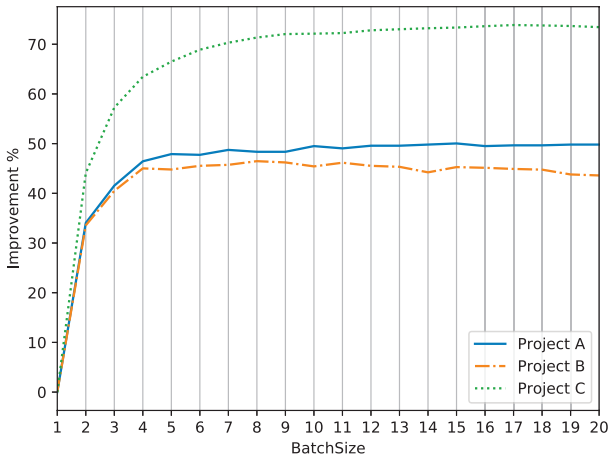

Figure 5: Improvement in test executions for different *BatchSizes*. In an ideal environment, we see a logarithmic increase with most of the savings in executions being realized before batches of size four.

5.2 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 [27]. 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 4. The *FlakeRate* will limit the most cost-effective *BatchSize*.

In Section 4.1, 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. We found that the respective *FlakeRates* for Projects A, B, and C are 23.6%, 23.1%, and 16.9%.

Figure 6 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 increase in the number of executions as flaky failures become more frequent in larger batch sizes.

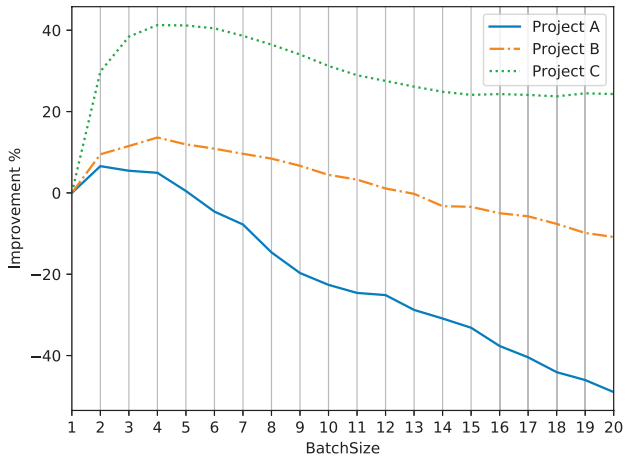

Figure 6: Improvements in test executions considering the *FlakeRates*. The *FlakeRate* controls the *BatchSize* and the project with the highest flake rate does not see any advantage above *BatchSize* = 2. Project C still attains high execution savings, at 41% with a *BatchSize* of 4.

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%. Project C has 37% fewer flakes than the other projects and has a 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 6, 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 most cost-effective 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 *FlakeRate* 1.37 times higher 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.

5.3 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 2, we use Figure 2 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 1 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 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 =

$$\frac{NumCorrectCulpritPredictionsAtK}{TotalCulprits} \tag{9}$$

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 =

$$1 - \frac{NumExecutionsBisection}{NumExecRisk} \tag{10}$$

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

5.3.1 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 . . . 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.

5.3.2 Results for culprit risk prediction. The results of our analysis are shown in Table 1. 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 5.0% and 2.6% more total executions than *Bisection*, 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 *Bisection*.

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 *Bisection* 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 SufficientAndCorrectAt2 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 *Bisection* 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 accuracy at Top2 are sufficient to reduce the number of executions when compared with *Bisection*. We combined the *BugModel* and the *TestExecutionHistory* model, but noted a reduction in accuracy and savings.

> Both culprit risk prediction models are effective, but *TestExecutionHistory* outperforms the *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 *Bisection* this results in 9.0% and 7.6% fewer executions, respectively.

6 THREATS TO VALIDITY

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

6.1 External 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 *CulpritRate* and *FlakeRate*, Project A has twice as many culprits as C and Project B has 1/3 more flaky failures. A recent report from Facebook [12] shows that in practice, testing approaches should start with the assumption that all tests are flaky. Since 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, replicating our study on other system may help further understand the generalizability of our findings.

6.2 Construct validity

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*.

As explained in Section 4.2, 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 paper 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 because of the reduced the quality of the culprit risk predictions.

6.3 Internal validity

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.

7 RELATED WORK

In this section, we discuss the prior research that is related to this paper.

7.1 Batch testing and bisection

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

Table 1: We are able to reduce the number of executions relative to normal bisection by 9% and 7.6% for Projects B and C. Note: since there can be multiple culprits in a batch, TestTop1 is often not sufficient to find all the culprits. Note: Project A is excluded because its most cost-effective *BatchSize* is 2, so Top1 = Top2 = TestAll. Note: we only display Top1 and Top2 because with a *BatchSize* of 4, Top3 = Top4 = TestAll.

	$TopK = 1$		$TopK = 2$	
Project B	Sufficient and Correct	Difference in Executions	Sufficient and Correct	Difference in Executions
BugModel	22%	5.0%	59%	-7.4%
TestFile	33%	0.7%	**63%**	**-9.0%**
Project C	Sufficient and Correct	Difference in Executions	Sufficient and Correct	Difference in Executions
BugModel	34%	2.6%	59%	-4.3%
TestFile	46%	-5.3%	**66%**	**-7.6%**

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 [40].

One of the common approaches used for finding a culprit in a group of failing commits is bisection. When commits are ordered, GitBisection [1] 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) [52].

At Ericsson each commit under test should be independent of the other commits. As a result, 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 2). 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 *CulpritRate* and the *FlakeRate*. We then guide bisection by using historical models, instead of statistical dependencies.

7.2 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 [16, 21–24, 33–35, 37, 41, 44, 46]. The Google culprit finder [52] 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 [18]. Building on these ideas, researchers have developed sliding windowed predictions [9], used association rule mining [4], and test co-failure distributions [51]. 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 [6]. 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.

7.3 Bug models

Recent works have extensively 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 [2, 7, 8, 10, 11, 13, 20, 25, 26, 28–32, 38, 39, 42, 43, 45, 53].

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. [19] 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, and complexity metrics from software revision history. They 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. [17] 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. [49] 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 [17], 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. [48] propose another similar study for predicting defective changes using a two-layer ensemble learning approach. Young et al. [50] propose a replication of this study.

What all these studies have in common is predicting the buggy commits as early as possible, *i.e.* 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. Recent study by Ananthanarayanan et al. [3] aims to build prediction models for prevent a commit from breaking the build of software. Such prediction models may be adopted in our approach to further improve the our prediction of culprit commits.

8 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. This paper is the first work to study the most cost-effective *BatchSize* based on the number of true test failures, *CulpritRate*. 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 provide a fundamental insight into the cost of batching and an actionable use for commit risk prediction models. We make three major contributions:

(1) We find the higher the *CulpritRate* 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 *CulpritRate*, can optimally use *BatchSize* = 9 and have savings above 72% of executions over testing all commits individually.
(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 *TestExecutionHistory* model achieves an average SufficientAndCorrectAt2 of 64.5% and outperforms the *BugModel*. By using these risk predictions compared to *Bisection* we need 7.6% to 9.0% fewer executions.

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.

ACKNOWLEDGEMENT

We would like to thank Ericsson for providing access to their system used in our study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of Ericsson and/or its subsidiaries and affiliates. Moreover, our results do not reflect the quality of Ericsson's products.

REFERENCES

[1] [n. d.]. bisect - https://git-scm.com/docs/gitbisect. ([n. d.]). https://git-scm.com/docs/gitbisect
[2] Fumio Akiyama. 1971. An Example of Software System Debugging.. In *IFIP Congress (1)*, Vol. 71. 353–359.
[3] Sundaram Ananthanarayanan, Masoud Saeida Ardekani, Denis Haenikel, Balaji Varadarajan, Simon Soriano, Dhaval Patel, and Ali-Reza Adl-Tabatabai. 2019. Keeping Master Green at Scale. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. ACM, New York, NY, USA, Article 29, 15 pages. https://doi.org/10.1145/3302424.3303970
[4] Jeff Anderson, Saeed Salem, and Hyunsook Do. 2014. Improving the Effectiveness of Test Suite Through Mining Historical Data. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, New York, NY, USA, 142–151. https://doi.org/10.1145/2597073.2597084
[5] P. Bhattacharya and I. Neamtiu. 2010. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *2010 IEEE International Conference on Software Maintenance*. 1–10. https://doi.org/10.1109/ICSM.2010.5609736
[6] M. Campbell, K. Martin, F. Bozóki, and M. Atkinson. 2017. Dynamic Test Selection Using Source Code Changes. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 597–598. https://doi.org/10.1109/QRS-C.2017.111
[7] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. 2009. Software Dependencies, Work Dependencies, and Their Impact on Failures. *IEEE Transactions on Software Engineering* 35, 6 (Nov 2009), 864–878. https://doi.org/10.1109/TSE.2009.42
[8] M. D'Ambros, M. Lanza, and R. Robbes. 2010. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 31–41. https://doi.org/10.1109/MSR.2010.5463279
[9] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. 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)*. ACM, New York, NY, USA, 235–245. https://doi.org/10.1145/2635868.2635910
[10] T. Gyimothy, R. Ferenc, and I. Siket. 2005. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering* 31, 10 (Oct 2005), 897–910. https://doi.org/10.1109/TSE.2005.112
[11] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. 2012. A Systematic Literature Review on Fault Prediction Performance in Software Engineering. *IEEE Transactions on Software Engineering* 38, 6 (Nov 2012), 1276–1304. https://doi.org/10.1109/TSE.2011.103
[12] M. Harman and P. O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 1–23. https://doi.org/10.1109/SCAM.2018.00009
[13] A. E. Hassan. 2009. Predicting faults using the complexity of code changes. In *2009 IEEE 31st International Conference on Software Engineering*. 78–88. https://doi.org/10.1109/ICSE.2009.5070510

[14] K. Herzig and N. Nagappan. 2015. Empirically Detecting False Test Alarms Using Association Rules. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 39–48. https://doi.org/10.1109/ICSE.2015.133

[15] He Jiang, Xiaochen Li, Zijiang Yang, and Jifeng Xuan. 2017. 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)*. IEEE Press, Piscataway, NJ, USA, 712–723. https://doi.org/10.1109/ICSE.2017.71

[16] R. Just, G. M. Kapfhammer, and F. Schweiggert. 2012. 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*. 11–20. https://doi.org/10.1109/ISSRE.2012.31

[17] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (June 2013), 757–773. https://doi.org/10.1109/TSE.2012.70

[18] Jung-Min Kim and Adam Porter. 2002. 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)*. ACM, New York, NY, USA, 119–129. https://doi.org/10.1145/581339.581357

[19] S. Kim, E. J. Whitehead Jr., and Y. Zhang. 2008. Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering* 34, 2 (March 2008), 181–196. https://doi.org/10.1109/TSE.2007.70773

[20] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. 2007. Predicting Faults from Cached History. In *29th International Conference on Software Engineering (ICSE'07)*. 489–498. https://doi.org/10.1109/ICSE.2007.66

[21] P. Konsaard and L. Ramingwong. 2015. 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)*. 1–6. https://doi.org/10.1109/ECTICon.2015.7207103

[22] N. Kukreja, W. G. J. Halfond, and M. Tambe. 2013. Randomizing regression tests using game theory. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 616–621. https://doi.org/10.1109/ASE.2013.6693122

[23] Manoj Kumar, Arun Sharma, and Rajesh Kumar. [n. d.]. 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 ([n. d.]), 949–971. https://doi.org/10.1002/spe.2263 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2263

[24] Mohsen Laali, Huai Liu, Margaret Hamilton, Maria Spichkova, and Heinz W. Schmidt. 2016. Test Case Prioritization Using Online Fault Detection Information. In *Reliable Software Technologies – Ada-Europe 2016*, Marko Bertogna, Luis Miguel Pinho, and Eduardo Quiñones (Eds.). Springer International Publishing, Cham, 78–93.

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

[26] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2014. 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)*. ACM, New York, NY, USA, 192–201. https://doi.org/10.1145/2597073.2597076

[27] John Micco. 2016. Flaky Tests at Google and How We Mitigate Them. https://goo.gl/rxFGiw. (2016).

[28] Audris Mockus. 2010. Organizational Volatility and Its Effects on Software Defects. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, New York, NY, USA, 117–126. https://doi.org/10.1145/1882291.1882311

[29] A. Mockus and D. M. Weiss. 2000. Predicting risk of software changes. *Bell Labs Technical Journal* 5, 2 (April 2000), 169–180. https://doi.org/10.1002/bltj.2229

[30] R. Moser, W. Pedrycz, and G. Succi. 2008. 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*. 181–190. https://doi.org/10.1145/1368088.1368114

[31] J. C. Munson and T. M. Khoshgoftaar. 1992. The detection of fault-prone programs. *IEEE Transactions on Software Engineering* 18, 5 (May 1992), 423–433. https://doi.org/10.1109/32.135775

[32] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. 2010. Change Bursts as Defect Predictors. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*. 309–318. https://doi.org/10.1109/ISSRE.2010.25

[33] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. [n. d.]. Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system. *Software Testing, Verification and Reliability* 25, 4 ([n. d.]), 371–396. https://doi.org/10.1002/stvr.1572 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1572

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

[35] Tanzeem Bin Noor and Hadi Hemmati. 2017. 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)*. ACM, New York, NY, USA, 2–11. https://doi.org/10.1145/3127005.3127006

[36] L. Padgham, Z. Zhang, J. Thangarajah, and T. Miller. 2013. Model-Based Test Oracle Generation for Automated Unit Testing of Agent Systems. *IEEE Transactions on Software Engineering* 39, 9 (Sept 2013), 1230–1244. https://doi.org/10.1109/TSE.2013.10

[37] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. 2008. 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)*. ACM, New York, NY, USA, 75–86. https://doi.org/10.1145/1390630.1390641

[38] Danijel Radjenovic, Marjan Hericko, Richard Torkar, and Ales Zivkovic. 2013. Software fault prediction metrics: A systematic literature review. *Information and Software Technology* 55, 8 (2013), 1397 – 1418. https://doi.org/10.1016/j.infsof.2013.02.009

[39] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. 2011. 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)*. ACM, New York, NY, USA, 322–331. https://doi.org/10.1145/2025113.2025157

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

[41] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. 2015. 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)*. ACM, New York, NY, USA, 237–247. https://doi.org/10.1145/2786805.2786878

[42] Emad Shihab, Zhen Ming Jiang, Walid M. Ibrahim, Bram Adams, and Ahmed E. Hassan. 2010. 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)*. ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/1852786.1852792

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

[44] Satwinder Singh and Fategarh Sahib. 2014. Optimized Test Case Prioritization with multi criteria for Regression Testing.

[45] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto. 2015. The Impact of Mislabelling on the Performance and Interpretation of Defect Prediction Models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 812–823. https://doi.org/10.1109/ICSE.2015.93

[46] Song Wang, Jaechang Nam, and Lin Tan. 2017. QTEP: Quality-aware Test Case Prioritization. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 523–534. https://doi.org/10.1145/3106237.3106258

[47] J. Xuan, H. Jiang, Z. Ren, and W. Zou. 2012. Developer prioritization in bug repositories. In *2012 34th International Conference on Software Engineering (ICSE)*. 25–35. https://doi.org/10.1109/ICSE.2012.6227209

[48] Xinli Yang, David Lo, Xin Xia, and Jianling Sun. 2017. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology* 87 (2017), 206 – 220. https://doi.org/10.1016/j.infsof.2017.03.007

[49] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. 2015. Deep Learning for Just-in-Time Defect Prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*. 17–26. https://doi.org/10.1109/QRS.2015.14

[50] S. Young, T. Abdou, and A. Bener. 2018. 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)*. 42–47.

[51] Y. Zhu, E. Shihab, and P. C. Rigby. 2018. Test Re-Prioritization in Continuous Testing Environments. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 69–79. https://doi.org/10.1109/ICSME.2018.00016

[52] Celal Ziftci and Jim Reardon. 2017. Who Broke the Build?: Automatically Identifying Changes That Induce Test Failures in Continuous Integration at Google Scale. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '17)*. IEEE Press, Piscataway, NJ, USA, 113–122. https://doi.org/10.1109/ICSE-SEIP.2017.13

[53] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting Defects for Eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering (PROMISE '07)*. IEEE Computer Society, Washington, DC, USA, 9–. https://doi.org/10.1109/PROMISE.2007.10